

# **ZHI# – PROGRAMMING LANGUAGE INHERENT SUPPORT FOR ONTOLOGIES**

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Fridericiana zu Karlsruhe (TH)

**genehmigte**

**Dissertation**

von

**Alexander Paar**

aus Bad Wildungen

Tag der mündlichen Prüfung: 21.07.2009

Erster Gutachter: Prof. Dr. Walter F. Tichy

Zweiter Gutachter: Prof. Dr. Peter H. Schmitt







# TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Theses	9
1.2	Position on the Market	10
1.3	Conceptual Modeling with the Web Ontology Language	11
1.3.1	From networks to Description Logics	11
1.3.2	The $\mathcal{SHOIN}(\mathbf{D})$ Description Logic	18
1.3.3	XSD – XML Schema Definition	25
1.3.4	RDF – The Resource Description Framework	28
1.3.5	RDFS – The Resource Description Framework Schema	32
1.3.6	OWL – The Web Ontology Language	34
1.3.7	The CHIL OWL DL Ontology	37
1.4	Interpretation of Object-Oriented Programming Languages	41
1.4.1	The untyped lambda-calculus	42
1.4.2	The simply typed lambda-calculus	46
1.4.3	The simply typed lambda-calculus with subtyping	48
1.4.4	Properties of typing and subtyping	50
<b>2</b>	<b>The Zhi# Compiler Framework</b>	<b>53</b>
2.1	The Zhi# Programming Language	54
2.2	Architecture and Implementation	56
2.3	Framework Extension Points	63
2.3.1	Typing extension point	64

2.3.2	Program transformation extension point . . . . .	65
2.4	Type System Cooperation . . . . .	73
2.5	Related Work . . . . .	76
2.6	Summary . . . . .	78
<b>3</b>	<b>The <math>\lambda_C</math>-Calculus . . . . .</b>	<b>81</b>
3.1	Facets . . . . .	82
3.1.1	Fundamental Facets . . . . .	82
3.1.2	Constraining Facets . . . . .	83
3.2	Type Derivation . . . . .	87
3.3	Subtyping . . . . .	89
3.4	Properties of the $\lambda_C$ -Type System . . . . .	91
3.5	Type Inference . . . . .	92
3.6	Implementation . . . . .	94
3.7	Constraint Arithmetic . . . . .	97
3.8	Related Work . . . . .	101
3.9	Summary . . . . .	102
<b>4</b>	<b>The CHIL OWL API . . . . .</b>	<b>105</b>
4.1	The CHIL Knowledge Base Server . . . . .	107
4.1.1	Architectural model . . . . .	108
4.1.2	The CHIL OWL API . . . . .	113
4.2	A Formally Specified OWL API . . . . .	120
4.2.1	Notational framework . . . . .	122
4.2.2	Examples . . . . .	126

4.2.3	The CHIL OWL API testing framework . . . . .	128
4.3	Example Scenario Implementation . . . . .	133
4.4	Related Work . . . . .	134
4.4.1	Off-the-shelf ontology management systems . . . . .	135
4.4.2	Knowledge base interface specifications . . . . .	137
4.5	Summary . . . . .	138
<b>5</b>	<b>XSD Aware Compilation – Types and... Constraints . . . . .</b>	<b>141</b>
5.1	Integrating XSD with the C# Programming Language . . . . .	142
5.1.1	Referencing XML Schema Definitions . . . . .	143
5.1.2	Static typing . . . . .	145
5.1.3	Type inference . . . . .	156
5.1.4	Compilation to C# . . . . .	165
5.1.5	Dynamic checking . . . . .	168
5.2	Related Work . . . . .	170
5.3	Summary . . . . .	172
<b>6</b>	<b>OWL Aware Compilation – Complex Data, Simple Code . . . . .</b>	<b>175</b>
6.1	Integrating OWL DL with the C# Programming Language . . . . .	178
6.1.1	Referencing OWL DL Ontologies . . . . .	178
6.1.2	Static typing . . . . .	181
6.1.3	Auxiliary properties and methods . . . . .	195
6.1.4	Compilation to C# . . . . .	197
6.1.5	Dynamic checking . . . . .	200
6.2	Example Scenario Implementation . . . . .	207

6.3	OWL and XSD . . . . .	209
6.4	Integration of the CHIL OWL API . . . . .	209
6.5	Related Work . . . . .	211
6.6	Summary . . . . .	215
<b>7</b>	<b>Validation and Evaluation . . . . .</b>	<b>217</b>
7.1	Technical Validation . . . . .	218
7.2	Microscopic Evaluation . . . . .	222
7.2.1	XML data types . . . . .	223
7.2.2	XML APIs vs. XML data types . . . . .	228
7.2.3	OWL concept constructors . . . . .	231
7.2.4	OWL role constructors and restrictions . . . . .	251
7.2.5	OWL APIs vs. OWL types . . . . .	264
7.3	Macroscopic Evaluation . . . . .	280
7.3.1	OCL invariants in Zhi# . . . . .	282
<b>8</b>	<b>Conclusion and Outlook . . . . .</b>	<b>291</b>
8.1	Conclusion . . . . .	291
8.2	Outlook . . . . .	299
<b>A</b>	<b>The Zhi# Compiler Application . . . . .</b>	<b>303</b>
<b>B</b>	<b>Zhi# External Program Transformation Functions . . . . .</b>	<b>306</b>
B.1	XSD Program Transformation Functions . . . . .	307
B.2	OWL Program Transformation Functions . . . . .	312
<b>C</b>	<b>The <math>\lambda_C</math>-Calculus . . . . .</b>	<b>321</b>



<b>D</b>	<b>XSD Type Inference Rules</b>	<b>327</b>
<b>E</b>	<b>XSD Compile-Time Arithmetic</b>	<b>329</b>
E.1	XSD Type Arithmetic	329
E.2	XSD Constraint Arithmetic	332
<b>F</b>	<b>CHIL OWL API Preconditions</b>	<b>335</b>
<b>G</b>	<b>The CHIL OWL API</b>	<b>342</b>
G.1	The <i>ITellingABox</i> Interface	342
G.2	The <i>IAskingABox</i> Interface	350
G.3	The <i>ITellingTBox</i> Interface	354
G.4	The <i>IAskingTBox</i> Interface	368
<b>H</b>	<b>Mapping of the CHIL OWL API</b>	<b>377</b>
H.1	The <i>ITellingABox</i> Interface	378
H.2	The <i>IAskingABox</i> Interface	382
H.3	The <i>IAskingTBox</i> Interface	384
<b>I</b>	<b>Zhi# Syntax and Semantics</b>	<b>387</b>
I.1	Syntax	388
I.2	Semantics	390
I.2.1	XML Schema Definition	391
I.2.2	Web Ontology Language	401
	<b>References</b>	<b>407</b>

## LIST OF FIGURES

1.1	Two intersecting class hierarchies in a software system . . . . .	2
1.2	Position on the market . . . . .	10
1.3	An example network . . . . .	13
1.4	The RDF triple . . . . .	28
1.5	A graph of two RDF statements . . . . .	28
1.6	An example UML diagram . . . . .	33
1.7	The Semantic Web stack . . . . .	37
2.1	Zhi# compiler framework . . . . .	57
2.2	Zhi# compiler plug-in . . . . .	58
2.3	Compilation of Zhi# programs . . . . .	59
2.4	Execution of Zhi# programs . . . . .	60
2.5	Zhi# compiler framework (class level) . . . . .	61
3.1	Architecture of the $\lambda_C$ -type system implementation . . . . .	95
3.2	$\lambda_C$ -type pool . . . . .	97
3.3	$\lambda_C$ -constraint arithmetic algorithm . . . . .	100
4.1	CHIL Knowledge Base Server components . . . . .	108
4.2	CHIL Knowledge Base Server stand-alone application . . . . .	111
4.3	CHIL Knowledge Base Server Eclipse plug-in . . . . .	111
4.4	CHIL Knowledge Base Server architectural model . . . . .	112
4.5	CHIL OWL API meta-level schema . . . . .	114
4.6	CHIL OWL API object-level schema . . . . .	120

4.7	CHIL OWL API Hoare triple schema . . . . .	123
4.8	Basic precondition PC-OWLAPI-DECLARED-CONCEPT . . . . .	125
4.9	Formally specified CHIL OWL API methods . . . . .	126
4.10	Application of the Floyd-Hoare pre-strengthening rule . . . . .	127
4.11	Application of the Floyd-Hoare sequencing rule . . . . .	128
4.12	CHIL OWL API test case generation . . . . .	129
4.13	CHIL OWL API testing framework . . . . .	132
5.1	XSD aware compilation . . . . .	143
5.2	Autocompletion of XSD types . . . . .	145
5.3	Implicit subtype relationship between XSD types . . . . .	147
5.4	XML data types in Zhi#programs . . . . .	148
5.5	XSD validation . . . . .	169
6.1	OWL aware compilation . . . . .	179
6.2	Protégé individual editor . . . . .	193
7.1	Code base size . . . . .	218
7.2	Zhi# compiler code base size . . . . .	218
7.3	Code base complexity . . . . .	219
7.4	Compilation time . . . . .	221
7.5	Context class for attribute invariants . . . . .	284
A.1	Zhi# server application . . . . .	304
A.2	Zhi# Eclipse-based frontend . . . . .	305

## LIST OF TABLES

1.1	The basic Description Logic $\mathcal{AL}$ . . . . .	20
1.2	The Description Logic $\mathcal{ALC}_{\mathcal{R}^+}$ , extends $\mathcal{AL}$ (Table 1.1) . . . . .	21
1.3	The Description Logic $\mathcal{SHIF}$ , extends $\mathcal{ALC}_{\mathcal{R}^+}$ (Table 1.2) . . . . .	22
1.4	The Description Logic $\mathcal{SHOIN}(\mathbf{D})$ . . . . .	23
1.4	The Description Logic $\mathcal{SHOIN}(\mathbf{D})$ . . . . .	24
1.5	XSD constraining facets . . . . .	26
1.6	Untyped lambda-calculus ( $\lambda$ ) . . . . .	42
1.7	Simply typed lambda-calculus ( $\lambda_{\rightarrow}$ ) . . . . .	47
1.8	Simply typed lambda-calculus with subtyping ( $\lambda_{<:}$ ) . . . . .	49
2.1	Zhi# comparison operators . . . . .	55
2.2	The <i>IEExternalTypeSystem</i> extension point . . . . .	64
2.3	The <i>IEExternalCompiler</i> extension point . . . . .	66
3.1	Terminology of the $\lambda_C$ -calculus . . . . .	82
3.2	Comparison operators for XSD . . . . .	84
3.3	S-CSTRVSPACE . . . . .	85
3.4	S-CSTRWIDTH . . . . .	85
3.5	S-CSTRDEPTH . . . . .	86
3.6	TD-CSTRAPP . . . . .	87
3.7	TD-SUBS . . . . .	87
3.8	S-VSPACE . . . . .	90
3.9	S-WIDTH . . . . .	90

3.10	S-DEPTH	90
3.11	S-APP	91
3.12	TI-IFADD	93
3.13	TI-ASSIGNREM	93
4.1	CHIL OWL API element interpretations	115
4.2	CHIL OWL API exceptions	117
4.3	Hoare logic axioms and rules	122
4.3	Hoare logic axioms and rules	123
4.4	<i>SHOIN</i> ( <b>D</b> ) metavariables	124
4.5	Off-the-shelf ontology management systems	135
5.1	Isomorphic mappings between .NET and XSD types	147
5.2	TI-IFADD	156
5.3	TI-ASSIGNREM	156
5.4	TI-FORADD	158
5.5	TI-WHILEADD	158
5.6	XSD built-in primitive types	165
7.1	TBox used for comparison	265
C.1	Constrained types calculus ( $\lambda_C$ ) syntax	321
C.1	Constrained types calculus ( $\lambda_C$ ) syntax	322
C.2	Constrained types calculus ( $\lambda_C$ ) evaluation	323
C.3	Constrained types calculus ( $\lambda_C$ ) typing	324
C.4	Constrained types calculus ( $\lambda_C$ ) subtyping	325

C.4	Constrained types calculus ( $\lambda_C$ ) subtyping . . . . .	326
C.5	Constrained types calculus ( $\lambda_C$ ) type derivation and substitution . . . . .	326
D.1	TI-IFADD . . . . .	327
D.2	TI-FORADD . . . . .	327
D.3	TI-WHILEADD . . . . .	327
D.4	TI-ASSIGNREM . . . . .	328
E.1	XSD type arithmetic . . . . .	329
E.1	XSD type arithmetic . . . . .	330
E.1	XSD type arithmetic . . . . .	331
E.2	XSD constraint materialization rules . . . . .	332
E.3	XSD constraint arithmetic rules . . . . .	333
E.3	XSD constraint arithmetic rules . . . . .	334

## DANKSAGUNG

Diese Dissertation entstand während meiner Zeit als wissenschaftlicher Mitarbeiter und Doktorand am Lehrstuhl Programmiersysteme am Institut für Programmstrukturen und Datenorganisation (IPD) der Universität Karlsruhe (TH).

Ich bin meinem Doktorvater Professor Walter F. Tichy aufrichtig dankbar für sein Vertrauen in meine Arbeit, für seine Geduld mit deren Fertigstellung und für lehrreiche und vertrauensvolle Gespräche, die ich über den Nutzen für meine Arbeit hinaus als nicht selbstverständlich sehr zu schätzen weiß.

Die endgültige Fertigstellung verdankt diese Arbeit der Bereitschaft von Professor Peter H. Schmitt als Korreferent zur Verfügung zu stehen. Seine detaillierten Anmerkungen und wohlwollenden Worte waren mir eine große Hilfe. Für die Begutachtung meiner Arbeit durch Professor Peter H. Schmitt bin ich außerordentlich dankbar.

Eine erfolgreiche Disputation wurde mir ermöglicht durch Professor Rudi Studer und Professor Ralf H. Reussner, die als Prüfer zur Verfügung standen, und Professor Frank Bellosa als Mitglied des Promotionsausschusses der Fakultät für Informatik. Ich danke Herrn Dekan Professor Heinz Wörn für die Leitung meiner Promotionsprüfung.

Auf aktuelle Semantic Web Technologien wurde ich erstmals aufmerksam gemacht durch einen von Marc Schanne gehaltenen Vortrag im Rahmen einer Lehrstuhlklausurtagung auf Schloß Dagstuhl im Oktober 2002.

Für zuverlässige und gewissenhafte Beiträge zu meiner Arbeit und für das Erstellen von Technologie-Demonstratoren danke ich sehr den hilfswissenschaftlichen Mitarbeitern Tuan Kiet Bui, Stefan Mirevski, Ivan Popov und Jing Zhi Yue.

Meine Arbeit ist ein Beitrag zu dem integrierten europäischen Forschungsprojekt “Computers in the Human Interaction Loop” (CHIL). Ich danke den Projekt-Kollegen von allen 15 beteiligten Forschungseinrichtungen für hilfreiche Anmerkungen und die engagierte und kollegiale Zusammenarbeit, die das CHIL-Konsortium zu allen Zeiten aus-

gezeichnet hat. Besonders danke ich Herrn Gerhard Sutschet und Dr. Kym Watson vom Fraunhofer-Institut für Informations- und Datenverarbeitung (IITB) in Karlsruhe für ihre Hilfestellungen bei der Durchführung des CHIL-Projekts. Besonderer Dank gebührt ebenfalls Dr. John Soldatos und Kostas Stamatis vom Athens Information Technology für eine großartige Zusammenarbeit beim Einsatz meiner Arbeit im Rahmen des CHIL-Projekts und der Durchführung von gemeinsamen Technologie-Demonstrationen im Rahmen der CHIL Technology Days. Für seine Kommentare über die Programmierbarkeit von ontologischen Wissensbasen danke ich Dr. Jan Kleindienst, IBM Prag. Seine Anmerkungen im Rahmen eines CHIL-Projekttreffens waren letztlich der Auslöser für meine Arbeit.

Besonders dankbar bin ich den Veranstaltern des International Workshop on Software Language Engineering 2007, Professor Dragan Gašević von der Athabasca University in Kanada und Professor Ralf Lämmel, damals Forscher bei Microsoft Research, für ihre ermutigenden Kommentare und die Begleitung meiner Arbeit über den Workshop hinaus.

Meine Arbeit und meine Zeit als wissenschaftlicher Mitarbeiter wurden begleitet und erleichtert von der großen Hilfsbereitschaft meines Kollegen Jürgen Reuter, die über eine übliche kollegiale Zusammenarbeit immer weit hinausgereicht hat.

Mein Werdegang mit Studium der Informatik und schließlich der Promotion wurde mir ermöglicht von meinem Großvater und bestem Freund Karl Kirchner und meiner fürsorglichen Großmutter Edith Kirchner. Ihnen habe ich alles zu verdanken. Ich bin meiner Mutter Eva-Gabriele Lamsbach-Paar zutiefst dankbar für ihre selbstlose Unterstützung.

Ich bin schließlich unendlich dankbar für die Unterstützung meiner lieben Frau Agnes. Als Fachfrau im Bereich der angewandten Informatik und des wissenschaftlichen Arbeitens verdanke ich ihr viele wertvolle Ratschläge. Ich danke ihr für ihr Verständnis für die vielen Abende wo ich versprach, früher vom Institut nach Hause zu kommen und es dann doch nicht tat. Ich danke ihr für ihren Zuspruch für meine Arbeit und ihre lieben Worte, die mir stets Ermutigung und Ansporn zugleich waren. Liebe Agnes, ohne Deine Hilfe hätte ich das Erreichte niemals geschafft.



## VITA

- 10/2002–12/2008    Wissenschaftlicher Mitarbeiter am Institut für Programmstrukturen und Datenorganisation an der Universität Fridericiana zu Karlsruhe (TH).
- 10/1999–07/2002    Hauptstudium Informatik (Diplom) an der Universität Fridericiana zu Karlsruhe (TH).
- 08/2001–04/2002    Gaststudent an der University of California, Irvine, USA.
- 10/1997–09/1999    Grundstudium Informatik (Vordiplom) an der Technischen Universität Clausthal-Zellerfeld.
- 07/1996–06/1997    Grundwehrdienst beim Fallschirmpanzerabwehrbataillon 262 in Merzig/Saar und an der Luftlande- und Lufttransportschule Altenstadt in Oberbayern.
- 1996                 Abitur am Gustav Stresemann Gymnasium in Bad Wildungen.
- 1977                 Geboren am 21. Mai 1977 in Bad Wildungen, Deutschland.

## AWARDS

- 11/2008             Silber- und Bronzemedaille beim ACM ICPC South Western Europe Regional Contest als Trainer der Teams *KAmaleon* und *Keine Ahnung*.
- 07/2001             Ernennung zum Microsoft Certified Trainer.
- 06/2001             Zweiter Platz bei der IEEE Computer Society International Design Competition.
- 07/2000–07/2002    Stipendium des Freundeskreises der Fakultät für Informatik an der Universität Karlsruhe (TH).



## PUBLICATIONS

*Searching and Using External Types in an Extensible Software Development Environment* 2<sup>nd</sup> International Workshop on Search-driven development: Users, Infrastructure, Tools, and Evaluation (SUITE 2010), Cape Town, South Africa, 2010.

*Ontological Modeling and Reasoning*. Computers in the Human Interaction Loop, Human-Computer Interaction Series, Springer Verlag London, 2009.

*Zhi# – Programming Language Inherent Support for Ontologies*. 4<sup>th</sup> International Workshop on Software Language Engineering (ateM 2007), Nashville, TN, USA, 2007.

*A Formally Specified Ontology Management API as a Registry for Ubiquitous Computing Systems*. The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies (Applied Intelligence), Springer Verlag, 2007.

*Programming Language Inherent Support for Constrained XML Schema Definition Data Types and OWL DL*. 21<sup>st</sup> IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, 2006.

*A Formally Specified Ontology Management API as a Registry for Ubiquitous Computing Systems*. 3<sup>rd</sup> IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI 2006), Athens, Greece, 2006.

*A Pluggable Architectural Model and a Formally Specified Programming Language Independent API for an Ontological Knowledge Base Server*. 1<sup>st</sup> Australasian Ontology Workshop (AOW 2005), Sydney, Australia, 2005.

*An Ontology-based Framework for Dynamic Resource Management in Ubiquitous Computing Environments*. 2<sup>nd</sup> International Conference on Embedded Software and Systems (ICESS 2005), Xi'an, P.R. China, 2005.

*Zhi#*: *Programming Language Inherent Support for XML Schema Definition*. 9<sup>th</sup> IASTED International Conference on Software Engineering and Applications (SEA 2005), Phoenix, AZ, USA, 2005.

*Semantic Software Engineering Tools*. 18<sup>th</sup> Annual ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2003), Anaheim, CA, USA, 2003.

ABSTRACT

# ZHI# – PROGRAMMING LANGUAGE INHERENT SUPPORT FOR ONTOLOGIES

by

**Alexander Paar**

Universität Karlsruhe (TH), 2009

Widely used object-oriented programming languages such as Java or C# include a built-in static type system. Programming language type systems provide a conceptual framework that makes it particularly easy to design, understand, and maintain object-oriented systems. However, with the emergence of a variety of schema and ontology languages such as XML Schema Definition (XSD) or the Web Ontology Language (OWL) conventional built-in programming language type systems have reached their limits. In XSD, atomic types can be derived through the application of value space constraints. In OWL, types are inferred based on ontological reasoning. OWL object properties can be used to declare ad hoc relationships between ontological individuals.

In this work, an extensible compiler framework is presented that facilitates the cooperative usage of external type systems such as XSD and OWL, which depend on external classification and deduction engines, with C#. For the resulting programming language Zhi#, XSD and OWL compiler plug-ins were implemented in order to provide static typing and dynamic checking for constrained atomic data types and ontologies. XSD constraining facets and ontological inference rules were integrated with host language features such as method overriding. Zhi# programs are compiled into conventional C# and are interoperable with .NET assemblies. The proposed solution eases the development of Semantic Web applications and facilitates the use and reuse of knowledge in form of ontologies. Further compiler plug-ins may be implemented to provide for additional type systems such as of the Object Constraint Language (OCL).

ZUSAMMENFASSUNG

# ZHI# – PROGRAMMING LANGUAGE INHERENT SUPPORT FOR ONTOLOGIES

von

**Alexander Paar**

Universität Karlsruhe (TH), 2009

In weit verbreiteten objekt-orientierten Programmiersprachen wie Java oder C# werden Typprüfungen zur Übersetzungszeit vorgenommen. Statische Typsysteme bieten ein Rahmenwerk, das den Entwurf, das Verständnis und die Wartung objekt-orientierter Systeme vereinfacht. Ebenfalls können bestimmte Laufzeitfehler bereits zur Übersetzungszeit ausgeschlossen werden. Mit dem Aufkommen von Schemasprachen wie XML Schema Definition (XSD) oder Ontologie-Beschreibungssprachen wie der Web Ontology Language (OWL) werden jedoch die Grenzen von konventionellen statischen Typsystemen sichtbar. Von atomaren XML-Datentypen kann mittels Einschränkung ihres Wertebereichs abgeleitet werden. In OWL werden ontologische Schlussfolgerungen für die Typinferenz verwendet. Mit OWL Objekt-Eigenschaften können ad hoc-Beziehungen zwischen ontologischen Individuen deklariert werden.

In dieser Arbeit wird ein erweiterbares Compiler-Framework vorgestellt, das die kooperative Nutzung von externen Typsystemen wie XSD und OWL, die auf externen Klassifikations- und Deduktionsmechanismen beruhen, in der Programmiersprache C# vereinfacht. Für die in dieser Arbeit neu entwickelte Programmiersprache Zhi# wurden XSD- und OWL-Plugins implementiert, die erstmalig statische Typprüfung für eingeschränkte atomare XML-Datentypen und eine Kombination von statischer und dynamischer Typprüfung für Ontologien bieten. In Zhi# sind Einschränkungsprimitive von XSD und ontologische Schlussfolgerungsregeln mit Sprachkonstrukten und Techniken der Programmiersprache C# wie z.B. dem Überschreiben von Methoden integriert. Zhi#-Programme

werden in C# übersetzt und sind interoperabel mit .NET-Assemblies. Die vorgeschlagene Lösung vereinfacht die Entwicklung von Semantic Web-Anwendungen und erleichtert die Nutzung und Wiederverwendung von Wissen in Form von Ontologien.

Das Zhi#-Compiler-Framework erweitert Sprachkonstrukte und Techniken der Programmiersprache C# um externe Typisierungsmechanismen wie z.B. Typinferenz, Subsumption und Ableitung. Insbesondere werden eingeschränkte atomare Datentypen und Typinferenz basierend auf Kontroll- und Datenflussanalyse unterstützt. Unterstützung für externe Typsysteme wird von Compiler-Plugins bereitgestellt. Kooperation zwischen externen Typsystemen und zwischen externen Typsystemen und dem .NET-Typsystem wird mittels Delegation erreicht; jeder Teilausdruck wird von dem jeweils zuständigen Plugin überprüft und übersetzt. Die Architektur des Zhi#-Compiler-Frameworks mit zwei wohldefinierten Erweiterungspunkten für Typprüfung und Übersetzung ermöglicht die Implementierung von Plugins ohne vollständige Kenntnis der Grammatik der Programmiersprache C#. Ebenfalls ist innerhalb des Compiler-Frameworks kein a priori-Wissen über externe Typsysteme notwendig. Im Gegensatz zu naiven Ansätzen, die auf der Einführung von Wrapper-Klassen für externe Typdefinitionen basieren, ist der Mehraufwand kompilierter Zhi#-Programme konstant und wächst nicht mit der Anzahl importierter externer Typen.

XML-Datentypen können von eingebauten XSD-Typen durch die Anwendung von lexikalischen und Wertebereichs-Einschränkungsprimitiven abgeleitet werden. In dieser Arbeit wird eine Erweiterung des einfach typisierten Lambda-Kalküls mit Untertypen ( $\lambda_{<}$ ) für eingeschränkte atomare Datentypen vorgestellt. Die Typsicherheit des entwickelten  $\lambda_C$ -Typsystems wurde bewiesen auf Grundlage des Korrektheitsbeweises des  $\lambda_{<}$ -Kalküls. Das  $\lambda_C$ -Typsystem wurde vollständig in Java und C# implementiert. Insbesondere wurden diese Schnittstellen des  $\lambda_C$ -Typsystems für das XML Schema Definition-Typsystem implementiert sodass Typdefinitionen aus XSD-Dateien geladen und diese Typen in einer Hierarchie klassifiziert werden können.

Die Implementierung des  $\lambda_C$ -Typsystems bildet die Grundlage des Zhi#-Compiler-Plugins für XML Schema Definition. Das XSD-Compiler-Plugin bietet statische und dynamische Typprüfung für XML-Datentypen. Zur Übersetzungszeit werden die Typen von atomaren XML-Objekten entsprechend den Regeln des  $\lambda_C$ -Typsystems überprüft. Insbesondere ist der Zhi#-Compiler in der Lage, Auskunft zu geben, welche Einschränkung primitive genau von einer Zuweisung verletzt werden anstatt nur eine generelle Inkompatibilität festzustellen. XML-Datentypen können mit Sprachkonstrukten und Techniken der Programmiersprache C# wie z.B. dem Überschreiben von Methoden, benutzerdefinierten Operatoren und dynamischen Typprüfungen verwendet werden. Das XSD-Compiler-Plugin folgert die Typen von Variablen auf Grund von Kontroll- und Datenflussanalyse. Die Typinferenz-Regeln für *if*-Anweisungen im  $\lambda_C$ -Kalkül wurden ergänzt um Regeln für *for*- und *while*-Anweisungen in Zhi#-Programmen. Die Typinformation von Literalen wird aus den Literal-Werten gefolgert. Die Typen von binären Ausdrücken werden auf Grundlage einer Einschränkungs-Arithmetik für XML Schema Definition berechnet. Die erzeugten C#-Programme enthalten Anweisungen für dynamische Typprüfungen, um Modifikationen der XML-Schemata auch nach der Kompilierung und eine sichere Verwendung von Zhi#-Assemblies mit konventionellen .NET-Programmen zu ermöglichen.

Ontologische Daten werden in Wissensbasen verwaltet, auf die ontologische Schlussfolgerungen angewandt werden. Dazu wird in dieser Arbeit eine erweiterbare Architektur einer ontologischen Wissensbasis vorgestellt. Der CHIL Knowledge Base Server kann verwendet werden, um existierende Wissensdatenbanken zu adaptieren. In der gegenwärtigen Implementierung wird das Jena Semantic Web Framework, konfiguriert mit dem Pellet OWL DL-Schlussfolgerer und Speicher- und Datenbank-gestützten Ontologie-Modellen, adaptiert. Der CHIL Knowledge Base Server implementiert die formal spezifizierte CHIL OWL API, die auf Grundlage einer Kombination von Floyd-Hoare-Logik und Beschreibungslogik-Terminologie definiert wurde. Floyd-Hoare-Logik-Regeln können auf die API-Spezifikation angewandt werden, um auf der Metaebene Effekte von Operationen auf verwalteten Ontologien schlusszufolgern. Ergänzend zu der formalen Spezifikation ist



die Definition der CHIL OWL API als XML-Instanzdokument gegeben. Diese Beschreibung wurde verwendet, um automatisch Regressionstestcode für den CHIL Knowledge Base Server, XML-über-TCP-Server-Komponenten und eine Reihe von Klienten-Komponenten für Programmiersprachen wie z.B. Java, C# und C++ zu erzeugen. Die CHIL OWL API umfasst 91 formal spezifizierte Methoden und 32 technische Hilfsfunktionen.

Das Zhi#-Compiler-Plugin für OWL DL implementiert eine Kombination von statischer und dynamischer Typprüfung für ontologische Konzeptbeschreibungen. Die Existenz referenzierter ontologischer Konzepte und Rollen wird zur Übersetzungszeit überprüft. Teil der statischen Typprüfung ist außerdem die Überprüfung von disjunkten Konzeptbeschreibungen und Kardinalitätseinschränkungen und disjunkten Definitions- und Zielbereichs-Einschränkungen ontologischer Rollen. In Zhi# werden Definitions- und Zielbereichs-Einschränkungen ontologischer Rollen für automatisches Schlussfolgern verwendet und ermöglichen die Deklaration von ad hoc-Beziehungen. Mit dem *checked*-Operator kann Verhalten wie in Frame-basierten Wissensrepräsentationen erzwungen werden. Deklarierte RDF-Typen von ontologischen Individuen in Zhi#-Programmen werden zur Programmlaufzeit überprüft. Formale Hilfseigenschaften ontologischer Individuen, Rollen und statischer Konzeptreferenzen in Zhi#-Programmen erleichtern Aufgaben wie z.B. die Abfrage aller Individuen in der Erweiterung einer gegebenen Konzeptbeschreibung oder die Auflistung aller RDF-Typen eines gegebenen Individuums. Ontologische Schlussfolgerungen wurden integriert mit C#-Sprachkonstrukten wie dem *is*-Operator, der nun auch für dynamische Typprüfung von ontologischen Individuen verwendet werden kann. Ontologische Konzeptbeschreibungen können für formale Parameter von Methoden, benutzerdefinierten Operatoren und Indexern verwendet werden. Das OWL-Compiler-Plugin kann kooperativ mit dem XSD-Compiler-Plugin genutzt werden, um OWL-Datentyp-Eigenschaften zu unterstützen.

Die Zhi#-Entwicklungshilfsmittel umfassen den Zhi#-Compiler (siehe Abb. A.1) und ein Eclipse-basiertes Frontend mit u.a. einem Zhi#-Editor mit Syntaxhervorhebung und Autovervollständigung (see Abb. A.2).



# CHAPTER 1

## Introduction

In recent years, Semantic Web technologies such as RDF(S) [MM04, BG04b], DAML+OIL [HHP01], and their common Description Logics [BCM03] based successor OWL DL [MH04a] have paved the way for standardized formal conceptualizations of all kinds of knowledge. Numerous ontologies have been developed to conceptualize a plethora of domains of discourse [DAM04, Pro08]. Corporations from all sectors have braced to define company specific knowledge using Semantic Web technologies. Ontology engineering has become a business model for a number of companies. As the underlying standards have matured, tools for ontology engineering have emerged both in commercial as well as in academic fields. Knowledge acquisition systems such as Protégé [Sta06] make it particularly easy to construct domain ontologies and to enter data. Ontology management systems such as HP Labs' Jena [HP 04] can be used for loading ontologies from files and via the Internet and for creating, modifying, querying, and storing ontologies. Inference engines such as RACER [MH04b] and Pellet [Pel06] provide support for query answering. There is a growing set of tools, projects, and applications for ontology languages such as OWL.

However, processing ontological information programmatically is still laborious and error-prone. From the author's experience, this is mainly caused by the deficient integration of XML Schema Definition based type definitions, which may be the range of OWL datatype properties, and terminological knowledge in form of ontologies with widely used object-oriented programming languages. Neither compilers nor integrated development environments for existing object-oriented programming languages are aware of XML data type definitions and OWL concept descriptions. Up to now, ontologi-

cal concepts, roles, and individuals can only be used via ontology management systems [HP 04, Pel06, MH04b, Mot06], which merely provide APIs that have to be used in an explicit manner. Using generic APIs to address elements of an ontology prevents any of the handy features that programmers are accustomed to for programming language class definitions. In particular, there is no autocompletion and no type checking for XML data types and ontological concept descriptions in programming languages such as Java or C#. The burden to wisely manipulate ontological data is put on the programmer. This problem is even more evident since software systems are usually composed of two distinct class hierarchies as depicted in Fig. 1.1.

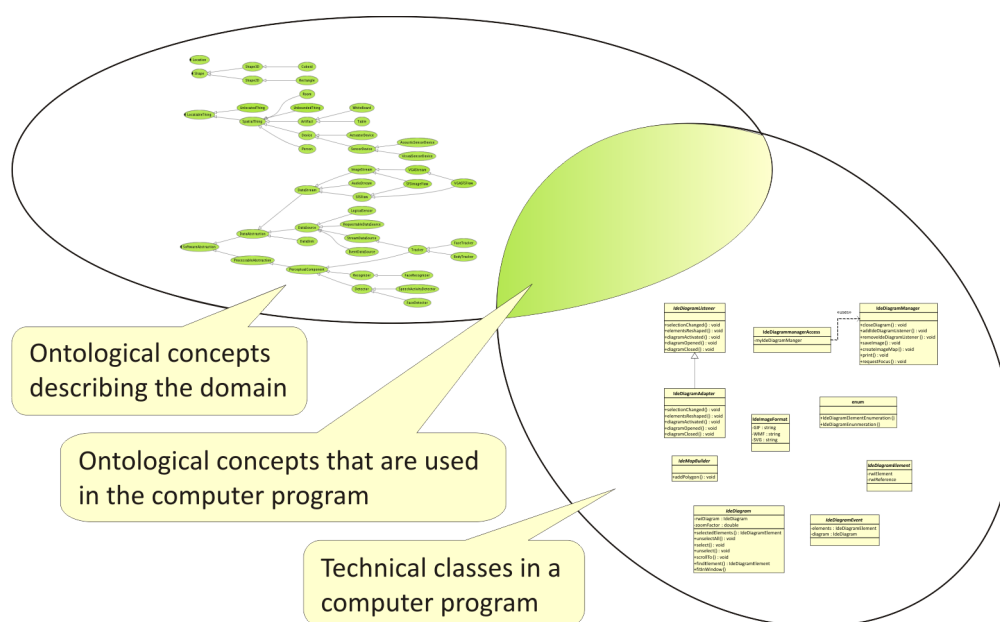


Figure 1.1: Two intersecting class hierarchies in a software system

One class hierarchy comprises knowledge about a problem domain in form of concepts of an ontology (i.e. domain specific concepts such as, for example, *Product*, *Price*, and *Invoice* in a business application) whereas the other one comprises classes that make up the technical framework of a software application (e.g., classes such as *System.IO.File* or *System.Console*). In order to handle both class hierarchies in one UML class model, technologies such as OMG's Ontology Definition Metamodel (ODM) [Obj05b] aim to

represent both models – the technical and the domain specific one – in the same MOF [Obj05a] modeling space. Thus, on the meta-level, ontologies and UML class models may be related to each other in one unified model, which even includes a certain degree of MOF-based static data type analysis.

Still, once the ontological and technical parts of the unified model are transformed into an ontology markup and a programming language, respectively, the unified model again falls apart into two distinct class hierarchies. Important parts of the conceptualization (i.e. the ontology) may then be (re-)defined as an integral part of the otherwise technical class hierarchy of the application with no respect to the actual ontology. For example, an ontology may define a concept *Person* whereas at the same time there may be a programming language class *Person* with identical semantics defined in a program's source code. This redundant class definition may arise from the need to have a proxy class for the ontological concept or from the unawareness of the identical concept definition in the ontology. As a consequence, it is particularly difficult to make sure that the application adheres to the conceptualization as defined in the ontology and to tell where in the program those ontological concepts are referenced. Redefining equivalent concepts or disregarding conceptual type information makes it impossible to efficiently share data among different components in a distributed software system and results in a lack of available ABox reasoning.

Even if ontological and technical classes are declared in the same model, ontological concepts and programming language classes reveal different behavior in terms of type inference, concept subsumption, and modeling features.

In statically typed object-oriented programming languages such as C#, properties are declared as class members. The domain of a property corresponds to the type of the containing host object. Only instances of the domain type can have the declared property. The range of a property (i.e. class attribute) is given by an explicit type declaration. This type declaration is authoritative, too. All objects that are declared to be values of a

property must be instances of the declared type. For ontological concepts and properties, which are subject to *automatic reasoning*, domain and range restrictions are interpreted differently. In contrast to C#, in OWL ontologies, property assignments do not fail immediately. Instead, the reasoner considers the declared piece of knowledge for subsequent deduction steps. If an ontological property relates an individual to another individual, and the property has a class as its range, then the other individual is inferred to belong to this range class. For example, the property *hasChild* may be stated to have the range of *Human*. From this a reasoner can deduce that if ALICE is related to BOB by the *hasChild* property (i.e., Bob is the child of Alice), then BOB is a *Human*. In the same way, if *hasChild* is stated to have the domain *Parent*, the reasoner can deduce that ALICE is a *Parent*. These different property domain and range semantics alone make it particularly difficult – if not impossible – to simply substitute an OWL ontology by an isomorphic set of C# wrapper classes.

Erik Meijer and Peter Drayton note that “at the moment that you define a [programming language] class *Person* you have to have the divine insight to define all possible relationships that a person can have with any other possible object or keep type open” [MD04]. In contrast, OWL properties form a hierarchy of their own, which can be extended independently of the concept hierarchy. Thus, OWL properties facilitate *ad hoc relationships* between objects that may not have been foreseen when a concept was defined. The Zhi# programming language provides the power of the “.” to access such properties of ontological individuals using normal member access. Members can be added on a per instance basis by declaring OWL property values for ontological individuals.

Processing OWL data implicitly requires the use of constrained<sup>1</sup> atomic XML Schema Definition data types [BM04b], which may be the range of OWL datatype properties.

---

<sup>1</sup>In this work, the term “constrained types” refers to atomic data types that represent a value space, which may be constrained by explicitly defined constraining facets (e.g., *xsd:minExclusive*). This is different to constraint-based type inference algorithms found in the literature where constraints are not checked but rather recorded for later consideration.

For example, an ontological concept *Person* may have defined a property *hasAge* of type *xsd#unsignedInt*<sup>2</sup>. This type could be mapped to the C# data type *System.UInt32*. If, however, the XML schema defined a further constrained atomic data type such as *unsignedIntLessThan110* in order to constrain possible *hasAge* values of a *Person* to reasonable values less than 110, there would be no appropriate C# data type with a value space that comprises integer values between 0 and 110. Instead, assignments to objects of type *System.UInt32* would have to be explicitly checked to be *schema valid*. An XML instance document – or in this case an instance of a constrained atomic data type – is said to be a valid instance of a schema if there is an XML schema given, and the content of the XML instance document – or of the data type value – conforms to the content model as defined in the schema. Up to now, schema validation has been particularly error-prone since there is no isomorphic mapping between XML data types and programmatic data types.

In this work, the Zhi#<sup>3</sup> programming language is presented that implements programming language inherent support for atomic XML Schema Definition data types and the Web Ontology Language OWL DL. The objective of this work is to make constrained XML data types and ontological concept and role definitions first class citizens of a conventional object-oriented programming language and to automate tedious validation and type checking tasks.

The Zhi# programming language is a pluggable extension of ECMA standard C# 1.0 [HWG02]. XML and OWL compiler components can be activated to operate separately with standard C# code or to cooperate. External types can be included using the keyword *import*, which permits the use of external types in a Zhi# namespace.

```
1 import XML xsd = http://www.w3.org/2001/XMLSchema;  
2 import OWL owl = http://www.w3.org/2002/07/owl;
```

---

<sup>2</sup>In this work, the XML prefixes *xs* and *xsd* are bound to the XML Schema Definition namespace <http://www.w3.org/2001/XMLSchema>.

<sup>3</sup>Zhi (Chinese): Knowledge, information, wisdom.

The Zhi# compiler framework supports the usage of arithmetic (+, -, \*, /, &, |, ^, %), relational (>=, >, ==, <, <=), and logical (&&, ||) operators with external types. The dot operator '.' can be used to access members of external types. Types of different type systems may be used cooperatively in one single statement. For example, a .NET *System.Int32* variable can be assigned an XML data type value *age*, which may be defined as a property value of an instance of the ontological concept *Person*.

```
1 #ont#Person Alice = [...];  
2 int i = Alice.#ont#hasAge;
```

In Zhi#, external type definitions can almost unrestrictedly be used with almost all C# programming language features (since Zhi#'s support for external types is a language feature and not (yet) a feature of the runtime, similar restrictions to the usage of external types apply as for generic type definitions in the Java programming language). For example, methods can be overridden using external types, user defined operators can have external input and output parameters, and arithmetic and logical expressions can be built up using external objects.

As detailed in Sections 2.5, 5.2, and 6.5 there have been a number of prior systems to provide programming language inherent support for XML Schema Definition and to extend a host language with pluggable type systems. The Zhi# approach is distinguished by a combination of features that is targeted to make external type systems available in an object-oriented programming language using conventional object-oriented notation.

- The Zhi# compiler framework provides two extension points that can be used to implement type checking and program transformation capabilities for external type systems. The interfaces of both extension points are for the most part context free with respect to the code structure of C# programs. A compiler plug-in for an external type system can therefore be implemented without exhaustive knowledge about the syntax of the C# programming language.



- Both XML Schema Definition type definitions and Web Ontology Language concept descriptions require type system specific classifiers to compute the subsumption hierarchies of data types and concepts. Additionally, ontology management systems include a deduction component that can make implicit knowledge explicit (i.e. automatic reasoning). The Zhi# compiler framework is tailored to facilitate the integration of such external classifier and reasoner components with the type checking of Zhi# programs.
- Because XML data types are inductively defined through their value spaces (the set of values for a given data type) it makes sense to infer the possible values that an expression in a program can have at runtime. The Zhi# compiler framework facilitates the implementation of type inference based on control and data flow analysis. The XSD compiler plug-in implements control and data flow based type inference plus a constraint arithmetic to infer the types of binary expressions that relate constrained atomic data types.
- Zhi#'s OWL compiler plug-in makes the property centric modeling features of the Web Ontology Language available via C#'s object-oriented notation (i.e. normal member access). The power of the “.” can be used to declare ad hoc relationships between ontological individuals and to declare members on a per instance basis.
- The number of the created proxy classes for XSD type definitions and OWL concept descriptions is constant and does not depend on the number of imported XML data types and OWL concept descriptions.
- The Zhi# compiler framework and its XSD and OWL compiler plug-ins were fully implemented in C#. An Eclipse-based frontend was implemented including an editor with syntax highlighting and autocompletion.

Existing approaches in the field of programming language support for external schema languages such as XML Schema Definition simply map XSD type definitions to plain

C# (or Java) types. Thereby, only generic incompatibilities between types based on the generated wrapper classes can be reported at compile time. As a consequence, the resulting compiler messages are very imprecise (e.g., in the case of constrained XML data types). For ontology markup languages such as the Web Ontology Language, the typing rules of object-oriented programming languages are completely inapt. In contrast, the Zhi# compiler framework can be extended with relevant type checking and program transformation capabilities in order to provide concise type system-specific compile-time support for external schema and ontology languages. Depending on how external type references in Zhi# programs are transformed into conventional C# code, it is also possible for external compiler plug-ins to provide for dynamic type checking as well.

The outline of this work is as follows. The remaining sections of this introductory chapter describe the developments that led from semantic networks to the Description Logic  $SHOIN(\mathbf{D})$ , from the early implementations of SGML to XML, and from the combination of these technologies and the Resource Description Framework to the Web Ontology Language (OWL DL). Also, fundamental type system properties are illustrated by the simply typed lambda calculus with subtyping. Chapter 2 describes the technical aspects of the Zhi# compiler framework, which was extended with XML and OWL compiler plug-ins. The XML and OWL plug-ins are based on the constrained types calculus discussed in Chapter 3 and on the CHIL OWL API elucidated in Chapter 4, respectively. Chapter 5 describes the integration of constrained types with conventional C# programming language features such as method overloading and overriding. Chapter 6 discusses the differences between ontologies and statically typed object-oriented programming languages. It will be shown how Zhi#'s static and dynamic checking of OWL concept descriptions and ontological roles contributes to the terminological validity of modifications of assertional ontological data. Chapter 7 indicates the applicability and usefulness of the presented approach. Chapter 8 summarizes the accomplished results and outlines future work on programming language inherent support for ontologies. Related work is discussed at the end of each chapter.

## 1.1 Theses

**Thesis 1** *The compiler of a general purpose object-oriented programming language can be made extensible to typing and subtyping mechanisms of a priori unknown external domain-specific type systems.*

- (i) *Constraint-based type derivation and value space-based subtyping of atomic data types as in the XML Schema Definition type system can be made first class citizens of a general purpose object-oriented programming language.*
- (ii) *Ontological concept descriptions, role definitions, and reasoning can be integrated with the type checking of a general purpose object-oriented programming language.*
- (iii) *Object-oriented notation is sufficient to manipulate the assertional knowledge of Description Logic knowledge bases.*
- (iv) *No a priori knowledge about the external type systems is required in the compiler framework and no exhaustive knowledge about the host language grammar is required to implement compiler plug-ins for external type systems.*

Thesis 1 will be validated by a working implementation of the extensible Zhi# compiler. Two compiler plug-ins will augment the complete safe (i.e. managed) fragment of ECMA 334 standard C# version 1.0 with static typing and dynamic checking for XML data types, OWL DL concept descriptions, and ontological roles.

**Thesis 2** *The Zhi# solution eases the development of Semantic Web applications and promotes the use and reuse of knowledge in form of data type schemas and ontologies.*

- (i) *The integration of constrained atomic data types with a general purpose object-oriented programming language reduces the number of runtime validation errors for XML data types.*
- (ii) *The integration of ontological reasoning with a general purpose object-oriented programming language facilitates the use of OWL DL concept descriptions and ontological roles.*

Thesis 2 will be microscopically and macroscopically validated in order to demonstrate the ease of use and the practicability of the novel Zhi# programming language features.

## 1.2 Position on the Market

In 1959, Peter Drucker coined the term *knowledge worker* [Dru93]. A knowledge worker is someone who works primarily with information or one who develops and uses knowledge in the workplace. Nowadays the term particularly includes those in the information technology fields, such as computer programmers. It was Peter Drucker, too, who declared *business value* as the proper goal of a firm. Especially, a firm should create business value for customers, employees (especially knowledge workers), and distribution partners. In this work, a solution is presented that provides business value to knowledge workers (i.e. computer programmers) by facilitating the development of Semantic Web applications, which generate and process ontological data.

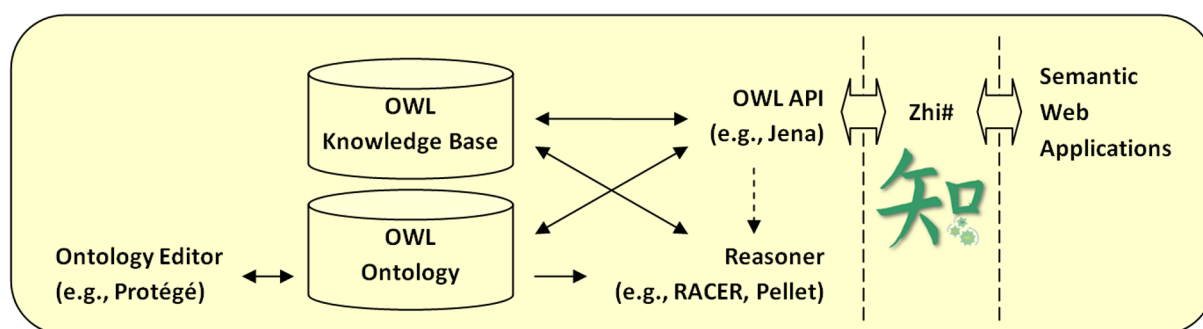


Figure 1.2: Position on the market

The Zhi# solution improves the ease-of-use of existing ontology management systems and should be of interest to anyone who develops Semantic Web applications. Everyone who uses OWL APIs to programmatically modify and query ontological knowledge bases will greatly benefit from the convenient notation and the improved type checking that Zhi# provides for XML data types and OWL concept descriptions. XML Schema Definition data types and ontological concept descriptions can be used natively in Zhi# programs. Programmers do not need to use tedious and error-prone ontology management APIs or generate a host of proxy classes anymore. Zhi# programs are interoperable with conventional .NET assemblies and can be used concurrently with API-based knowledge

base clients, which allows for a smooth migration of an existing code-base. Eventually, the Zhi# solution reduces the dependency on particular OWL APIs. Even for compiled Zhi# programs the Zhi# runtime library can be substituted to utilize different ontology management systems.

### 1.3 Conceptual Modeling with the Web Ontology Language

The main purpose of the Zhi# programming language, which is described in this work, is to make it particularly easy to process ontological knowledge in form of OWL DL knowledge bases. This chapter addresses the theoretical work in Description Logics and the foundations of XML Schema Definition and of the Resource Description Framework that have led to the development of the Web Ontology Language (OWL). In particular, this chapter is to motivate the application of the Description Logics based Web Ontology Language (OWL DL) as a formalism to express knowledge.

#### 1.3.1 From networks to Description Logics

This subsection summarizes the emergence of Description Logics from earlier technologies that had previously been used to model knowledge. Also, shortcomings of older efforts are described followed by a brief introduction to the syntax and the basic features of the Description Logic *SHOIN(D)*, which constitutes the formal foundation of the Web Ontology Language (OWL DL).

As of today, Knowledge Representation (KR) as a field of Artificial Intelligence refers to methods for providing formal high-level descriptions of a domain of discourse that are epistemologically and computationally adequate to build intelligent applications. In particular, a knowledge representation language must be expressive enough to model all required arguments of the domain of discourse. While providing the necessary representational adequacy, a knowledge representation system must also be computationally

effective [Sow91]. It should not only specify how single fragments of knowledge are to be expressed but also how the complete knowledge base as a whole is to be structured to retrieve relevant information easily and to make it particularly efficient to apply a reasoner to automatically infer implicit consequences from explicitly represented knowledge.

In the 1960s, network based notations – often based on graphical interfaces – were among the first knowledge representation schemes to be developed. In contrast to logic-based knowledge representation formalisms, in non-logical approaches, knowledge used to be represented by means of some ad hoc data structures where reasoning was accomplished by similarly application specific and informal procedures. Both semantic networks [Qui67] and frames [Min81] are incarnations of such specialized notations in a sense that sets of individuals and their relationships are represented by the structure of a network. Network-based approaches boast intuitive cognitive notion and user-centric origin, which derived from experiments on recall of human memory. In the first place, most networked-based systems used to behave rather differently despite similar graphical and syntactical components. A huge step towards providing sound semantics to graphical representation structures was the identification of hierarchical structures as in monotonic inheritance networks [Bra77, Bra79]. A pictorial representation of a simple terminology is given in Fig. 1.3. The link between *Person* and *Age* says that “*A person has an age*”. The “is-a” relationship between *Philosopher* and *Person* says that “*A philosopher is a person*”. Consequently, one can infer that a *Philosopher* has an *Age* since it inherits all properties of the more general concept *Person*.

A further step to improve the ease of representation and the efficiency of reasoning was the recognition that it is possible to translate a semantic network into first-order predicate calculus [Hay79, Cha81]. Nodes of a semantic network can be characterized as unary predicates. Binary predicates can denote relations between classes or sets of individuals. It even turned out that only fragments of first-order logic are required to cover both frames and semantic networks [BL85].

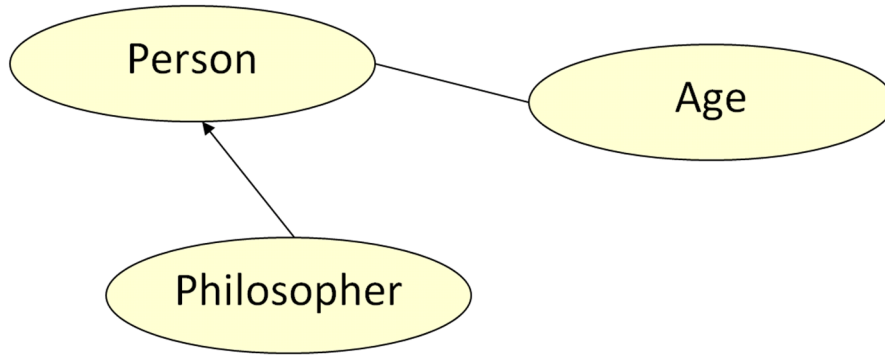


Figure 1.3: An example network

The first approaches to logic-based knowledge representation developed in the 1970s. Logic-based formalisms mostly employed variants of first-order predicate calculus where reasoning amounts to verifying logical consequence. First-order logic is an extension of propositional logic, which formulates the logic of mathematical objects called propositions. In propositional logic, these objects are represented in formal deduction systems called propositional calculi whose atomic formulas are propositional variables. A propositional calculus is a formal system  $L = (A, \Omega, Z, I)$ . The alpha set  $A$  is a finite set of the most basic elements of the formal language  $L$  called proposition symbols or propositional variables. The omega set  $\Omega$  is a finite set of elements called operator symbols, which is typically partitioned into two disjoint subsets  $\Omega_1 = \{\neg\}$  and  $\Omega_2 = \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ . The zeta set  $Z$  is a finite set of inference rules that can be used to derive logically equivalent expressions from a given expression. The iota set  $I$  is a finite set of initial points that are called axioms. The set of well-formed formulas  $L$  is recursively defined as follows.

1. Any element of the alpha set  $A$  is a formula of  $L$ .
2. If  $p$  is a formula, then  $\neg p$  is a formula.
3. If  $p$  and  $q$  are formulas, then  $(p \wedge q)$ ,  $(p \vee q)$ ,  $(p \rightarrow q)$ , and  $(p \leftrightarrow q)$  are formulas.
4. Nothing else is a formula of  $L$ .

Complex formulas can be constructed by repeated application of these four rules. For example, by rule 1  $p$  is a formula, by rule 2  $\neg p$  is a formula, by rule 1  $q$  is a formula, and by rule 3 ( $\neg p \rightarrow q$ ) is a formula.

Given a set of formulas that are assumed to be true other true formulas can be derived using the inference rules shown below. With the first eight non-hypothetical rules certain well-formed formulas can simply be produced from other well-formed formulas. The last two rules use hypothetical reasoning, which temporarily assumes a possibly unproven hypothesis to be part of the set of inferred formulas to see if certain other formulas can be inferred.

1. Double negative elimination: From  $\neg\neg p$ , infer  $p$ .
2. Conjunction introduction: From  $p$  and  $q$ , infer  $(p \wedge q)$ .
3. Conjunction elimination: From  $(p \wedge q)$ , infer  $p$ ; from  $(p \wedge q)$ , infer  $q$ .
4. Disjunction introduction: From  $p$ , infer  $(p \vee q)$ ; from  $q$ , infer  $(q \vee p)$ .
5. Disjunction elimination: From  $(p \vee q)$ ,  $(p \rightarrow r)$ ,  $(q \rightarrow r)$ , infer  $r$ .
6. Bi-conditional introduction: From  $(p \rightarrow q)$ ,  $(q \rightarrow p)$ , infer  $(p \leftrightarrow q)$ .
7. Bi-conditional elimination: From  $(p \leftrightarrow q)$ , infer  $(p \rightarrow q)$ ; from  $(p \leftrightarrow q)$ , infer  $(q \rightarrow p)$ .
8. Modus ponens: From  $p$ ,  $(p \rightarrow q)$ , infer  $q$ .
9. Conditional proof: If accepting  $p$  allows a proof of  $q$ , infer  $(p \rightarrow q)$ .
10. Reductio ad absurdum: If accepting  $p$  allows a proof of  $q$  and a proof of  $\neg q$ , infer  $\neg p$ .

Propositional logic is decidable. A sentence can be shown to be a tautology by devising a truth table; the formula is a tautology if every assignment of truth values makes it true.



It is, though, not known whether this method is efficient. The equivalent problem to show that a formula is satisfiable is a canonical example of an NP-complete problem. In any case, propositional logic only sets out the features of very elementary kinds of propositional reasoning. For example, it cannot be used to work out the logical connections of the following argument.

$$\frac{\text{Some men are philosophers. All philosophers seek wisdom.}}{\text{Some men seek wisdom.}}$$

In propositional logic, this argument would be a sequence of three distinct simple sentences, which would not be related to each other. The reason for this is that the connections between the simple statements are due to the fact that the predicate *philosopher* of the first premise “*Some men are philosophers*” occurs as a subject in the second premise “*All philosophers seek wisdom*”. Propositional logic lacks features to express such kinds of connections.

In first-order predicate calculus, atomic formulas are predicates with one or more subjects such as  $P(s_1, \dots, s_n)$ , which, in contrast to propositional calculi, can be related to each other via quantification. Where  $\phi$  is any sentence, the new constructions  $\forall x . \phi$  and  $\exists x . \phi$ , read “for all  $x$ ,  $\phi$ ” and “for some  $x$ ,  $\phi$ ”, are introduced. Quantifiers make it possible to say for how many individuals a particular statement is true. The universal quantifier ‘ $\forall$ ’ formalizes the notion that something (i.e. a logical predicate) is true for *everything* (i.e. every relevant thing). The existential quantifier ‘ $\exists$ ’ predicates a property or relation to at least one member of the domain.

While monadic predicate logic (i.e. predicate logic with only predicates with arity = 1) is decidable, it may be insufficient to model even simple arguments as follows.

$$\frac{\text{Philosophers are men.}}{\text{Therefore philosophers' thoughts are thoughts of men.}}$$

The conclusion cannot be written in monadic predicate logic without hiding the connection between the premise “*Philosophers are men*” and the conclusion “*Philosophers'*

*thoughts are thoughts of men*". In particular, a 2-place predicate is required to write  $Thought(x, y)$  for "*x is the thought of y*". For general predicate logic with arbitrary numbers of predicate places, there is no mechanical decision procedure for determining derivability or validity in a finite number of steps (i.e. general predicate logic is undecidable [Chu36]).

In contrast, Description Logics (DL) are intended to be Knowledge Representation (KR) systems that should answer the queries of a user within a reasonable amount of time. Unlike first-order theorem provers, the *decision procedures* of a DL system should always terminate, for both positive and negative answers.

Description Logics (DL) are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description). Within a knowledge base one can distinguish between general knowledge about a domain of discourse (the *intensional knowledge*) and assertions that pertain to particular incarnations (the *extensional knowledge*). Accordingly, a DL knowledge base comprises two components. The general terminology is contained in the TBox. The contingent knowledge about particular individuals is contained in the ABox. Chapter 4 will present an API to tell (i.e. modify) and ask (i.e. query) both the TBox and ABox of a DL knowledge base.

The fundamental inference on concept expressions is *subsumption*, typically written  $C \sqsubseteq D$ . Subsumption is the problem of checking whether a concept description denoted by  $D$  (the *subsumer*) is more general than the one denoted by  $C$  (the *subsumee*). Also, individuals can automatically be classified to be instances of particular concept descriptions. The inference of subsumption and instance relationships from the definition of the concepts and the properties of the individuals is a remarkable difference to explicitly given "is-a" relationships in semantic networks. A special case of subsumption with the sub-

sumer being the empty concept is *satisfiability*, which is the problem of checking whether a concept description does not necessarily denote the empty concept. For empty concepts the set of individuals of this concept is always empty.

Investigating the effect of the expressiveness of a Description Logic on the computational complexity of its reasoning algorithms has been the main area of research on Description Logics since the beginning of the 1980s.

Early implementations of DL systems such as KL-ONE, K-REP, BACK, and LOOM [BS89, MDW91, Pel91, Gre91] implemented *structural comparison* algorithms [Lip82] where concept descriptions are normalized and transformed into labeled graphs. Reasoning amounts to recursively comparing the syntactic structure of the normalized descriptions [Neb90]. While algorithms for *structural subsumption* are sound and boast polynomial complexity they are complete only for rather inexpressive Description Logics, i.e. for more expressive DLs, they cannot detect all existing subsumption relationships. The observation that structural subsumption may be too weak concluded the era of network based subsumption algorithms and triggered the development of tableaux calculi.

This new algorithmic paradigm was influenced by Brachman and Levesque’s seminal paper “The tractability of subsumption in frame-based description languages” [BL84]. Brachman and Levesque elucidated the tradeoff between the expressiveness of a knowledge representation language and the computational complexity to reason over instantiations of such representations.

The first tableau-based subsumption algorithm was proposed by Schmidt-Schauß and Smolka [SS91] for a language  $\mathcal{ALC}$  (Attributive Concept Descriptions with Complements) at the beginning of the 1990s. This approach and its extensions to decide subsumption for a number of other DLs [Hol90, Baa90, BH91, HB91, Han92, DLN95, Hor98] were found out to be terminating specializations of the tableau calculus for first-order predicate logic. A tableau-based algorithm breaks down the concepts in the knowledge base and tries to build a model with inferred constraints on its elements. The decision procedure either

stops when obvious contradictions occur while building the model or it terminates and yields a “canonical” model of the knowledge base. In 1991 it also became clear that DLs are closely related to modal logics [Sch91].

Following thorough analysis of the complexity of reasoning was the development of highly optimized systems such as FaCT [Hor98], RACER [HM01], and DLP [Pat98] in the mid 1990s, which showed that tableau-based algorithms lead to good practical behavior even for expressive DLs. Based on tableau-based algorithms the open-source reasoner Pellet [Pel06, SPG07] supports the full expressivity of the  $\mathcal{SHOIN}(\mathbf{D})$  Description Logic (i.e. the Web Ontology Language OWL DL).

With the advent of the Semantic Web at the beginning of the 21<sup>st</sup> century, more industrial strength DL systems are being developed that will prospectively be capable of processing even large scale knowledge bases and expressive representation languages such as the  $\mathcal{SHOIN}(\mathbf{D})$  Description Logic, which is described in the following subsection.

### 1.3.2 The $\mathcal{SHOIN}(\mathbf{D})$ Description Logic

Description Logics evolved from the need to give formally rigid semantics to the elements and structures of semantic networks. DL languages comprise two disjoint alphabets of symbols to denote atomic concepts and roles, which can be seen as unary and binary predicate symbols, respectively. Accordingly, the variable free DL concept expression  $C$  is given a set theoretic interpretation; it denotes the set of all individuals of an interpretation domain for which  $C(x)$  is true. More complex terms can be build from the basic symbols using constructors such as the *intersection of concepts*. The DL concept description  $C \sqcap D$  can be regarded as the first-order logic sentence  $C(x) \wedge D(x)$ , which includes all individuals under consideration for which both  $C(x)$  and  $D(x)$  is true. Roles are interpreted as pairs of individuals that are related to each other in a *subject-property-object* form.

Different DLs provide different sets of concept constructors. A key feature and common to all DLs are value restrictions that limit the values of ontological roles to the set of

individuals for which certain properties can be asserted. For example, the value restriction  $\forall R.C$  requires that all individuals that are in the relationship  $R$  with the concept being described are elements of the interpretation of concept  $C$ . Thus, if an individual of the concept description  $\forall R.C$  is related by the property  $R$  to a second individual, then the second individual can be inferred to be an instance of concept  $C$ .

Knowledge Representation systems based on Description Logics boast inference capabilities that make it possible to infer implicit knowledge from explicitly represented knowledge. The *instance algorithm* determines whether an individual is an instance of a concept description, i.e. the individual is always interpreted as an element of the given concept. The *subsumption algorithm* allows one to determine subconcept-superconcept relationships. A concept  $C$  is subsumed by  $D$  if the set of instances denoted by  $C$  is a subset of the set of instances denoted by  $D$  (i.e. the first description is always interpreted as a subset of the second one). The *consistency algorithm* checks that the assertions and terminological axioms in a knowledge base are non-contradictory.

Description Logic languages are distinguished by the constructors they provide. The basic Description Logic  $\mathcal{AL}$  (Attributive Language) shown in Table 1.1 was introduced in 1991 by Schmidt-Schauß and Smolka [SS91]. In the remainder of this chapter, the letters  $A$  and  $B$  will be used for atomic concepts, the letters  $R$  and  $U$  for abstract roles and datatype roles, respectively, and the (possibly subscripted) letter  $C$  for complex concept descriptions.

The semantics are given by means of an *interpretation*  $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$  consisting of a non-empty domain  $\Delta^{\mathcal{I}}$  and a mapping  $\cdot^{\mathcal{I}}$ , which interprets atomic and complex concepts, and roles as shown in Table 1.1.

In the Description Logic  $\mathcal{AL}$ , concept descriptions are formed using the constructors conjunction, value restriction, and existential restriction. Negation can only be applied to atomic concepts and only the universal concept  $\top$  can be used with the existential restriction over a role.

Table 1.1: The basic Description Logic  $\mathcal{AL}$

Constructor Name	Syntax	Semantics
atomic concept	$A$	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
top level concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
bottom concept	$\perp$	$\perp^{\mathcal{I}} = \{\}$
atomic negation	$\neg A$	$(\neg A)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
intersection	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
value restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y \rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
limited existential quantification	$\exists R.\top$	$(\exists R.\top)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y \rangle \in R^{\mathcal{I}}\}$

Using the constructors of the  $\mathcal{AL}$  language one could model a business meeting scenario as it is considered in the domain of the CHIL research project [Inf04]. First, atomic concepts for persons and meetings are introduced.

$$Person \quad Meeting$$

Using a general concept inclusion employees are defined as a subset of all persons under consideration.

$$Employee \sqsubseteq Person$$

A *general concept inclusion* (GCI) is of the form  $C_1 \sqsubseteq C_2$ , where  $C_1, C_2$  are concepts. A finite set of GCIs is called a TBox. An interpretation  $\mathcal{I}$  is a model of a TBox  $\mathcal{T}$  iff it satisfies all GCIs in  $\mathcal{T}$ , i.e.  $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$  holds for each  $C_1 \sqsubseteq C_2 \in \mathcal{T}$ . Concept definitions of the form  $C_1 \equiv C_2$  stand for the two GCIs  $C_1 \sqsubseteq C_2$  and  $C_2 \sqsubseteq C_1$ . A concept name is called *defined* if it occurs on the left-hand side of a definition; *primitive* otherwise.

The concept description for meeting participants, i.e. persons who attend a meeting, can be constructed by using the intersection of all persons with a limited existential quantification over the *attendsMeeting* role.

$$MeetingParticipant \equiv Person \sqcap \exists attends Meeting. \top$$

A special marketing meeting can be defined using a value restriction for its topics.

$$MarketingMeeting \equiv Meeting \sqcap \forall hasTopic. MarketingTopic$$

In addition, external guests are those persons who are not employees.

$$Guest \equiv Person \sqcap \neg Employee$$

The given terminology already exploits the language features of  $\mathcal{AL}$ . The more expressive language  $\mathcal{ALC}_{\mathcal{R}^+}$  is obtained by adding the negation of arbitrary concepts (indicated by the letter  $\mathcal{C}$ , for “complement”) and transitive roles ( $\mathcal{R}^+$ ) as shown in Table 1.2.

Table 1.2: The Description Logic  $\mathcal{ALC}_{\mathcal{R}^+}$ , extends  $\mathcal{AL}$  (Table 1.1)

Constructor Name	Syntax	Semantics
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Axiom Name	Syntax	Semantics
object role transitivity	$\text{Trans}(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$

With the additional constructor the concept of a meeting that is attended by external guests can be defined as follows.

$$OpenMeeting \equiv Meeting \sqcap \neg \forall hasParticipant. Employee$$

The transitivity of a role can be used by a reasoner to infer entailments (e.g., a person who is located in a meeting room, which is located in a building, is located in that building as well).

In order to avoid very long names for Description Logics, the abbreviation  $\mathcal{S}$  has been introduced for the  $\mathcal{ALC}_{\mathcal{R}^+}$  language. Prominent members of the  $\mathcal{S}$ -family are  $\mathcal{SHIF}$ ,

which which lays out the formal basis for the Web Ontology Language OWL Lite (in OWL Lite only the values 0 and 1 are allowed for cardinality restrictions), and  $\mathcal{SHOIN}(\mathbf{D})$ , which corresponds to the Web Ontology Language OWL DL. The  $\mathcal{SHIF}$  Description Logic extends  $\mathcal{ALC}_{\mathcal{R}^+}$  with role hierarchies ( $\mathcal{H}$ ), inverse roles ( $\mathcal{I}$ ), and number restrictions ( $\mathcal{F}$ ) as shown in Table 1.3.

In order to avoid role expressions such as  $R^{-}$ , a function  $\mathbf{Inv}$  is defined, which returns the inverse of a role. Let  $\mathbf{R}$  be a set of role names. The set of  $\mathcal{SHIF}$ -roles is then  $\mathbf{R} \cup \{R^- \mid R \in \mathbf{R}\}$ .

$$\mathbf{Inv}(r) := \begin{cases} r^- & \text{if } r \text{ is a role name,} \\ s & \text{if } r = s^- \text{ for a role name } s. \end{cases}$$

Table 1.3: The Description Logic  $\mathcal{SHIF}$ , extends  $\mathcal{ALC}_{\mathcal{R}^+}$  (Table 1.2)

Constructor Name	Syntax	Semantics
at least restriction	$\geq nR$	$(\geq nR)^{\mathcal{I}} = \{x \mid  \{y.\langle x, y \rangle \in R^{\mathcal{I}}\}  \geq n\}$
at most restriction	$\leq nR$	$(\leq nR)^{\mathcal{I}} = \{x \mid  \{y.\langle x, y \rangle \in R^{\mathcal{I}}\}  \leq n\}$
inverse role	$R^-$	$(R^-)^{\mathcal{I}} = (R^{\mathcal{I}})^-$
Axiom Name	Syntax	Semantics
object role inclusion	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$

Using a number restriction a small meeting can be defined as a meeting that has at most five participants.

$$\mathit{SmallMeeting} \equiv \mathit{Meeting} \sqcap \leq 5 \mathit{hasParticipant}$$

The fact that a meeting is attended by all people who are present in the meeting is expressed by an inverse role as follows.

$$\mathit{attendedBy} \equiv \mathit{attendsMeeting}^-$$



Attending a meeting shall be considered a special case of attending an event.

$$\textit{attendsMeeting} \sqsubseteq \textit{attendsEvent}$$

The constructors and axioms of the *SHIF* language already provide for extensive domain models. Still, in order to cover the expressiveness of OWL DL requires two more fundamental features resulting in the Description Logic *SHOIN(D)*, which extends *ALC<sub>R+</sub>* with role hierarchies ( $\mathcal{H}$ ), nominals (i.e. “one of”-constructors) ( $\mathcal{O}$ ), inverse roles ( $\mathcal{I}$ ), number restrictions ( $\mathcal{N}$ ), and a data type theory  $\mathbf{D}$  as shown in Table 1.4. According to OWL DL semantics, the semantic domain comprises two disjoint parts. The abstract domain  $\Delta^{\mathcal{I}}$  comprises abstract objects (i.e. individuals); the concrete domain  $\Delta_{\mathbf{D}}^{\mathcal{I}}$  contains concrete objects (i.e., data type values). The data type theory  $\mathbf{D}$  is a mapping from a set of data types (e.g., XML Schema Definition types) to sets of values (e.g., from *xsd#integer* to the integers) plus a mapping from data values to their denotations (e.g., from "1" to the integer 1).

Let  $\mathbf{A}$ ,  $\mathbf{D}$ ,  $\mathbf{R}_A$ ,  $\mathbf{R}_D$ , and  $\mathbf{I}$  be pairwise disjoint sets of concept names, data type names, abstract role names, data type (or concrete) role names, and individual names. The set of *SHOIN(D)*-roles is then  $\mathbf{R}_A \cup \{R^- \mid R \in \mathbf{R}_A\} \cup \mathbf{R}_D$ .

Table 1.4: The Description Logic *SHOIN(D)*

---

Constructor Name	Syntax	Semantics
atomic concepts $\mathbf{A}$	$A$	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
data types $\mathbf{D}$	$D$	$D^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
abstract roles $\mathbf{R}_A$	$R$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
data type roles $\mathbf{R}_D$	$U$	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
individuals $\mathbf{I}$	$o$	$o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
data values	$v$	$v^{\mathcal{I}} = v^{\mathbf{D}}$

Table 1.4: The Description Logic  $\mathcal{SHOIN}(\mathbf{D})$ 

inverse role	$R^-$	$(R^-)^{\mathcal{I}} = (R^{\mathcal{I}})^-$
top level concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
bottom concept	$\perp$	$\perp^{\mathcal{I}} = \{\}$
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
disjunction	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
one of	$\{o_1, \dots, o_n\}$	$\{o_1, \dots, o_n\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \dots, o_n^{\mathcal{I}}\}$
exists restriction	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y \rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y \rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
at least restriction	$\geq nR$	$(\geq nR)^{\mathcal{I}} = \{x \mid  \{y. \langle x, y \rangle \in R^{\mathcal{I}}\}  \geq n\}$
at most restriction	$\leq nR$	$(\leq nR)^{\mathcal{I}} = \{x \mid  \{y. \langle x, y \rangle \in R^{\mathcal{I}}\}  \leq n\}$
datatype exists	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y \rangle \in U^{\mathcal{I}} \text{ and } y \in D^{\mathbf{D}}\}$
datatype value	$\forall U.D$	$(\forall U.D)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y \rangle \in U^{\mathcal{I}} \rightarrow y \in D^{\mathbf{D}}\}$
datatype at least	$\geq nU$	$(\geq nU)^{\mathcal{I}} = \{x \mid  \{y. \langle x, y \rangle \in U^{\mathcal{I}}\}  \geq n\}$
datatype at most	$\leq nU$	$(\leq nU)^{\mathcal{I}} = \{x \mid  \{y. \langle x, y \rangle \in U^{\mathcal{I}}\}  \leq n\}$
datatype one of	$\{v_1, \dots, v_n\}$	$\{v_1, \dots, v_n\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \dots, v_n^{\mathcal{I}}\}$
<b>Axiom Name</b>	<b>Syntax</b>	<b>Semantics</b>
concept inclusion	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
object role inclusion	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
object role transitivity	$\text{Trans}(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
datatype role inclusion	$U_1 \sqsubseteq U_2$	$U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
individual inclusion	$o : C$	$o^{\mathcal{I}} \in C^{\mathcal{I}}$
individual equality	$o_1 = o_2$	$o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$
individual inequality	$o_1 \neq o_2$	$o_1^{\mathcal{I}} \neq o_2^{\mathcal{I}}$

In 2003, Horrocks and Patel-Schneider [HP04] showed that computing ontology entailment in the Web Ontology Language OWL DL has the same complexity as computing knowledge base satisfiability in  $\mathcal{SHOIN}(\mathbf{D})$ . While procedures for deciding subsumption in  $\mathcal{SHOIN}(\mathbf{D})$  have exponential worst-case complexity, the design of “practical” algorithms, which cope well with problems that occur in realistic applications, has been the subject of active investigation [BCM03].

### 1.3.3 XSD – XML Schema Definition

In 1967, William W. Tunnicliffe articulated the idea of separating the formatting of a document from its content. Based on this *generic coding* Charles Goldfarb, Edward Mosher and Raymond Lorie developed the Generalized Markup Language GML<sup>4</sup>. GML markups (tags) can be used to define the structure of documents (e.g., headers, chapters) without further specifying how particular elements of this structure are to be presented. In 1986, the Standard Generalized Markup Language [Woo95] became an ISO standard [Int86]. However, its complexity has prevented its widespread application.

In the late 1990s an SGML subset referred to as the Extensible Markup Language (XML) [BPS06] appeared and soon became a W3C standard in 1999. XML is simpler to parse and process than full SGML and has been widely adopted since then as a format to exchange data. Despite its name XML is not a language with a given vocabulary but rather a set of syntax rules that can be applied to create languages. Such languages are called *applications* of XML. XML documents can be any kind of well-formed (syntactically correct) Unicode character strings (e.g., files, data streams). The relevant information of an XML document is represented by its Information Set [CT04] whose integral parts are *elements* and *attributes*. The content of an element can be text or other elements. In addition, elements can be attributed with name/value pairs. Both elements and attributes may occur in *namespaces*, which are a mechanism to create globally unique names based on the URI naming scheme [BFM98].

---

<sup>4</sup>Goldfarb used their surnames to make up the term GML.

XML Schema Definition (XSD) [FW04] is a schema definition language to constrain the content model of an XML instance document to a specific hierarchical element structure and particular element data types. The XSD specification comprises two parts. “XML Schema Part 1: Structures” [TBM04] sets out the structural part of the XML Schema Definition language, which can be used to define content models of complex (aggregated) data types. “XML Schema Part 2: Datatypes” [BM04b] defines the type system for atomic types. It has been influenced by the ISO 11404 [Int96] standard on language-independent data types as well as the type system of SQL [Int99] or programming languages such as Java [GJS05]. In XSD, an atomic data type is a three-tuple consisting of a set of distinct values, called its value space, a set of lexical representations called its lexical space, and a set of fundamental facets that characterize properties of the value space such as the notion of equality or an order relation. There are 19 primitive built-in atomic data types (e.g., *xsd#string*, *xsd#decimal*, *xsd#dateTime*) from which user-defined types can be derived through the application of *constraints*. XSD defines 12 different constraining facets as shown in Table 1.5.

Table 1.5: XSD constraining facets

<i>length</i>	defines the number of <i>units of length</i>
<i>minLength</i>	defines the minimum number of <i>units of length</i>
<i>maxLength</i>	defines the maximum number of <i>units of length</i>
<i>pattern</i>	constrains the lexical space to literals that match a specific pattern
<i>enumeration</i>	constrains the value space to a specified set of values
<i>minInclusive</i>	defines the inclusive lower bound of the value space
<i>minExclusive</i>	defines the exclusive lower bound of the value space
<i>maxExclusive</i>	defines the exclusive upper bound of the value space
<i>maxInclusive</i>	defines the inclusive upper bound of the value space
<i>totalDigits</i>	defines the maximum number of values in the value space
<i>fractionDigits</i>	defines the minimum difference between values in the value space
<i>whiteSpace</i>	controls the normalization of string data types (modifying)

Constraining facets can be *modifying* (i.e. string literals that are assigned to instances of constrained types are implicitly modified upon assignment) or *enforcing* (i.e. certain value space constraints must hold for the interpretations of string literals that are assigned to instances of constrained types). Using constraint based type derivation, a data type *age* can be derived from the built-in XML data type *xsd#integer* through the application of the constraints *xsd:minInclusive* and *xsd:maxExclusive* as follows.

```
1 <xsd:schema xmlns:xsd="..." >
2   <xsd:simpleType name="age">
3     <xsd:restriction base="xsd:integer">
4       <xsd:minInclusive value="0"/>
5       <xsd:maxExclusive value="110"/>
6     </xsd:restriction >
7   </xsd:simpleType>
8 </xsd:schema>
```

Recent general purpose programming languages such as C# [HWG02] include a number of system types for which an isomorphic mapping exists to built-in atomic XSD types (e.g., *System.Int32*  $\mapsto$  *xsd#int*). Still, the notion of constraining facets and value space based subtyping has not been embedded yet with nominal type systems of object-oriented programming languages.

Besides XML Schema Definition there are a number of other schema languages. Document Type Definition DTD [BPS06] was the original schema definition language inherited from SGML. While some markup languages are still defined with DTD today the non-XML syntax, the lack of data types, and the lack of support for namespaces have made many users switched to XSD. Relax NG [Int03] is the top competitor to W3C's XSD and is considered superior by many people in the XML community. On the other hand, major software companies have come out strongly in favor of standardizing on XML Schema Definition. From today's perspective XSD is widely accepted and is referenced by a number of other W3C technologies such as the Web Ontology Language.

### 1.3.4 RDF – The Resource Description Framework

In 1999, slightly more than one year after the standardization of XML 1.0, the Resource Description Framework (RDF) [MM04] became a W3C Recommendation. RDF is a Semantic Web technology to describe resources on the Web and to capture semantic relationships between them. An RDF document contains one or more “descriptions” of resources. A description is a set of statements. In the RDF model such statements are also called “triples” as they comprise the resource (subject), its properties (predicate), and the property values (object) as shown in Fig. 1.4.

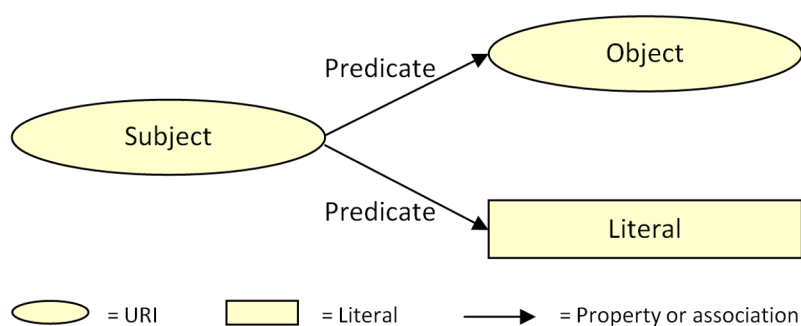


Figure 1.4: The RDF triple

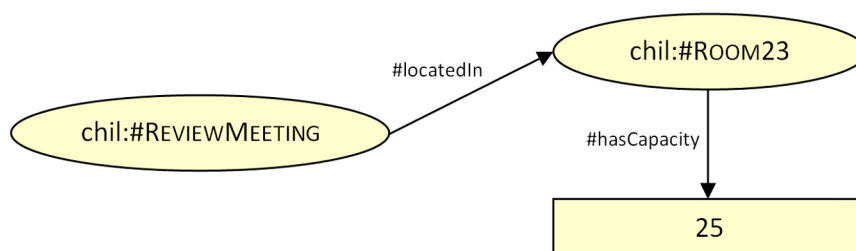


Figure 1.5: A graph of two RDF statements

The *subject* is the resource that is being described by the ensuing predicate and object (in logic, the subject is the term about which something is asserted). The *predicate* is the relation between the subject and the *object*, which is either a resource referred to by the predicate or a literal value such as a string or integer. The combination of the three elements subject, predicate, and object is called a *statement*. A simple RDF graph

comprising two statements about the venue of a review meeting and the capacity of a room is shown in Fig. 1.5.

In RDF, everything can be considered a resource as long as it can be unambiguously identified by a resource ID, i.e. it has an identity. It is therefore good practice to use URIs that comprise a namespace part and a local name. For example, the URI “<http://www.chokycola.com#company>” may be used denote a form of business ownership instead of friends coming for a visit. In this work, for the sake of brevity, the fragment identifier “#” may be used as a shortcut; it would be more accurate to use the absolute URI instead.

RDF features the concept of *non-contextual* modeling. Unlike XML, the context of an RDF statement cannot be determined beforehand. In XML documents, the semantics of elements depend on their position in the hierarchy. Their relationship and those of elements’ names and their values are given implicitly. In contrast, RDF explicitly states the relationships between resources. Thus, it is possible to dynamically add to lists of RDF statements. XML’s contextual modeling may be more appropriate for high-bandwidth transactions with fixed contexts of documents and messages. Non-contextual modeling should be considered in changing environments where flexibility is an issue.

Beyond the representation as a graph there are a number of other ways to capture knowledge expressed as a list of triples. Following the linguistic model of subject, predicate, and object the following two English statements express the relevant information of the RDF graph shown in Fig. 1.5.

The review meeting takes place in room no. 23.

Room no. 23 has a capacity of 25.

It is conceivable to (automatically) extract sentences like these from daily routines and business processes. As a result of such a bottom-up approach corporate knowledge may accrue into a common knowledge base. In the N3 notation [Ber05], which was designed

by Tim Berners-Lee as a compact and human-readable format for RDF data, the two preceding sentences may then be written as follows.

```
1 @prefix chil: <http://chil.server.de#>.
2 @prefix id: <#>.
3 <chil:REVIEWMEETING> <id:takesPlacesIn> <chil:ROOM23>.
4 <chil:ROOM23> <id:hasCapacity> 25.
```

Finally, there is a standardized XML syntax for RDF called RDF/XML [BM04a] as shown in the following listing.

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:id="#" xmlns:chil="http://chil.server.de#">
5   <rdf:Description
6     rdf:about="http://chil.server.de#REVIEWMEETING">
7     <id:takesPlaceIn>
8       <rdf:Description
9         rdf:about="http://chil.server.de#ROOM23">
10        <id:hasCapacity>25</id:hasCapacity>
11      </rdf:Description>
12    </id:takesPlaceIn>
13  </rdf:Description>
14 </rdf:RDF>
```

RDF also derives benefit from XML Schema Definition by allowing explicit typing of literal values. Values of typed literals comprise a string (the lexical form of the literal) and a data type (identified by a URI). Line 10 of the above listing can be replaced by a predicate that refers to a typed literal as follows. Note that RDF literals are always statically typed (i.e. there is no way to dynamically infer the data type of a literal from some schema or ontology definition).



```
1 <id:hasCapacity
2   rdf:datatype="http://www.w3.org/2001/XMLSchema#positiveInteger">
3   25
4 </id:hasCapacity>
```

Line 4 of the preceding N3 listing can accordingly be augmented to include the type information as follows.

```
1 <chil:ROOM23>
2 <id:hasCapacity>
3   "25"^^http://www.w3.org/2001/XMLSchema#positiveInteger
```

The RDF/XML representation is commonly used as the “serialization format” because of its syntactic compatibility with other XML-based technologies. However, RDF’s open grammar, which allows arbitrary namespace-qualified elements to occur at any place in RDF documents, subtle differences in how XSD and RDF process namespaces, and advanced RDF features such as its container models and the possibility of “making statements about statements“ (reification) are the reasons why RDF does not yet play well with XML documents.

Whereas containers affect the modeling of a single statement (for example, an object becoming a collection of values), reification refers to treating a statement as the object of another statement. Reification is akin to statements as arguments instead of statements as facts. A common application of reification is, for example, to make assertions about the trustworthiness of information.

Also, alternate representations of the same kind of information and the nesting of a list of statements in a hierarchical XML syntax make it particularly difficult for humans to directly author RDF data in this format. The benefits of combining XML and RDF became for the first time increasingly apparent with the formation of the W3C’s Web Ontology Working Group in November 2001 and their objective to develop a standard ontology language for the Semantic Web during the time until May 2004.

### 1.3.5 RDFS – The Resource Description Framework Schema

RDF properties represent relationships between resources. As such, they correspond to attribute/value pairs in object-oriented programming languages. There is, however, no way to define *classes*, i.e. templates for objects that are composed of properties (data members). Also, RDF lacks mechanisms to describe the meanings of properties (e.g., there is no standardized property to define “is-a”-relationships between resources) nor relations between them.

The Resource Description Framework Schema (RDFS) [BG04b] is a semantic extension of RDF. Formal semantics [HM04] are needed for two reasons. First, a lack of formal semantics would hamper the development of RDF(S) implementations. Secondly, technologies such as the Web Ontology Language, which are layered on top of RDFS, have formal semantics. They require RDFS to have formal semantics that they can extend. Otherwise every layer on top of RDF would have to define its own RDF semantics.

RDFS defines a standardized set of resources and properties that can be used to define domain specific RDF vocabularies (i.e. ontologies) that comprise classes, properties, and instances. The RDFS class and property system is similar to type systems of object-oriented programming languages such as Java. The main difference is that a class is not defined in terms of the properties its instances may have. Unlike the centralized *resource centric* approach of object-oriented systems, *property centric* RDFS vocabularies are developed in a decentralized fashion. RDFS properties are defined independently of a class, i.e. anyone is able to define new properties for a class. The relationship of a property with instances of a class is given by its *domain* and *range* restrictions, which define the types of an instance to which a property applies.

Fig. 1.6 shows a simple UML diagram that comprises the definition of an “is-a”-relationship between the two classes *Meeting* and *Event*, a class *Room* with a positive integer valued data member called *hasCapacity*, and an instance of the *Room* class with a given capacity.

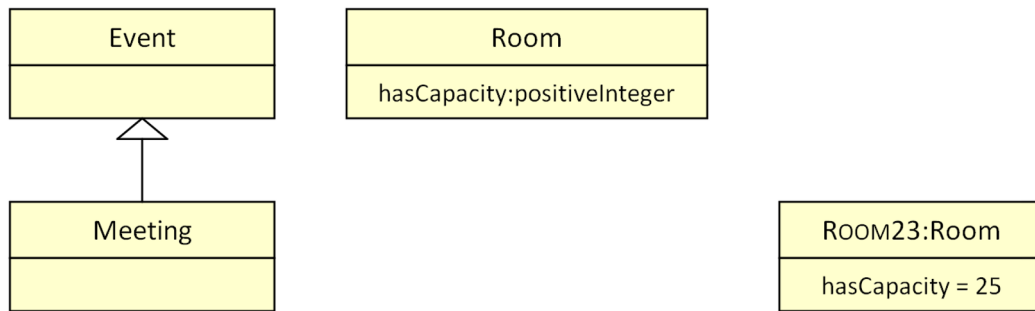


Figure 1.6: An example UML diagram

The following listing is the RDF Schema for the class and instance model of Fig. 1.6. The RDFS element *rdfs:Class* denotes the group of resources that are RDF Schema classes; *rdfs:subClassOf* states that one class is a subclass of another.

```

1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
4 <rdfs:Class rdf:ID="Meeting">
5 <rdfs:subClassOf>
6 <rdfs:Class rdf:ID="Event"/>
7 </rdfs:subClassOf>
8 </rdfs:Class>
9 <rdfs:Class rdf:ID="Room"/>
10 <rdf:Property rdf:ID="hasCapacity">
11 <rdfs:domain
12 rdf:resource="#Room"/>
13 <rdfs:range
14 rdf:resource="http://www.w3.org/2001/XMLSchema#positiveInteger"/>
15 </rdf:Property>
16 <Room rdf:ID="ROOM23">
17 <hasCapacity
18 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">25</hasCapacity>
19 </Room>
20 </rdf:RDF>
  
```

RDF distinguishes between a class and the set of its instances. Associated with each class is a set, called the *extension* of the class, which is the set of the instances of the class. This distinction is made in order to avoid conflicts with the axiom of foundation of Zermelo-Fraenkel set theory. In RDF, two classes may have the same set of instances but be different classes.

The element *rdf:Property*, which is an instance of *rdfs:Class*, is the class of RDF properties. The domain and range of a property is defined by the elements *rdfs:domain* and *rdfs:range*, respectively. Note that in RDFS there is no direct way to indicate class specific property restrictions. A typed node element is used to define an instance of the class *Room* with a *hasCapacity* value of 25.

The next subsection introduces the Web Ontology Language, which uses most of the RDFS modeling primitives.

### 1.3.6 OWL – The Web Ontology Language

In the year 2000, the DARPA Agent Markup Language (DAML) program [The00] originated a more expressive schema language than RDFS. Around the same time the European Union-based Ontology Inference Layer (OIL) research project [FHH01] aimed at developing an ontology based infrastructure for the Semantic Web. Soon both groups became aware of each other and joined their efforts in the DAML+OIL approach [HHP01]. By March 2001, two versions of DAML+OIL had been produced, which in turn were superseded by the Web Ontology Language (OWL<sup>5</sup>) [MH04a], which became a W3C Recommendation in 2004 [DS04].

The need for a language layer on top of RDF(S) came from the fact that the expressiveness of both RDF and RDF Schema is deliberately limited. To put it in a nutshell,

---

<sup>5</sup>The natural acronym for Web Ontology Language would be WOL instead of OWL. Rumor has it that the language was named OWL in allusion to an owl in A. A. Milne's Winnie-the-Pooh stories. Unlike the other woodlanders the owl is the only character capable of writing its name; still, only with a typo. Sometimes, the acronym OWL is also considered as a tribute to William A. Martin's *One World Language* KR project from the 1970s.

RDF is (roughly) limited to binary predicates and RDF Schema is (roughly) limited to a subclass and property hierarchy with domain and range definitions of these properties. The following enumeration of missing features is taken from [AH03].

- Local scope of properties: In RDFS, one cannot declare range restrictions that are local to some classes only. For example, one cannot say that a marketing meeting has only marketing topics on its agenda and a technology meeting only technology topics.
- Disjointness of classes: In RDFS, one can only declare subclass relations while it is not possible to state that two classes such as *Guest* and *Employee* are disjoint.
- Boolean combination of classes: The constructors union, complement, and intersection are not supported by RDFS.
- Cardinality restrictions: In RDFS, there is no way to place restrictions on how many distinct values a property may or must have (e.g., a small meeting must have at most five participants).
- Special characteristics of properties: In RDFS, properties cannot be declared to be transitive (e.g., *locatedIn*) or the inverse of other properties (e.g., *attendsMeeting* and *attendedBy*).

W3C's Web Ontology Working Group thought out OWL to comply with the main requirements on an ontology markup language. In addition to adding sufficient expressive power to RDFS, the prospective Web Ontology Language was to have a well-defined syntax and semantics. Also, it should provide for efficient reasoning. Unfortunately, the combination of the powerful RDFS modeling primitives such as *rdfs:Class* (the class of all classes) with a full-fledged logic led to a computationally intractable language called *OWL Full*. OWL Full is a straightforward extension of RDFS using the RDF meaning of classes and properties. In this way, OWL Full is fully upward compatible with RDFS.

OWL Full has the same constructors as the Description Logic  $\mathcal{SHOIN}(\mathbf{D})$  (see Table 1.4). Any OWL Full modeling primitive can be combined in arbitrary ways with RDF and RDF Schema. This is why there are no decision procedures for OWL Full, which limits its practical usability to scenarios that depend on RDF Schema's meta-modeling features (e.g., defining classes of classes). As a consequence, two sublanguages of OWL Full were defined to regain computational efficiency.

*OWL DL* is the subset of OWL Full that restricts particular usages of  $\mathcal{SHOIN}(\mathbf{D})$  constructors. As a consequence, OWL DL is not fully upward compatible with RDF. Still, every legal OWL DL document is still a legal RDF document. Full RDF compatibility was traded off for the possibility of efficient reasoning. In 2003, Horrocks and Patel-Schneider [HP04] showed that computing ontology entailment in OWL DL has the same complexity as computing knowledge base satisfiability in  $\mathcal{SHOIN}(\mathbf{D})$ .

*OWL Lite* is an even further restricted flavor. It abides by the same semantic restrictions as OWL DL and is limited to constructors provided by the Description Logic  $\mathcal{SHLF}(\mathbf{D})$ . Hence, highly optimized Description Logic systems such as FaCT [Hor98] and RACER [HM01] can be used to provide reasoning services for OWL Lite. The differences between OWL DL and OWL Lite can be found in [MH04a].

All OWL species use the RDF syntax. Ontological individuals are declared as in RDF using RDF descriptions and typing information. Moreover, the OWL elements *owl:Class*, *owl:ObjectProperty*, and *owl:DatatypeProperty* are derived from their RDF(S) counterparts *rdfs:Class* and *rdf:Property*, respectively.

As for RDF, a number of syntactic forms for OWL have been defined. RDF/XML [DS04] is the primary syntax for OWL though it is most difficult for humans to read. The OWL Abstract Syntax [PHH04] is used by the language specification. A graphical OWL syntax [Sch02] was based on the Universal Modeling Language UML [BRJ99]. The XML presentation syntax for OWL [HEP03] was defined as a dialect similar to the OWL Abstract Syntax with human readability in mind.

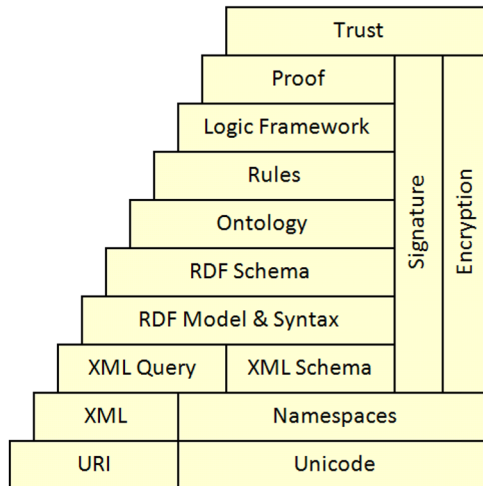


Figure 1.7: The Semantic Web stack

Ontologies and their accompanying inference rules are the topmost layer of the Semantic Web stack depicted in Fig. 1.7 that is considered in this work. The Semantic Web may possibly be complemented by a logic-framework that will allow the sharing of logic proofs in order to establish a “Web of trust”, which is the final Web in Tim Berners-Lee’s three-part vision of a collaborative Web, a Semantic Web, and a Web of trust.

### 1.3.7 The CHIL OWL DL Ontology

In the CHIL research project [Inf04], the Web Ontology Language OWL DL was deployed in order to implement programming language independent knowledge representation for system, component, and domain modeling [PR09].

The heterogeneous CHIL software environment comprises a plethora of *perceptual components*, which are based on image and speech recognition technologies. In order to interoperate, all of these components are required to produce data that is also meaningful to prospective consumers of this kind of information. These client components may be written in a variety of different programming languages. Moreover, CHIL services define a number of different scenarios such as meetings or other human-to-human interactions.

The CHIL Ontology was developed to provide a formal high-level description of the CHIL domain of discourse that can be efficiently used to build intelligent applications. The CHIL ontology introduces a common programming language agnostic domain model that can be used to formally represent elements of the CHIL domain of discourse.

By using the Description Logics based Web Ontology Language OWL DL to model concepts, properties, and individuals of the CHIL domain of discourse, it is possible to use a reasoner in order to automatically make implicit knowledge explicit.

The CHIL Ontology was devised by first defining the relevant concepts of the CHIL domain, and then using these concepts to specify properties of objects and individuals occurring in the domain. As of today, “intelligent” refers to the ability to automatically infer implicit consequences from explicitly represented knowledge. Leveraging the expressiveness of the Web Ontology Language, the CHIL Ontology provides a description of the CHIL world that can be efficiently used to build intelligent applications.

The following knowledge base contains information about an exemplary CHIL scenario, which may illustrate applications of OWL modeling and reasoning features in the CHIL domain of discourse. The TBox and ABox are given in Description Logics notation (i.e. the symbol  $\top$  denotes the top level concept, the relational operator  $\sqsubseteq$  denotes a subconcept/subrole relation, and  $\equiv$  denotes equivalence).

**TBox**  $Moderator \sqsubseteq Participant \sqsubseteq Person \sqsubseteq \top$ ,  $Room \sqsubseteq \top$ ,  $Meeting \sqsubseteq Event \sqsubseteq \top$ ,  
 $ActivityLevel \sqsubseteq \top$ ,  $\geq 1hasActivityLevel \sqsubseteq Meeting$ ,  $\top \sqsubseteq \forall hasActivityLevel.ActivityLevel$ ,  
 $\geq 1takesPlaceIn \sqsubseteq Event$ ,  $\top \sqsubseteq \forall takesPlaceIn.Room$ ,  $hosts \equiv takesPlaceIn^-$ ,  
 $ActivityLevel(HIGH)$ ,  $ActiveMeeting \equiv Meeting \sqcap \exists hasActivityLevel.HIGH$ ,  
 $ActivityLevel(LOW)$ ,  $\geq 1scheduledAt \sqsubseteq Event$ ,  $\top \sqsubseteq \forall scheduledAt.xsd\#dateTime$ ,  
 $\geq 1hasModerator \sqsubseteq Meeting$ ,  $\top \sqsubseteq \forall hasModerator.Moderator$ ,  $moderates \equiv hasModerator^-$ ,  
 $\geq 1hasParticipant \sqsubseteq Event$ ,  $\top \sqsubseteq \forall hasParticipant.Participant$ ,  
 $participatesIn \equiv hasParticipant^-$ ,  $hasModerator \sqsubseteq hasParticipant$ ,  $\top \sqsubseteq \leq 1hasModerator$



**ABox** *Person*(ALICE), *Person*(BOB), *Person*(CHARLIE), *Room*(ROOM248),  
*Event*(PROJECTMEETING), *takesPlaceIn*(PROJECTMEETING, ROOM248),  
*scheduledAt*(PROJECTMEETING, 2008-06-27T13:00:00Z),  
*moderates*(ALICE, PROJECTMEETING), *participatesIn*(BOB, PROJECTMEETING),  
*participatesIn*(CHARLIE, PROJECTMEETING).

The given ABox declares three persons Alice, Bob, and Charlie. There is a project meeting that takes place at a certain time (given as an XML Schema Definition *dateTime* value) in room number 248. The meeting is moderated by Alice. Bob and Charlie participate, too.

The following list contains tell and ask operations on the given knowledge base. The input ( $\exists$ ) and output knowledge ( $\sqsupset$ ) is given in natural language (NL) and RDF triple-syntax (RDF). Note how much the RDF triple notation resembles the natural language sentences.

1.  $\sqsupset$  NL: Room no. 248 hosts the project meeting.  
RDF: ROOM248 *hosts* PROJECTMEETING
2.  $\sqsupset$  NL: The project meeting is scheduled at 27 June 2008, 1:00 PM.  
RDF: PROJECTMEETING *scheduledAt* 2008-06-27T13:00:00Z
3.  $\sqsupset$  NL: The project meeting has the participants Alice, Bob, and Charlie.  
RDF: PROJECTMEETING *hasParticipant* {ALICE, BOB, CHARLIE}
4.  $\sqsupset$  NL: Alice is a moderator.  
RDF: ALICE *rdf:Type* Moderator
5.  $\sqsupset$  NL: The project meeting event is a meeting.  
RDF: PROJECTMEETING *rdf:Type* Meeting

6.  $\Rightarrow$  NL: Elsie moderates the project meeting.  
 RDF: ELSIE *moderates* PROJECTMEETING
7.  $\Leftrightarrow$  NL: Alice and Elsie are the same persons.  
 RDF: ALICE *owl:sameAs* ELSIE
8.  $\Rightarrow$  NL: The project meeting has a high activity level.  
 RDF: PROJECTMEETING *hasActivityLevel* HIGH
9.  $\Leftrightarrow$  NL: The project meeting is an active meeting.  
 RDF: PROJECTMEETING *rdf:type* ActiveMeeting

The storyline develops as follows. A meeting summarization perceptual component queries all events that are hosted in room number 248. Because the ontological role *hosts* is defined as the inverse of *takesPlaceIn* the reasoner automatically derives the fact that room number 248 hosts the project meeting (1: *inverse roles*). The meeting summarizer queries the scheduled start time of the project meeting (2) and starts recording. It needs to assemble a list of participants and queries the ontological knowledge base. Bob and Charlie are participants of the project meeting since the ontological role *hasParticipant* is declared to be the inverse of *participatesIn*. Alice is a meeting participant, too, because the ontological role *hasModerator* is a subproperty of *hasParticipant* (3: *inverse roles, subroles*).

In particular, the RDF type of individuals Bob and Charlie is inferred to be *Participant* because of the domain restriction of the *participatesIn* role. Analogously, Alice is inferred to be a *Moderator* because of the domain restriction of *moderates* (4: *property domain restrictions*). Because of the range restriction of the *moderates* role, the project meeting *Event* is classified as a *Meeting* (5: *property range restrictions*). Note how the domain and range restrictions of ontological roles are different from authoritative class definitions in object-oriented programming languages and how ontological roles can be used to declare ad hoc relationships between individuals.

Another example for intuitive ontological reasoning are functional roles such as *hasModerator* along with several property values. Suppose the additional piece of information that Elsie moderates the project meeting under consideration (6). Because there can only be one value for the functional role *hasModerator* the reasoner concludes that the ontological individuals ALICE and ELSIE must refer to the same entities in the described world (7: *inverse roles, functional roles, equivalent individuals*). Note that because in OWL there is no *unique name assumption* with no additional information a reasoner will not deduce that two individuals are distinct.

In course of the project meeting, the activity level, which is detected by the smart room technologies, changes from low to high (8). Because of the nominal definition in the ontology, the project meeting is inferred to be an *ActiveMeeting* (9: *nominals*).

For a complete list of *SHOIN(D)* concept constructors see [HP04]. More examples of ontological reasoning can be found in The Description Logic Handbook [BCM03].

## 1.4 Interpretation of Object-Oriented Programming Languages

This section provides an interpretation of the primary concepts of commonly used object-oriented programming languages such as C# and Java. Based on the typing rules of the simply typed lambda calculus familiar features such as subtyping and base types are presented. These features constitute the formal foundation of the  $\lambda_C$ -calculus, which is presented in Chapter 3. In the  $\lambda_C$ -calculus, structural subtyping schemes of the simply typed lambda calculus with subtyping were used to devise constrained types.

The untyped lambda-calculus was developed by Alonzo Church and his co-workers in the 1920s and 1930s [Chu41]. In the lambda-calculus, all computation is reduced to the basic operations of function definition and application. The lambda-calculus can be viewed simultaneously as a simple programming language *in which* computations can be described and as a mathematical object *about which* statements can be proved. This section owes much to [Pie02].

### 1.4.1 The untyped lambda-calculus

This subsection alludes to the syntax and the operational semantics of the untyped lambda-calculus ( $\lambda$ ). The abstract grammar shown in Table 1.6 should be read as shorthand for an inductively defined set of abstract syntax trees.

Table 1.6: Untyped lambda-calculus ( $\lambda$ )

---

<b><i>Syntax</i></b>	
$t ::=$ $x$ $\lambda x.t$ $tt$	<i>terms:</i> <i>variable</i> <i>abstraction</i> <i>application</i>
$v ::=$ $\lambda x.t$	<i>values:</i> <i>abstraction value</i>
<b><i>Evaluation</i></b>	$t \rightarrow t'$
$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$	(E-APP1)
$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2}$	(E-APP2)
$(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$	(E-APPABS)

---

The expression “ $\lambda n. \dots$ ” can be thought of as a shorthand for “the function that, for  $n$ , yields  $\dots$ ”. Also note that in the lambda-calculus, *everything* is a function, i.e. the arguments accepted by functions are themselves functions, and the results returned by functions are functions as well. According to the given grammar, one can give an inductive definition of valid terms of  $\lambda$  as follows.

**Definition 1.1 (Terms)** Let  $\mathcal{V}$  be a countable set of variable names. The set of terms is the smallest set  $\mathcal{T}$  such that

1.  $x \in \mathcal{T}$  for every  $x \in \mathcal{V}$ ;
2. if  $t_1 \in \mathcal{T}$  and  $x \in \mathcal{V}$  then  $\lambda x.t_1 \in \mathcal{T}$ ;
3. if  $t_1 \in \mathcal{T}$  and  $t_2 \in \mathcal{T}$  then  $t_1 t_2 \in \mathcal{T}$ .

Just like the given grammar, the inductive definition says nothing about the use of parentheses to mark compound subterms. This is due to the fact that  $\mathcal{T}$  is actually defined as a set of trees, not as a set of strings. Parentheses can be used to assign an indented underlying tree form to a linearized form of terms.

In this text, the metavariable  $t$  (as well as  $s$  and  $u$ ) can range over arbitrary lambda-terms. The metavariable  $x$  (as well as  $y$  and  $z$ ) stands for arbitrary variables. In particular,  $x$ ,  $y$ , etc. may also be used as object-language variables. In such cases the context will make clear which is which.

An occurrence of the variable  $x$  is said to be *bound* when it occurs in the body  $t$  of an abstraction  $\lambda x.t$  (i.e.  $\lambda x$  is a binder whose scope is  $t$ ). A term with no free variables is said to be *closed*; closed terms are also called *combinators*. The simplest combinator, called the *identity* function  $\lambda x.x$  does nothing but return its argument.

After the formulation of the syntax of the language one also needs to define rules how terms can be evaluated. There are three basic approaches to define the semantics of a language.

*Denotational semantics* take meanings of terms to be mathematical objects such as numbers and functions. In order to give denotational semantics to a language one has to find collections of semantic domains and interpretation functions, which map terms into elements of the defined domains. Denotational semantics can be useful to abstract from evaluation details and to highlight essential concepts of a language. Also, properties of

the chosen collection of semantic domains can be used to derive laws for reasoning about program behavior.

*Axiomatic semantics* particularly focus attention on the process of reasoning about programs. Instead of first defining program behaviors and then deriving laws from these definitions, axiomatic methods take the laws themselves as the definition of a language.

*Operational semantics* specify program behaviors by defining abstract machines, which use language terms as their machine code. A state of a machine is just a term; a machine's behavior is defined by a transition function that, for each state of the machine, either gives the next state by performing a step of simplification on a term or declares that the machine has halted. The final state that a machine reaches when started with term  $t$  as its initial state is taken as the meaning of  $t$ .

Up to the 1980s, operational semantics were regarded as inferior to denotational and axiomatic semantics until technical problems such as the treatment of nondeterminism, concurrency, and procedures made operational semantics appear to be more attractive – especially in the light of new developments such as Plotkin's Structural Operational Semantics [Plo81], which is used in this work. In contrast to so-called big-step semantics, which directly formulate the notion of “this term evaluates to that final value”, Plotkin's small-step style of operational semantics defines evaluation relations by individual steps of computation that are necessary to rewrite a term until it eventually becomes a value – a term that has finished computing and cannot be reduced any further.

Table 1.6 defines an evaluation relation  $t \rightarrow t'$  on terms. If  $t$  is the state of the abstract machine at a given moment, then the machine can make a step of computation and change its state to  $t'$ . In its pure form, the lambda-calculus does not have numbers or built-in arithmetic operations. The only way how terms compute is the application of functions to arguments, which themselves are functions. Each step in the computation corresponds to rewriting an application, whose left-hand component is an abstraction, by substituting the right-hand component for any occurrence of the bound variable in the abstraction's

body. In this way, the application  $(\lambda x.t_{12})t_2$  evaluates in one step to  $[x \mapsto t_2]t_{12}$ , which means “the term obtained by replacing all bound occurrences of  $x$  in  $t_{12}$  by  $t_2$ ”. Following Church, a term of the form  $(\lambda x.t_{12})t_2$  is called a redex (“reducible expression”). The operation of rewriting a redex according to the above rule is called *beta-reduction*. Several different beta reduction strategies have been studied over the years. They differ in the order of the evaluation of redexes during an evaluation.

Under full beta reduction any redex may be reduced at any time while under the normal order strategy, the leftmost, outermost redex is always reduced first. The call by name strategy is even more restrictive, allowing no reductions inside abstractions. Variants of this evaluation strategy are found in Algol 60 [BBG63] and Haskell [HJW92]. In the call by value strategy only outermost redexes are reduced and a redex is reduced only when its right-hand side has already been reduced to a value. In this work, the call by value strategy is used since this strategy is the easiest to be augmented with more advanced programming language features such as exceptions.

The substitution function  $[x \mapsto s]t$  is defined inductively over its argument  $t$  where terms are worked with “up to Church’s alpha-conversion”, which includes the renaming of bound variables in order to avoid free variables becoming bound (variable capture). In particular, terms that differ only in the names of bound variables are interchangeable in all contexts. This convention renders the following definition of the substitution function “as good as total” because variables can be renamed whenever necessary.

**Definition 1.2 (Substitution)** *The substitution function  $[x \mapsto s]t$  is defined as follows.*

- $[x \mapsto s]x = s$
- $[x \mapsto s]y = y$      *if  $y \neq x$*
- $[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1$      *if  $y \neq x$  and  $y \notin FV(s)$*
- $[x \mapsto s](t_1 t_2) = [x \mapsto s]t_1 [x \mapsto s]t_2$

**Definition 1.3 (Free variables)** *The set of free variables of a term  $t$ , written  $FV(t)$ , is defined as follows.*

- $FV(x) = \{x\}$
- $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$
- $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

The lambda-calculus does not provide inherent support for multi-argument functions. Instead, higher-order functions can be used that return functions as results. In this way,  $f = \lambda(x,y).s$  can be rewritten as  $f = \lambda x.\lambda y.s$ , which is a function that, given a value  $v$  for  $x$ , yields a function that, given a value  $w$  for  $y$ , yields the desired result. The transformation of multi-argument functions into higher-order functions is called *currying* in honor of Haskell Curry, while this technique is commonly credited to Moses Isajewitsch Schönfinkel [Sch24] and Haskell Curry actually denied inventing this idea.

#### 1.4.2 The simply typed lambda-calculus

In this subsection, the untyped lambda-calculus is augmented with elementary type analysis features. Since the pure lambda-calculus is Turing complete, there is no way to implement complete type inference for all programs that may evaluate correctly at runtime (e.g., expressions may diverge, which would make the type checker diverge as well). Rather, a conservative type analysis is presented that does only make use of static information that is already available at compile time. Moreover, programs are explicitly typed. This is in contrast to implicitly typed programming languages where a type checker has to infer or reconstruct implicit type information. Instead, lambda-abstractions such as  $\lambda x.t$  are written as  $\lambda x : T_1.t_2$ , where the annotation on the bound variable gives information about of which type the argument will be. The syntax and evaluation rules of the pure simply typed lambda-calculus are given in Table 1.7. Differences to the definitions of the untyped lambda-calculus are highlighted in gray.



Table 1.7: Simply typed lambda-calculus ( $\lambda_{\rightarrow}$ )

---

*Extends Table 1.6*

---

**Syntax**

$t ::=$

$\lambda x : T.t$

*terms:*

*abstraction*

$v ::=$

$\lambda x : T.t$

*values:*

*abstraction value*

$T ::=$

$T \rightarrow T$

$A$

*types:*

*type of functions*

*atomic type*

$\Gamma ::=$

$\emptyset$

$\Gamma, x : T$

*contexts:*

*empty context*

*term variable binding*

**Evaluation**

$t \rightarrow t'$

$(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

(E-APPABS)

**Typing**

$\Gamma \vdash t : T$

$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$

(T-VAR)

$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2}$

(T-ABS)

$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$

(T-APP)

---

A typing context  $\Gamma$  is a sequence of variables and their types. The “comma” operator extends  $\Gamma$  by adding a new binding on the right. The empty context is either written as  $\emptyset$  or it is just omitted. The typing rule for abstractions is given by (T-ABS). Variables have whatever type one is currently assuming them to have in a context  $\Gamma$ .

Programming and data type languages often contain definitions of base types – sets of simple, unstructured values which are regarded as being indivisible. For example, the XML Schema Definition data type language uses 19 built-in atomic data types to represent, for instance, real numbers, strings, and Gregorian calendar dates. Table 1.7 already contains the syntactic form of uninterpreted base types where the letter  $A$  stands for atomic types. In Chapter 3, this definition will be extended with interpretations that cover constrained atomic data types.

### 1.4.3 The simply typed lambda-calculus with subtyping

A main characteristic of object-oriented languages is that an object can emulate another object that has fewer methods or fields, since the former supports the entire protocol (i.e. the interface) of the latter. Without subtyping, the well-behaved term  $(\lambda r : \{x : \mathbf{Nat}\}.r) \{x = 0, y = 1\}$ , where  $\mathbf{Nat}$  represents the type of natural numbers, would not be typeable since the given function defines a formal parameter of type  $\{x : \mathbf{Nat}\}$  while the actual type of the argument is  $\{x : \mathbf{Nat}, y : \mathbf{Nat}\}$ . However, it is intuitively safe to pass arguments of type  $\{x : \mathbf{Nat}, y : \mathbf{Nat}\}$  because the function just expects a field  $\{x : \mathbf{Nat}\}$  and does not make any assumptions what other fields the argument may or may not have.

The basic stipulations that define  $S$  as a subtype of  $T$ , written  $S <: T$ , whenever “every value described by  $S$  is also described by  $T$ ”, that is, “the elements of  $S$  are a subset of the elements of  $T$ ”, are given in Table 1.8. In particular, the subtype relation is reflexive and transitive. A maximum type  $Top$  is introduced to cover the type *Object*, which is found in most object-oriented programming languages.

Table 1.8: Simply typed lambda-calculus with subtyping  
( $\lambda_{<}$ )

---

*Extends Table 1.7*

---

**Syntax**

$T ::=$   
 $Top$

*types:*  
*maximum type*

**Typing**

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

(T-SUB)

**Subtyping**

$S <: T$

$S <: S$

(S-REFL)

$$\frac{S <: U \quad U <: T}{S <: T}$$

(S-TRANS)

$S <: Top$

(S-TOP)

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

(S-ARROW)

$\{l_i : T_i^{i \in 1..n+k}\} <: \{l_i : T_i^{i \in 1..n}\}$

(S-RCDWIDTH)

$$\frac{\text{for each } i \ S_i <: T_i}{\{l_i : S_i^{i \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}}$$

(S-RCDDEPTH)

$$\frac{\{k_j : S_j^{j \in 1..n}\} \text{ is a permutation of } \{l_i : T_i^{i \in 1..n}\}}{\{k_j : S_j^{j \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}}$$

(S-RCDPERM)

---

Also, Table 1.8 adds the subtyping rules that pertain to record data types. The so-called *width subtyping* rule (S-RCDWIDTH) for record data types captures the intuition that subtypes of record types have more fields while it is safe for supertypes to “omit” fields at the end of a record type. A longer record data type with more fields describes a more specific and thus smaller set of values. According to the *depth subtyping* rule (S-RCDDEPTH), fields of record data types may also vary as long as for each field of a supertype there is a more specific field defined for the subtype. The record permutation rule (S-RCDPERM), which says that field projection is insensitive to the order of the fields, makes the subtype relation a preorder (instead of a partial order) because there are many pairs of distinct (record-) types where each is a subtype of the other. Similar stipulations as for the subtype relation of record data types are formulated in Chapter 3 for constrained atomic data types.

#### 1.4.4 Properties of typing and subtyping

Commonly, type *safety* is referred to as the fact that well typed terms in a programming language can always be evaluated up to their final values without reaching a “stuck” state, which is not designated a final value and where no further evaluation rules can be applied. The *safety* (also called *soundness*) of the  $\lambda_{\rightarrow}$  and  $\lambda_{<}$ -calculus are shown in [Pie02] based on the *progress* and *preservation* theorems<sup>6</sup>.

**Theorem 1.1 (Progress)** *A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules): If  $t : T$  then either  $t$  is a value or else there is some  $t'$  with  $t \rightarrow t'$ .*

**Theorem 1.2 (Preservation)** *If a well-typed term takes a step of evaluation, then the resulting term is also well-typed: If  $t : T$  and  $t \rightarrow t'$ , then  $t' : T$ .*

---

<sup>6</sup>The slogan “safety is progress plus preservation” was articulated by Robert Harper; a variant was proposed in 1994 by Andrew K. Wright and Matthias Felleisen. Don Roberts noted that “static types give me the same feeling of safety as the announcement that my seat cushion can be used as a floatation device”.

These two properties guarantee that a well-typed term can never reach a *stuck* state during evaluation, where a closed term is stuck when it is in normal form but not a value. In practice, stuck states correspond to runtime errors such as exceptions or segmentation faults. For the  $\lambda_C$ -calculus introduced in Chapter 3 it will as well be shown that the *progress* and *preservation* theorems continue to hold in the presence of value space-based subtyping.



## CHAPTER 2

### The Zhi# Compiler Framework

The Zhi# compiler framework makes the C# programming language extensible with external type systems that can use a number of different subtyping mechanisms. The type system of the C# programming language implements *nominal* subtyping. In nominative type systems type compatibility is determined by explicit declarations. A type is a subtype of another if and only if it is explicitly declared to be so in its definition.

The XML Schema Definition type system extends nominal subtyping with *value space-based* subtyping. An atomic data type is a subtype of another if it is explicitly declared to be so in its definition or if its value space (i.e. the set of values for a given data type) is a subset of the value space of the other type. The subset relation of the types' value spaces is sufficient. The two types do not need to be in an explicitly declared derivation path.

In the Web Ontology Language, nominal subtyping is augmented by *ontological reasoning*. An inferred class hierarchy can include additional subsumption relations between ontological concept descriptions. Ontological individuals can be explicitly declared to be of a given type and they can be inferred to be in the extension of further concept descriptions based on, for example, particular property values or cardinalities.

Some object-oriented programming languages provide a limited set of isomorphic mappings from XSD data types to programmatic types. In general, however, compilers for programming languages such as Java or C# are unaware of the subtyping mechanisms that are used for XSD and OWL – and the list of conceivable external type systems and subtyping mechanisms can be arbitrarily extended.

The Zhi# compiler framework makes the C# programming language extensible in respect of external type definitions and subtyping mechanisms. The resulting programming language Zhi# adds a modicum of syntactical extensions to import and address external types. The Zhi# compiler framework provides two extension points for type checking and program transformation that can be implemented by compiler plug-ins for external type systems.

## 2.1 The Zhi# Programming Language

The Zhi# programming language is a proper superset of ECMA 334 standard C# version 1.0 [HWG02]. The only syntactical extensions, which are entailed by Zhi#'s extensibility with respect to external type systems, are the following.

External types, which may, for example, be declared in XSD files or OWL ontologies, can be included using the keyword *import*, which works analogously for external types like the C# *using* keyword for .NET programming language type definitions. It permits the use of external types in a Zhi# namespace such that, one does not have to qualify the use of a type in that namespace. An *import* directive can be used in all places where a *using* directive is permissible. As shown below, the *import* keyword is followed by a *type system evidence*, which specifies the external type system (i.e. compiler plug-in) that is responsible for the import of the type definitions in the given external namespace. Unlike *using* directives, the alias, which can subsequently be used to represent the external namespace name, is not optional but must be specified. Like *using* directives, *import* directives do not provide access to any namespaces that may be nested in the specified external namespace (based on the namespace scheme of the external type system).

```
import type_system_evidence alias = external_namespace;
```

In Zhi# program text that follows an arbitrary number of *import* directives, external type and property references must be fully qualified using an alias that is bound to the



namespace in which the external type is defined. Type and property references have the syntactic form `#alias#local_name` (both the namespace alias and the local name must be preceded by a '#'-symbol). It is not possible to 1) use external namespace references at arbitrary positions in Zhi# programs and 2) use unqualified external type references. External namespace references at arbitrary positions are not supported because external type systems may use arbitrary schemes to denote a namespace. However, without syntactical restrictions that are known beforehand it is not feasible to allow for such arbitrary naming schemes in the Zhi# language grammar. Even for XSD and OWL namespaces, which follow the generic syntax rules for URIs [BFM98], it may not be desirable to repeatedly use lengthy namespaces instead of handy aliases.

Finally, the C# language grammar was extended with binary comparison operators to facilitate the inference of constrained atomic XML data types (see Chapter 5). The following table lists Zhi# comparison operators with their corresponding XSD constraining facets. Constraining facets marked with an asterisk required additional operators.

Table 2.1: Zhi# comparison operators

<b>Constraining facet</b>	<b>Zhi# operator</b>	<b>Constraining facet</b>	<b>Zhi# operator</b>
<i>xsd:length*</i>	?=	<i>xsd:minLength*</i>	?>
<i>xsd:maxLength*</i>	?<	<i>xsd:pattern*</i>	??
<i>xsd:enumeration*</i>	\$=	<i>xsd:maxInclusive</i>	<=
<i>xsd:maxExclusive</i>	<	<i>xsd:minExclusive</i>	>
<i>xsd:minInclusive</i>	>=	<i>xsd:totalDigits*</i>	%%
<i>xsd:fractionDigits*</i>	%.		

External types can be used in Zhi# programs in all places where .NET types are admissible except for type declarations (i.e. external types can only be imported but not declared in Zhi# programs). For example, methods can be overridden using external types, user defined operators can have external input and output parameters, and arithmetic and logical expressions can be built up using external objects. Because Zhi#'s

support for external types is a language feature and not (yet) a feature of the runtime, similar restrictions to the usage of external types apply as for generic type definitions in the Java programming language (e.g., methods cannot be overloaded based on external types from the same type system at the same position in the method signature).

In Zhi# programs, types of different type systems can cooperatively be used in one single statement. As shown in line 5 in the following code snippet, the .NET *System.Int32* variable *age* can be assigned the XML data type value of the OWL datatype property *hasAge* that is declared for the ontological individual ALICE.

```
1 import OWL chil = http://chil.server.de;
2 class C {
3     public static void Main() {
4         #chil#Person alice = new #chil#Person("#chil#ALICE");
5         int age = alice.#chil#hasAge;
6     }
7 }
```

## 2.2 Architecture and Implementation

The Zhi# compiler framework incorporates a full-fledged source-to-source compiler for C# version 1.0 [HWG02] plus the syntactical Zhi# language extensions. The language syntax was specified with an ANTLR [Par05] grammar from which the lexer and parser were generated automatically. The ANTLR AST is transformed into a Zhi# Code DOM representation. The developed Zhi# Code DOM comprises 74 classes to represent elements of a Zhi# program. A .NET *TreeView* component was implemented to visualize the code structure of Zhi# programs on the screen and in form of printouts.

In the Zhi# programming language, it is possible to natively reference external type definitions and cooperatively use these external types along with conventional C# language features such as method overriding. As a consequence, the Zhi# compiler must be

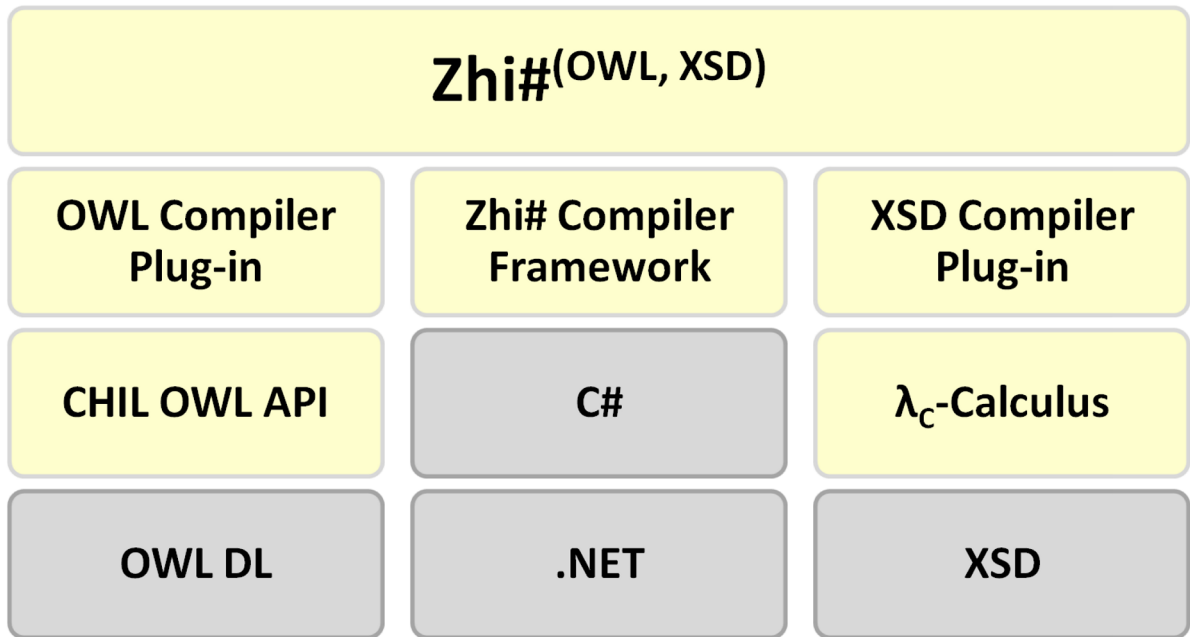


Figure 2.1: Zhi# compiler framework

extensible with respect to external typing mechanisms (e.g., subsumption, type derivation, type inference). The architecture of the Zhi# compiler on a component level is depicted in Fig. 2.1. The Zhi# compiler framework provides the core functionality to type check conventional C# programs and to transform program text into a pretty printed form. In particular, compiler components such as the type table, the symbol table, and visitors to create, traverse, analyze and modify the Code DOM representations of Zhi# programs are included in the compiler framework. These core components are implemented such that, they can be extended with support for external type systems such as XSD and OWL via two extension points in the framework for typing and subtyping (interface *IExternalTypeSystem*) and program transformation (interface *IExternalCompiler*). For example, the type table of the Zhi# compiler framework can handle method signatures that comprise external type definitions without a priori knowledge about the particular external type systems (e.g., XSD, OWL) that will eventually be used. On the other hand, compiler plug-ins for external type systems can be developed without exhaustive knowledge about the code structure of Zhi# programs.

Compiler plug-ins provide type checking and program transformation functionality in order to use C# programming language features with type definitions and subtyping mechanisms of external type systems. Up to now, two compiler plug-ins were implemented to support the value space based subtyping of the XML Schema Definition type system [BM04b] and subtyping based on ontological reasoning as defined by the Web Ontology Language [PHH04] in addition to the nominal subtyping of C#.

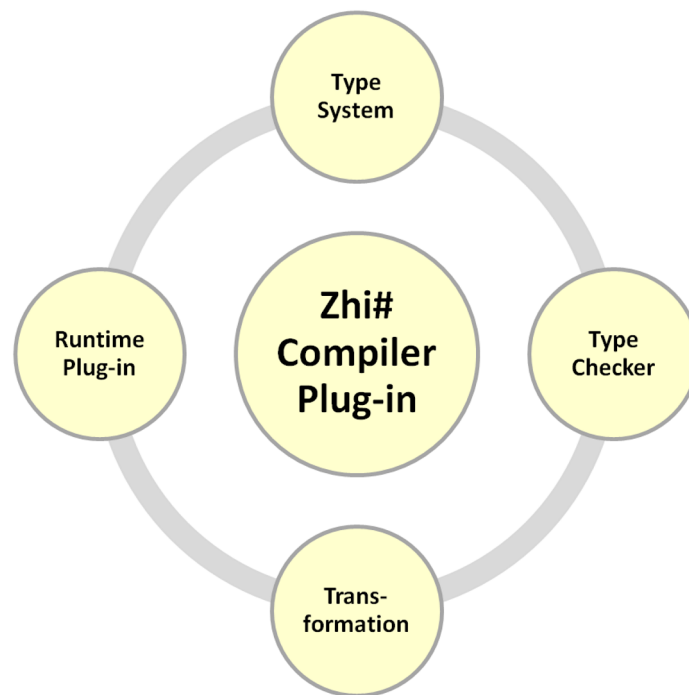


Figure 2.2: Zhi# compiler plug-in

As shown in Fig. 2.2 a Zhi# compiler plug-in comprises an implementation of the corresponding external type system. This implementation is effectively a classifier, which computes the subsumption hierarchy of the external types. The subsumption hierarchy is used by the external type checker, which computes the types of Zhi# expressions that reference external types (e.g., member access expressions, binary expressions) and reports ill-typed terms. Well-typed Zhi# expressions that involve external types are transformed into conventional C# code that references the associated runtime plug-in for the external type system. Section 2.3 elucidates the type checking and program transformation

extension points of the Zhi# compiler framework. The type derivation and type checking features of XML Schema Definition were implemented based on the  $\lambda_C$ -calculus (see Chapter 3). Ontologies are processed using an implementation of the CHIL OWL API (see Chapter 4). These libraries are required both at compile time as well as at runtime. At compile time, the Zhi# compiler plug-ins utilize these libraries to provide type inference and type checking for XSD and OWL type definitions as shown in Fig. 2.3.

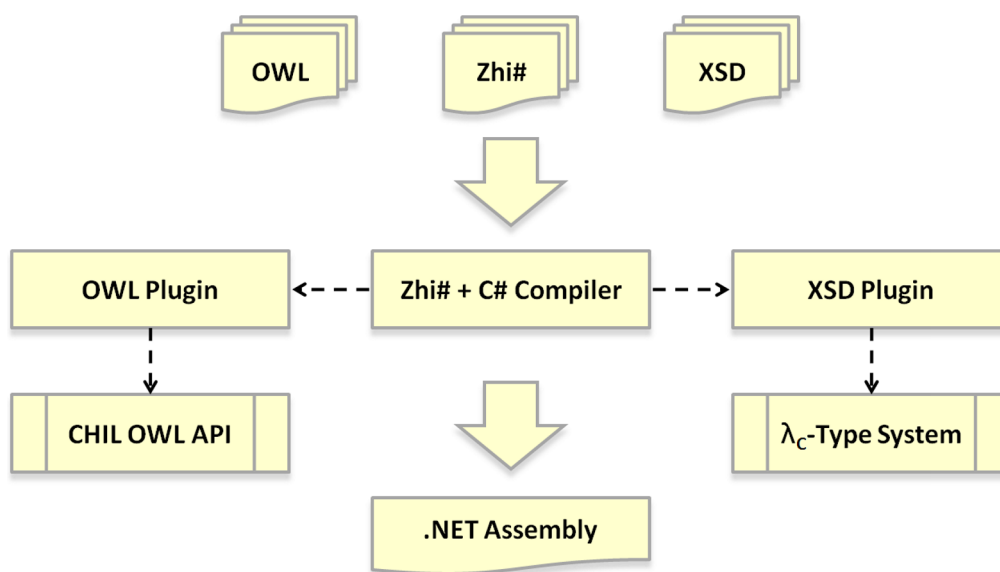


Figure 2.3: Compilation of Zhi# programs

At runtime, compiled Zhi# code (i.e. plain .NET IL bytecode) uses the Zhi# runtime library as shown in Fig. 2.4. The Zhi# runtime library comprises a core library that provides general functionality such as for the parsing of type names. The core library, which is implemented as a partial static class, can be supplemented by type system specific runtime libraries. The runtime library for XML Schema Definition uses the implementation the  $\lambda_C$ -calculus to mimic the behavior of constrained XSD data types. The runtime library for the Web Ontology Language uses the CHIL OWL API to process OWL DL knowledge bases. In contrast to approaches that are based on wrapper classes or additional code generation, the program overhead of compiled Zhi# programs is constant and does not grow with, for example, the number of referenced XSD or OWL types.

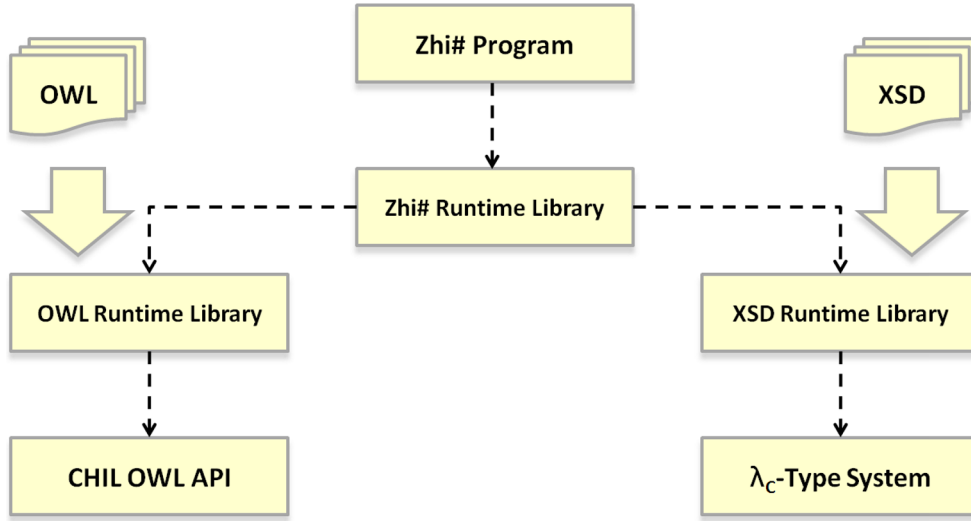


Figure 2.4: Execution of Zhi# programs

Fig. 2.5 depicts the architectural model of the Zhi# compiler framework on a class level. The *ZhiSharpCompiler* class aggregates all components of the framework. Zhi# source code and references to included .NET assemblies are passed into the *ZhiSharpCompiler* as *ZhiSharpSourceObject* and *LibraryFile* objects. Zhi# source code is tokenized and transformed into an AST by the *ZhiSharpLexer* and *ZhiSharpParser*, which were automatically generated based on an ANTLR [Par05] grammar of the Zhi# language specification. A Zhi# Code DOM representation is built from the ANTLR AST. The semantic analysis of input compilation units is performed on these Code DOM representations. Program transformation is implemented as a Code DOM transformation with subsequent transformation independent pretty printing (i.e. serialization of a Code DOM to program text). A *TypeTable*, which comprises type definitions of input Zhi# source code and referenced .NET assemblies, is built by the *TCVisitorPass0Types* and *TCVisitorPass0Members* visitors. Type names in method and property bodies are resolved by the *TCVisitorPass0Bodies* visitor. The *TCVisitorPass1* visitor performs the actual type checking of the Zhi# code, which includes the application of type inference mechanisms of activated *IExternalTypeSystems*. Note that the presence of implementations of such external type systems (e.g., *ExternalTypeSystemXSD*) is transparent to the

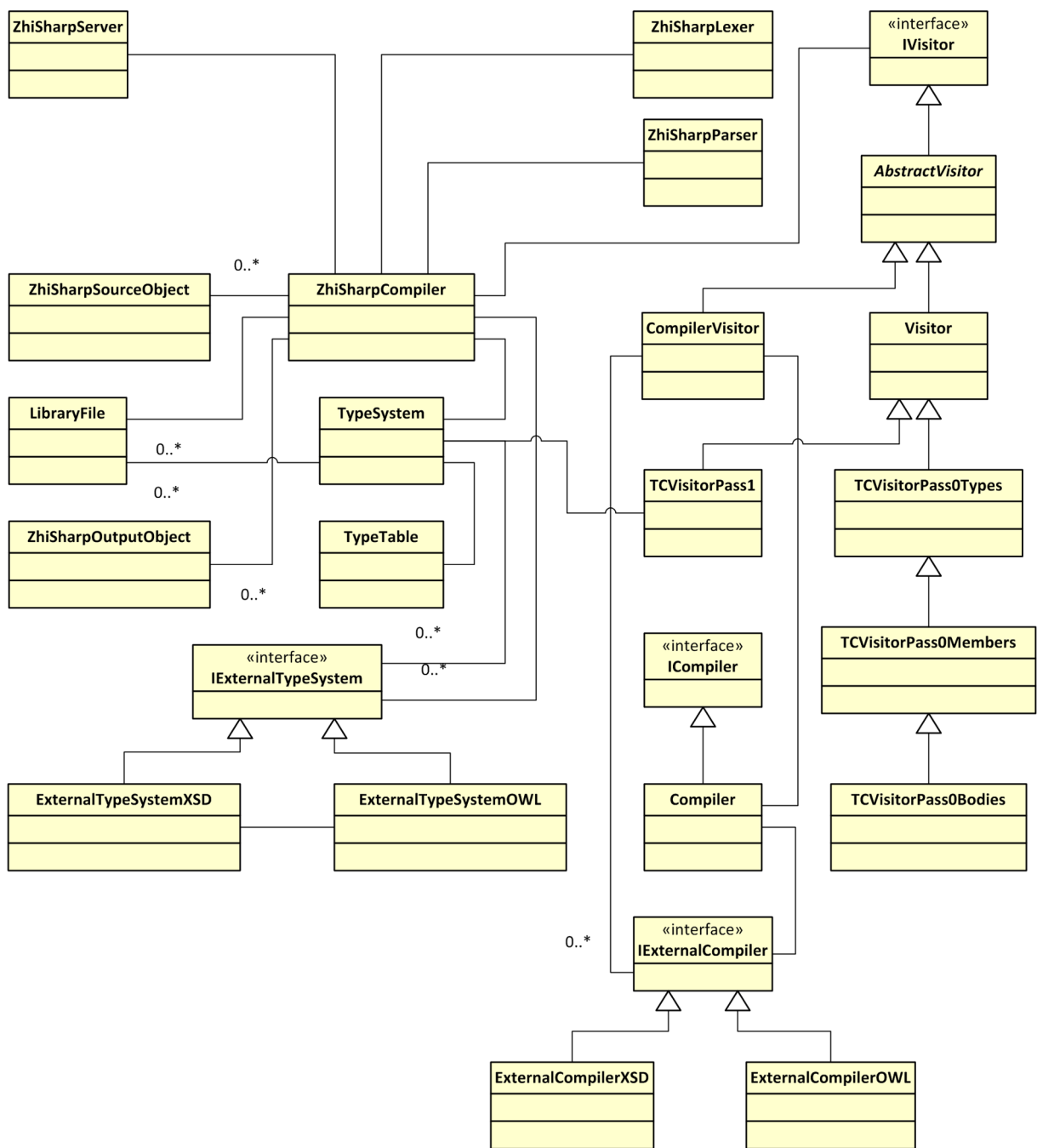


Figure 2.5: Zhi# compiler framework (class level)

*TCVisitorPass1* visitor since only the type system independent interfaces of the *TypeSystem* and *TypeTable* classes are used. Also, external type systems may cooperate in order to resolve, for example, member access expressions where the member of an external type is defined in another external type system (e.g., OWL datatype properties). Successfully type checked Zhi# code is transformed into conventional C# code by the *CompilerVisitor*. Again, a type system independent *Compiler* is used to make activated implementations of the *IExternalCompiler* interface such as *ExternalCompilerXSD* transparent to the Zhi# compiler framework.

The compilation of a Zhi# program into conventional C# is a six phase algorithm. Firstly, in-memory representations of external type definitions are created by the activated compiler plug-ins. The Zhi# compiler takes as input not only program source code and referenced .NET assemblies but also files with arbitrary content. These files are passed to the compiler plug-ins, which can opt to interpret the file content as type definitions. In practice, the file name extensions are used by the compiler plug-ins to identify relevant files. Secondly, a type table is built that contains all type and member declarations of input Zhi# compilation units and referenced .NET libraries. At this point, member declarations may already be based on external type definitions (e.g., a method can be declared that takes an XML data type value and returns an ontological individual). Thirdly, input compilation units are semantically analyzed. This includes type inference and type checking. Type inference is contributed by the compiler plug-ins. Thus, type system specific constraints can be added to the genuine type representations that are managed by the core compiler components. For example, for a string literal "hello" in a Zhi# program the XSD compiler plug-in infers the value space constraints *xsd:minLength = 5*, *xsd:length = 5*, and *xsd:maxLength = 5*, which are added to the type information of the string literal that is managed by the compiler framework. Type checking is augmented by the compiler plug-ins where needed for external types. Fourthly, Zhi# expressions that contain external type references are transformed into conventional C# by the respective compiler plug-ins for the external type definitions. These compiler plug-ins do not require



knowledge about the complete Zhi# language grammar. For the most part, only the syntax of object creation (i.e. keyword *new*), member access (i.e. the “.” operator), and binary expressions (e.g., assignments) will be necessary. Fifthly, the transformed Code DOM, which at this point does no longer contain external type references or Zhi# specific notations, is serialized to C#. Finally, the generated C# code, which is correct by generation, is compiled into a .NET assembly and linked with the Zhi# runtime library using Microsoft’s `csc` C# compiler. A Microsoft `msbuild` task was implemented to make the compilation of Zhi# programs into .NET assemblies actually a one-step process for developers. The build process is made even more transparent by the Eclipse-based Zhi# frontend (see Appendix A.2).

The Eclipse-based frontend boasts an Eclipse project type that includes a Zhi# editor with syntax highlighting and autocompletion for XSD and OWL type definitions and pretty printing of Zhi# source code. XML schemas (i.e. .xsd-files) and OWL ontologies (i.e. .rdf/.owl-files) that are comprised by a Zhi# project are automatically passed into the Zhi# compiler along with the actual Zhi# source files.

## 2.3 Framework Extension Points

The architectural model of the Zhi# compiler framework was designed for easy extensibility. Most of the program analysis tasks that require knowledge about the possibly complex code structure of C# programs were factored out of the framework extension points for typing and program transformation. Plug-ins for the Zhi# compiler framework can be developed without exhaustive knowledge about the Zhi# language grammar. The other way around, no a priori knowledge about external type systems is required by the compiler framework.

As a matter of course, the separation of the plug-ins from each other and from the compiler framework is also likely to facilitate maintainability and configurability since different features can be developed further and used independently.

### 2.3.1 Typing extension point

Table 2.2 lists the methods of the *IExternalTypeSystem* extension point (formal parameters and return types are omitted for brevity). There are three categories of methods.

Table 2.2: The *IExternalTypeSystem* extension point

	<b>Property/Method name</b>	<b>Description</b>
a)	<i>Evidence</i>	Gets the type system evidence of the current type system.
a)	<i>Name</i>	Gets the name of the current type system.
a)	<i>SetTypeSystem()</i>	Sets the associated type system of the current external type system.
a)	<i>AddExternalTypeSystem()</i>	Adds an associated external type system of the current type system.
a)	<i>LoadLibraryFiles()</i>	Loads library files (.dll, .xml, .owl-files).
a)	<i>ImportedNamespaces</i>	Gets the imported external namespaces.
a)	<i>AddImportedNamespace()</i>	Adds an imported external namespace.
a)	<i>ClearImportedNamespaces()</i>	Clears the list/dictionary of the imported external namespaces.
a)	<i>ResolveType()</i>	Resolves a specified type.
a)	<i>TypeNames</i>	Gets the external type names.
a)	<i>Properties</i>	Gets the external property names.
b)	<i>IsNonArrayNumericType()</i>	Determines whether an external type is a non-array numeric type.
b)	<i>IsNonArrayIntegerType()</i>	Determines whether an external type is a non-array integer type.
b)	<i>IsNonArrayStringType()</i>	Determines whether an external type is a non-array string type.
b)	<i>GetDirectBaseTypes()</i>	Gets the direct base types of an external type.
b)	<i>GetPrimitiveBaseTypes()</i>	Gets the primitive base types of an external type.
b)	<i>GetSupertypes()</i>	Gets the supertypes of an external type.
b)	<i>GetEquivalentTypes()</i>	Gets the equivalent types of an external type.
b)	<i>GetDisjointTypes()</i>	Gets the disjoint types of an external type.
b)	<i>GetSubtypes()</i>	Gets the subtypes of an external type.
b)	<i>IsSubtypeOf()</i>	Determines whether a type is a subtype of another type.
b)	<i>IsSupertypeOf()</i>	Determines whether a type is a supertype of another type.
c)	<i>GetLiteralConstraints()</i>	Gets the constraints for a literal node.
c)	<i>GetTypeOfPropertyAccess()</i>	Gets the type of an external property access.
c)	<i>GetTypeOfMethodInvocation()</i>	Gets the type of an external method invocation.
c)	<i>GetTypeOfIndexerAccess()</i>	Gets the type of an external indexer access.
c)	<i>CheckAssignment()</i>	Checks an assignment.
c)	<i>CheckISCompatibility()</i>	Checks the <i>is</i> compatibility.
c)	<i>CheckObjectCreationExpression()</i>	Checks an external object creation expression.
c)	<i>CheckBinaryExpression()</i>	Checks a binary expression (includes type inference).

There are technical methods, marked with an a) in the above list, for initializing the type system component of a compiler plug-in (e.g., loading type definitions from files) and yielding information about the managed type information. Methods marked with a b) in

the list provide the actual (sub-)typing functionality of an external type system. These methods likely require additional libraries that implement possibly complex classifiers and deduction engines (e.g., Description Logic reasoners). Note that there are not only methods that return the subtypes and supertypes of a given type but also equivalent and disjoint types. The third category of methods, marked with a c), is based on category b) methods and infers and checks the types of basic Zhi# terms such as assignments or member access expressions.

### 2.3.2 Program transformation extension point

Zhi# expressions that refer to external objects (e.g., ontological individuals) must be transformed into conventional C# where external type and object references are replaced by program code that utilizes runtime libraries, which manage the external data (e.g., ontological knowledge bases). In addition, as is the case for the XSD and OWL compiler plug-ins, dynamic type checks can be performed by the generated code and the runtime libraries.

Just like native .NET types, external type references can only occur in particular contexts in Zhi# programs. Also, given two cooperating Zhi# compiler plug-ins, only one plug-in may be responsible for handling expressions in which types from both type systems occur. As a consequence, it would be impractical if a compiler plug-in would have to implement a program transformation interface that covers the entire specification of the Zhi# programming language. Instead, the Zhi# compiler analyzes and pre-processes expressions that contain external types such that, external compiler plug-ins do only have to implement 14 program transformation functions. These functions transform Zhi# source code that contains external type references into conventional C# code. Table 2.3 lists the methods that must be implemented by compiler plug-ins for the *IExternalCompiler* extension point. Technical methods, marked with an a), are to initialize the program transformation component of a compiler plug-in and to yield information about the used

type system. Methods marked with a b) generate the C# expressions (i.e. Code DOM nodes) that are substituted for Zhi# terms that comprise external type and object references.

Table 2.3: The *IExternalCompiler* extension point

	<b>Property/Method name</b>	<b>Description</b>
a)	<i>Evidence</i>	Gets the type system evidence of the current compiler.
a)	<i>Name</i>	Gets the name of the type system of the current compiler.
a)	<i>SetTypeSystem()</i>	Sets the host language type system.
a)	<i>SetExternalTypeSystem()</i>	Sets the external type system of the current compiler.
b)	<i>UseNamespace()</i>	Returns an expression that imports the required namespace for an external namespace.
b)	<i>GetProxyType()</i>	Gets a host language proxy type for an external type.
b)	<i>Cast()</i>	Returns an expression that casts an expression to the proxy type of its type.
b)	<i>CreateObject()</i>	Returns an expression that creates an object from specified expressions.
b)	<i>GetObject()</i>	Returns an expression that gets an object from specified expressions.
b)	<i>CreateArray()</i>	Returns an expression that creates an array from specified expressions.
b)	<i>Assign()</i>	Returns an expression that assigns a source expression to a target expression.
b)	<i>Compute()</i>	Returns an expression that computes a binary expression.
b)	<i>PropertyAccess()</i>	Returns an expression that accesses a property of an external host object.
b)	<i>PropertyUpdate()</i>	Returns an expression that updates a property of an external host object.
b)	<i>IndexerAccess()</i>	Returns an expression that accesses an indexer of an external host object.
b)	<i>IndexerUpdate()</i>	Returns an expression that updates an indexer of an external host object.
b)	<i>MethodInvocation()</i>	Returns an expression that invokes a method on an external host object.
b)	<i>ForEach()</i>	Returns an expression that iterates over an iterator expression.

The *IExternalCompiler* interface is for the most part context-free. Only 14 methods that perform code transformations on the Zhi# Code DOM were necessary in order to cope well with constrained XML data types and complex OWL concept descriptions alike. The *UseNamespace* method returns a Code DOM node that imports the required .NET namespace (e.g., the namespace of the CHIL OWL API) for a specified external namespace (e.g., a namespace of an OWL ontology). The *GetProxyType* method gets the host language proxy type for a specified external type. An expression that casts a specified expression to the proxy type of a specified target type is returned by the *Cast*

method. Expressions that create and get an object and create an array from a specified list of expressions are returned by the *CreateObject*, *GetObject*, and *CreateArray* methods, respectively. The *Compute* method returns an expression that computes the specified external binary expression. The special case of an assignment expression is handled by the *Assign* method, which returns an expression that assigns a source expression to a target expression. Methods for property access and property updates, for C#-like indexer access and indexer updates, and for invoking methods on external host objects are provided in order to allow for full-fledged external objects. The *ForEach* method returns an expression that iterates over a given iterator expression.

The program transformation functions of the *IExternalCompiler* interface were specified in the following premise-conclusion form.

$$\frac{\Gamma, x_i : T_j \quad i \in 1..l, j \in 1..m \vdash T_j \in \Delta_k \quad j \in 1..m, k \in 1..n}{f_{\Delta}(x_1, \dots, x_l) = \begin{cases} \text{case 1} \\ \vdots \\ \text{case } u \end{cases}}$$

A particular implementation of a program transformation function is applicable only in a context that is given by its premise. The premise of a program transformation function comprises a typing context  $\Gamma$  and a number of type system evidences.

The *typing context*  $\Gamma$  is a sequence of variables and their types. The “comma” operator extends  $\Gamma$  by adding a new binding on the right. The empty typing context  $\emptyset$  denotes that there are no assumptions about type bindings.

In addition to the typing context, the second relevant context part are the *type system evidences* of the types used in the current type environment. Type system evidences identify types as being defined in a certain (external) type system. Type system evidences are used internally by the Zhi# compiler framework to select the appropriate type system and compiler plug-in for an expression of a certain type.

The notion that a type  $T$  is defined in the type system  $TS$  is written as  $T \in \Delta_{TS}$ . Accordingly, type  $T$ 's type system evidence is ' $TS$ '. Analogously, the fact that a namespace  $N$  is defined in the type system  $TS$  is written as  $N \in \blacktriangle_{TS}$ . The implementation of a program transformation function  $f$  that is provided by the compiler plug-in for type system  $TS$  is denoted  $f_{TS}$ . The indices  $\cdot_{NET}$  and  $\cdot_{ETS}$  denote the .NET type system and a generic external type system, respectively. The domains of non-array and array types of a type system  $TS$  are written as  $\Delta_{TS}^{\perp}$  and  $\Delta_{TS}^{[\ ]}$ , respectively, where  $\Delta_{TS} = \Delta_{TS}^{\perp} \cup \Delta_{TS}^{[\ ]}$ .

Note that the schemata of the function premises only contain generic external types. The decision *which* program transformation function is to be applied is taken without dependence on one particular external type system whereas the selection of the particular function *implementation* is based on concrete external types. The external compiler plug-in that provides the actual implementation of the function is selected based on the particular type system evidences of the types of the term under consideration.

The implementations of program transformation functions are given as case-statements where the associated value of the function is given for a number of cases. The case-statements are considered in order (i.e. for the sake of brevity case specifications may be complete only for their particular positions in the lists of cases).

The program transformation functions operate on the source code of Zhi# programs. The output is either type information or conventional C# source code. External program transformation functions may generate C# code that calls functions of the Zhi# runtime library. The Zhi# runtime library is extended with a library component for each external type system being used. Runtime library functions are given in the form  $f_{TS}(\dots)$  where  $TS$  denotes an (external) type system whose associated runtime library provides the function  $f$ . A compiler plug-in for an external type system may refer to functions that are provided by the runtime library of a different type system. In particular, compiled Zhi# code can make use of the .NET Base Class Library. In the following paragraphs the applicability rules of the 14 category b) program transformation methods of Table 2.3 are given.

**UseNamespace** The *UseNamespace* function returns an expression that imports the required .NET namespace for the specified external namespace. Typically, the import of an external namespace in a Zhi# program results in the import of the namespace of the associated runtime library for this external type system 'ETS'. For example, OWL namespace imports are substituted by an import of the .NET namespace of the Zhi# runtime library for the Web Ontology Language.

$$\frac{\emptyset \vdash \textit{namespace} \in \blacktriangle_{\text{ETS}}}{\textit{UseNamespace}_{\text{ETS}}(\textit{namespace}) = \left\{ \begin{array}{l} \\ \end{array} \right. \vdots}$$

**GetProxyType** The *GetProxyType* function yields the runtime proxy type for the specified external type  $T$ . The proxy type must be defined in the runtime library for the external type system 'ETS'.

$$\frac{\emptyset \vdash T \in \Delta_{\text{ETS}}}{\textit{GetProxyType}_{\text{ETS}}(T) = \left\{ \begin{array}{l} \\ \end{array} \right. \vdots}$$

**Cast** The *Cast* function returns an expression that implements a cast of the specified expression  $v$  to the proxy type of the specified target type  $T$ . The proxy type of a .NET type is the type itself.

$$\frac{\Gamma, v : V \vdash (V \in \Delta_{\text{ETS}} \vee T \in \Delta_{\text{ETS}})}{\textit{Cast}_{\text{ETS}}(T, v) = \left\{ \begin{array}{l} \\ \end{array} \right. \vdots}$$

**CreateObject** The *CreateObject* function returns an expression that creates an object from the specified expressions  $\{v_i\}_{i \in 1..n}$ . This object is an instance of the proxy type of the specified type  $T$ . Note that  $n$  here is allowed to be 0. In this case, the range  $1..n$  is empty and  $v_i\}_{i \in 1..n}$  is  $\{\}$ , which stands for an empty argument list (i.e. a parameterless constructor call).

$$\frac{\Gamma, v_i : V_i\}_{i \in 1..n} \vdash (T \in \Delta_{\text{ETS}}^{\perp} \vee \bigvee V_i \in \Delta_{\text{ETS}})}{\textit{CreateObject}_{\text{ETS}}(T, \{v_i\}_{i \in 1..n}) = \left\{ \begin{array}{l} \\ \end{array} \right. \vdots}$$

**GetObject** The *GetObject* function returns an expression that gets an object from the specified expressions  $\{v_i^{i \in 1..n}\}$ . This object is an instance of the proxy type of the specified external type  $T$ . Note that  $n$  here is allowed to be 0. In this case, the range  $1..n$  is empty and  $v_i^{i \in 1..n}$  is  $\{\}$ , which stands for an empty argument list (i.e. a parameterless constructor call).

$$\frac{\Gamma, v_i : V_i^{i \in 1..n} \vdash (T \in \Delta_{\text{ETS}}^{\perp} \vee \bigvee V_i \in \Delta_{\text{ETS}})}{\text{GetObject}_{\text{ETS}}(T, \{v_i^{i \in 1..n}\}) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**CreateArray** The *CreateArray* function returns an expression that creates an array from the specified expressions  $\{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}$ . This array is an instance of the proxy type of the specified external type  $T$ .

$$\frac{\Gamma, v_{ij} : V_{ij}^{i \in 1..n, j \in 1..m} \vdash T \in \Delta_{\text{ETS}}^{[\ ]}}{\text{CreateArray}_{\text{ETS}}(T, \{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**Assign** The *Assign* function is applied on terms of the form  $lvalue = rvalue$ . It yields an expression that executes the assignment at runtime. The *lvalue* is an expression that can appear as the destination of an assignment operator indicating where an *rvalue* should be stored. In .NET, an *lvalue* may be a variable or a field or an (indexed) property of a .NET host object. The Zhi# compiler first tries to select the external compiler plug-in that provides the applicable implementation of the *Assign* function based on the type system evidence of type  $T$ . If  $T$  is a .NET type, the external compiler plug-in is selected based on the type system evidence of type  $V$ .

$$\frac{\Gamma, lvalue : T, rvalue : V \vdash (T \in \Delta_{\text{ETS}} \vee V \in \Delta_{\text{ETS}})}{\text{Assign}_{\text{ETS}}(T, lvalue, rvalue) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**Compute** The *Compute* function returns an expression that computes the binary expression  $x \tau y$ . The returned expression is an instance of the (proxy type of the) specified



(external) type  $T$ . If no applicable user defined operator exists in the common type table, the Zhi# compiler first tries to select the external compiler plug-in that provides the applicable implementation of the *Compute* function based on the type system evidence of type  $X$ . If  $X$  is a native .NET type, the external compiler plug-in is selected based on the type system evidence of type  $Y$ . Expressions of the form  $x \tau= y$ , where  $\tau=$  shall be an arbitrary assignment operator (e.g.,  $+=$ ), are converted into the explicit form  $x = x \tau y$  and transformed using the *Compute* and *Assign* transformation functions.

$$\frac{\Gamma, x : X, y : Y \vdash (X \in \Delta_{\text{ETS}} \vee Y \in \Delta_{\text{ETS}})}{\text{Compute}_{\text{ETS}}(T, x, \tau, y) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**PropertyAccess** The *PropertyAccess* function returns an expression that accesses the property  $p$  (the meta-property  $q$ ) of the external host object  $o$  (of the external property  $p$ ). The property access is cast to the proxy type of the specified external type  $T$  (external host objects cannot contain .NET objects).

$$\frac{\Gamma, o : O \vdash O \in \Delta_{\text{ETS}}^{\perp}}{\text{PropertyAccess}_{\text{ETS}}(T, o, p, q) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**PropertyUpdate** The *PropertyUpdate* function returns an expression that updates the property  $p$  of the external host object  $o$  with value  $v$ . The property access is cast to the proxy type of the specified external type  $T$  (external host objects cannot contain .NET objects) in order to allow for cascaded assignments. Note that the semantics of a property update depends on type  $O$  and property  $p$  of the used type system. For example, in Zhi# programs, an assignment to a functional OWL datatype property *updates* the property value of the specified individual while an assignment to a non-functional OWL datatype property *adds* the specified value.

$$\frac{\Gamma, o : O, v : V \vdash O \in \Delta_{\text{ETS}}^{\perp}}{\text{PropertyUpdate}_{\text{ETS}}(T, o, p, v) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**IndexerAccess** The *IndexerAccess* function returns an expression that accesses the indexer with arguments  $\{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}$  of the external host object  $o$ . The indexer access is cast to the proxy type of the specified external type  $T$  (external host objects cannot contain .NET objects).

$$\frac{\Gamma, o : O \vdash O \in \Delta_{\text{ETS}}^\perp}{\text{IndexerAccess}_{\text{ETS}}(T, o, \{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**IndexerUpdate** The *IndexerUpdate* function returns an expression that updates the indexer with arguments  $\{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}$  of the external host object  $o$  with value  $v$ . The indexer access is cast to the proxy type of the specified external type  $T$  (external host objects cannot contain .NET objects) in order to allow for cascaded assignments.

$$\frac{\Gamma, o : O \vdash O \in \Delta_{\text{ETS}}^\perp}{\text{IndexerUpdate}_{\text{ETS}}(T, o, \{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}, v) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**MethodInvocation** The *MethodInvocation* function returns an expression that invokes the method  $m$  with arguments  $\{v_i^{i \in 1..n}\}$  on the external host object  $o$  (on the external property  $p$ ). The return value is cast to the (proxy type of the) specified (external) type  $T$ .

$$\frac{\Gamma, o : O \vdash O \in \Delta_{\text{ETS}}^\perp}{\text{MethodInvocation}_{\text{ETS}}(T, o, p, m, \{v_i^{i \in 1..n}\}) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

**ForEach** The *ForEach* function returns an expression that iterates over the object  $v$ . The yielded elements are cast to the proxy type of the specified external type  $T$ .

$$\frac{\Gamma, v : V \vdash (V \in \Delta_{\text{NET}}^\perp \wedge T \in \Delta_{\text{ETS}})}{\text{ForEach}_{\text{ETS}}(T, v) = \left\{ \begin{array}{l} \vdots \end{array} \right.}$$

See Appendix B.1 and B.2 for XML Schema Definition and Web Ontology Language specific implementations of the program transformation functions, respectively.

## 2.4 Type System Cooperation

The Zhi# compiler framework facilitates the cooperative usage of type definitions from different type systems (e.g., OWL datatype properties can be assigned XML data type values). Type checking and program transformation tasks are delegated to those compiler plug-ins that are responsible for the particular external parts of an expression. The selection of these compiler plug-ins is controlled by the classification of types into particular type systems. Internally, type names are classified into type systems based on their *type system evidences*. These evidences are initially assigned to type names by the *import* statement (see Section 2.1).

For member access expressions the type checking and program transformation order is top-down (i.e. from left to right within the expression). For compound terms the type checking and program transformation order is bottom-up (i.e. from inner expressions to outer expressions). For binary expressions that contain external objects, first the compiler plug-in for the type of the left operand is consulted. If the left operand is a .NET object or if the consulted external type system fails to type check the binary expression, the compiler plug-in for the type of the right operand is used. If this fails again the binary expression is rejected by the Zhi# type checker.

Zhi# programs with no external type references (i.e. conventional C# code) are exhaustively checked by the Zhi# compiler, too. An input Zhi# program is either rejected by the Zhi# type checker or the generated C# code is well-typed. Zhi# expressions that do not contain external objects are not subject to program transformations except for pretty-printing to the output C# code of the Zhi# compiler.

In the Zhi# code snippet below, a .NET and an XSD byte<sup>1</sup> variable are defined in line 1 and 2, respectively. The Zhi# type checker validates that the internal .NET type *byte* (i.e. *System.Byte*) exists in the imported .NET namespaces. Based on the type system evidence of the external type *xsd#byte* the XSD compiler plug-in is consulted to check

---

<sup>1</sup>In .NET a *byte* is an unsigned 8-bit integer, in XSD a *byte* ranges from -128 to +127.

the availability of this type definition in one of the imported XSD namespaces. For line 3, the XSD compiler plug-in is utilized to check that the type name *xsd#short*, which is used for the declaration of variable *c*, exists in one of the imported XSD namespaces. Next, the XSD compiler plug-in is used to compute the type of the binary expression *a+b* to be *xsd#decimal{%. 0}{>= -128}{<= 382}*<sup>2</sup> (a number with no fraction digits that is greater than or equal to -128 and less than or equal to 382, see Section 3.7 for a description of the XSD constraint arithmetic). Finally, the assignment of the binary expression *a+b* to *c* is validated based on the subsumption relation between the two types *xsd#short* (i.e. a 16-bit signed integer) and *xsd#decimal{%. 0}{>= -128}{<= 382}*.

```

1 byte a = 0;
2 #xsd#byte b = 0;
3 #xsd#short c = a + b;

```

For plain external variables and external members of .NET host objects, the type of the variable itself and of the complete compound expression, respectively, determines which external compiler plug-in is to be used for type checking and program transformation. For property access expressions of external host objects, the context of the member access has to be taken into account. If an expression of the form *externalHostObject.externalProperty* is used as an lvalue, the compiler plug-in for the type of the external host object is responsible for handling access to the specified property. This is the case in line 3 in the following code snippet. Variable *r* is declared to refer to an instance of the OWL class *Room*. Although the type (i.e. the range) of its functional property *hasCapacity* – and therefore the type of the whole member access expression – shall be *xsd#short*, only the OWL compiler plug-in can generate code that sets the value of property *hasCapacity* of individual ROOM248 accordingly. In line 4, the external member access expression is used as an rvalue. In this case, type checking and transformation of the binary expression *r.#ont#hasCapacity - 2* is delegated to the XSD compiler plug-in based on the XSD type of the compound left operand. The XSD compiler plug-in uses a reference to the compiled

---

<sup>2</sup>In Zhi#, type names can be followed by constraint notations of the form *{facet literal}*.

OWL member access expression in order to apply its program transformations to the complete binary expression. In order to keep the compilation of external member access expressions context-free with respect to their use as source values or target locations, both the lvalue and rvalue forms are created and stored as attributes of the member access expression node in the Zhi# Code DOM.

```

1 #xsd#short c = 8;
2 #ont#Room r = new #ont#Room("#ont#ROOM248");
3 r.#ont#hasCapacity = c;
4 int i = r.#ont#hasCapacity - 2;

```

Each binary expression in the above code snippet is compiled by the Zhi# compiler component that is responsible for the respective type system. The preceding listing would be compiled into the following C# code (for the sake of brevity, only the aliases “ont” and “xsd” are used instead of the fully qualifying OWL and XSD namespaces; .NET namespaces are omitted for types and static methods of the Zhi# runtime library).

```

1 RTSimpleType c = NewXSD("xsd#short", 8);
2 OWLIndividual r = AssertKindOf(CreateOWLIndividualByName("ont#Room248", "ont#Room"), "ont#Room");
3 SetOWLDatatypePropertyValue(AssertKindOf(r, "ont#Room"), "ont#hasCapacity", c);
4 System.Int32 i =
5     Convert.ToInt32((Subtraction("xsd#decimal{\\%. \\0\\"}{> \\-32771\\"}
6         {>= \\-32770\\"}{<= \\32765\\"}{< \\32766\\"}",
7             GetFunctionalOWLDatatypePropertyValue(AssertKindOf(r, "ont#Room"),
8                 "ont#hasCapacity",
9                 "xsd#short"),
10             NewXSD("System.Int32{> \\1\\"}{>= \\2\\"}{<= \\2\\"}{< \\3\\"}", 2))
11     ).ToString();

```

The code above uses static functions defined in the partial static class *ZhiSharpRuntime* of the Zhi# runtime library and its XSD and OWL specific extensions. The definition of the partial *ZhiSharpRuntime* class can be split over two or more source files (cf. C# keyword *partial*). Thus, its members could be declared private and still be visible to compiler plug-in implementors. Note that all external type system functionality that is provided by compiler plug-ins and extensions of the Zhi# runtime library is used in a “pay as you go” manner. In Zhi#, there is no performance or code size overhead for conventional C# code and Zhi# programs that do not use external type definitions.

## 2.5 Related Work

The Zhi# compiler framework makes external type systems *pluggable*. In [Bra04], Gilad Bracca argues in favor of Curry-style language definitions where a type system is neither syntactically nor semantically required. In fact, the evaluation rules of most common theoretical models of programming languages such as the  $\lambda$ -calculus do not depend on type rules. Still, Zhi# – as is conventional C# – is given in the Church-style where typing is prior to semantics (i.e. the behavior of ill-typed terms is undefined). In particular, the .NET type systems remains mandatory for .NET objects in Zhi# programs. External objects such as XML data type values or ontological individuals can only be used in the presence of the respective external type system (i.e. compiler plug-in). Also, XML data type inference is not optional with the type checker. The Church-style language specification of Zhi#, which is based on mandatory type systems, allows for usual programming techniques such as class-based encapsulation and static type-based overloading. Class-based encapsulation would be prohibitively expensive with optional type systems, where object-based encapsulation such as in Self or Smalltalk would be used instead. Andreae et al. developed a framework for implementing pluggable type systems for the Java programming language [ANM06]. Their JAVACOP system enforces user-defined typing constraints of Java types (e.g., *nonnull*, *confined*) written in a declarative rule language. In contrast, the Zhi# framework does not facilitate the *implementation* of external type systems but their incorporation with the type checker of the Zhi# programming language. It is hard to imagine to model the XSD and OWL classifier and deduction algorithms and their interplay with the Zhi# type checking in the JAVACOP rule language. Also, the JAVACOP framework does not support type inference and requires basic knowledge of the code structure of Java programs.

The described compiler framework facilitates the incorporation of schema and ontology languages – in form of type definitions – with the C# programming language. An inverse technique is, for example, the integration of Turing-complete JavaScript with XHTML

documents. Both approaches are asymmetric in a sense that a full-fledged programming language is augmented with external type definitions or vice versa. One can also find in a number of application domains examples of slightly more symmetric approaches such as, for example, the integration of SQL queries with C code. In practice the degree of integration is inversely related to the symmetry of the integration problem.

The Zhi# compiler framework is complementary to a long line of (partly symmetric) approaches to add *syntactic* extensibility to programming languages [BV04, Lea66, WC93, CMA94, BLS98, BG04a]. Most of these approaches support *syntactic safety*. Embedded code is checked at compile time to be syntactically correct. In [BV04], embedded code is used to compose XML documents. Proper program transformations and a properly defined underlying XML API guarantee that compositions (i.e. generated XML instance documents) are syntactically correct as well. However, nothing can be said about the validity of generated XML documents with respect to a given schema. This lack of type safety is inherent to most of the referenced approaches. In [BLS98], argument and result types of embedded code are checked but there is no error trailing (i.e. type checking errors in the expanded code are not traced back to the unexpanded syntax).

In contrast to syntactic language extensions, the Zhi# approach makes external type definitions amenable to full-fledged type checking in the context of their use within the host language. External types can be addressed following an object-oriented notation. In particular, external types are assumed to be organizable into taxonomies. Such a hierarchical organization provides for the reuse of methods and data that are located higher in the hierarchy. Objects are either atomic value types, whose value spaces may be constrained by means of constraining facets, or complex types, which can be seen as collections of named attributes. The author trusts that these two notations plus operator overloading plus some minor syntactical extensions for importing external namespaces and referencing external types are sufficient for a variety of applications. In fact, Bravenboer and Visser state that in object-oriented languages “language constructs are often sufficient for domain abstractions at the semantic level” [BV04].

## 2.6 Summary

The author devised a compiler framework that makes the C# programming language extensible with respect to external typing and subtyping mechanisms (e.g., type inference, subsumption, type derivation). External type definitions can be used in Zhi# programs in all places where .NET types are admissible except for type declarations. The resulting Zhi# programming language is a proper superset of ECMA 334 standard C# version 1.0. Three minor syntactical extensions are entailed by Zhi#'s support for external type systems. Firstly, external types can be included using the keyword *import*, which works analogously for external types like the C# *using* directive for .NET types. Secondly, in Zhi# program code external type references must be fully qualified using an alias that is bound (via an *import* directive) to the containing external namespace. Thirdly, the Zhi# language grammar includes additional binary comparison operators to facilitate the inference of constrained XML data types.

Zhi# programs are compiled into conventional C# code, which is correct by generation. The Zhi# compiler framework incorporates a complete source-to-source C# compiler including a Code DOM, which comprises 74 node types to represent elements of a Zhi# program. The compiler framework provides the core functionality to type check and transform conventional C# programs. The architectural model of the Zhi# compiler framework was designed for easy extensibility. Most of the program analysis tasks that require knowledge about the possibly complex code structure of C# programs were factored out of the framework extension points. Compiler plug-ins must implement extension points for (sub-)typing and program transformation. Both extension points are for the most part context free (i.e. the analysis and transformation of external expressions does not depend on the position of the expressions in the program text). Plug-ins for the Zhi# compiler framework can be developed without exhaustive knowledge about the Zhi# language grammar. In particular, only 14 methods that perform code transformations on the Zhi# Code DOM are necessary in order to cope well with external type systems as



different as constrained XML data types (see Appendix B.1) and complex OWL concept descriptions (see Appendix B.2). At the same time, no a priori knowledge about the supported external type systems of prospective compiler plug-ins is required in the compiler framework. In contrast to naïve approaches that introduce a proxy class for every external type definition or that are based on additional code generation, the program overhead of compiled Zhi# programs is constant and does not grow with, for example, the number of referenced XML data types or OWL concept descriptions.

The Zhi# compiler facilitates the cooperative usage of type definitions from different type systems. For example, XML data types can be used along with OWL datatype properties. Type checking and program transformation are accomplished by those compiler plug-ins that are responsible for the particular external expression parts. The compiler framework selects the compiler plug-ins based on the classification of types into type systems. Internally, types are classified into type systems based on their type system evidences that are initially assigned to type names by *import* directives. All external type system functionality is used in a “pay as you go manner”. There is no performance or code size overhead for Zhi# programs that do not use external type definitions.

The presented approach is not limited to the C# programming language but can equally be applied to any statically typed object-oriented programming language (e.g., Java). Also, external type systems are not limited to XML Schema Definition’s value space-based subtyping or ontological reasoning with the Web Ontology Language.

The current implementation of the Zhi# compiler is hosted by a .NET Windows Forms application (see Fig. A.1). This host application provides an XML-over-TCP interface, which is used by the Zhi# Eclipse frontend (see Fig. A.2). The Eclipse-based frontend boasts a Zhi# editor with syntax highlighting and autocompletion of XML data types, OWL concept descriptions, and ontological roles. An MSBuild task component was implemented that makes the Zhi# compiler available for the build system of Microsoft’s `msbuild.exe` tool and Visual Studio.



## CHAPTER 3

### The $\lambda_C$ -Calculus

This chapter elucidates an extension of the simply typed lambda calculus with subtyping ( $\lambda_{<}$ ) for constrained atomic (i.e. simple) data types. In this work, the term “constrained types” refers to atomic data types that represent a value space, which may be constrained by explicitly defined constraining facets (e.g., *xsd:minExclusive*, *xsd:maxExclusive*). This is different to constraint-based type inference algorithms found in the literature where constraints are not checked but rather recorded for later consideration.

Atomic data types can only have atomic values, which are not allowed to be further fractionalized even though this may be technically possible (e.g., the XML data type *xsd:string* is considered an atomic data type despite the fact that it comprises several distinguishable characters). The subtyping and type derivation features of the  $\lambda_C$ -calculus were developed in order to cover constraint-based type derivations as they are allowed for atomic XML Schema Definition data types. The notations that are used in this chapter are listed in Table 3.1. See Appendix C for a complete definition of the  $\lambda_C$ -calculus.

Analogously to the W3C Recommendation for XML data types [BM04b] we begin with the introduction of a number of unconstrained primitive data types  $P_1, \dots, P_n \in \mathcal{P}$ . In particular, built-in XML Schema Definition data types such as *xsd#duration*, *xsd#dateTime*, and *xsd#decimal* are valid elements of  $\mathcal{P}$  and will be denoted  $P_{\text{xsd\#duration}}$ ,  $P_{\text{xsd\#dateTime}}$ , and  $P_{\text{xsd\#decimal}}$ , respectively.

A primitive data type  $P$  is a three-tuple consisting of a set of distinct values denoted  $v(P)$ , called its *value space* (e.g., the value space  $v(P_{\text{xsd\#boolean}})$  is the set  $\{\text{true}, \text{false}\}$  to denote a logical true and a logical false), a set of lexical representations called its *lexical*

Table 3.1: Terminology of the  $\lambda_C$ -calculus

$P_1, \dots, P_n \in \mathcal{P}$	primitive base types (e.g., $P_{\text{xsd}\#\text{boolean}}$ , $P_{\text{xsd}\#\text{string}}$ )
$v(T)$	value space of type $T$
$c = \phi(TV)b$	constraint $c$ with base type parameter $TV$ and body $b$
$b = \{x x \in v(TV)\} \bigcap_{k \in 1..m} \{x x \prec \text{literal}_k\}$	constraint body $b$ with base type parameter $TV$
$T.c$	constraint application
$\bigcap_{i \in 1..n}^T c_i \equiv T.c_1 \dots c_n$	multiple constraint application
$c\{\{TV \leftarrow T\}\}$	type variable binding in constraint $c$
$v(c\{\{TV \leftarrow T\}\})$	value space of bound constraint $c$
$<:$	subtype of
$<::$	subconstraint of

space, and a set of *fundamental facets* that characterize properties of the value space (e.g., cardinalities, order relations). Each value in the value space  $v(P)$  of a data type  $P$  may be denoted by more than one literal of its lexical space (e.g., “1” and “1.0” are both valid lexical representations of the same  $P_{\text{xsd}\#\text{float}}$  value 1).

## 3.1 Facets

### 3.1.1 Fundamental Facets

A *fundamental facet* is an abstract property that semantically characterizes the values in a value space. The  $\lambda_C$ -calculus includes the fundamental facets *equality*, *order*, *boundedness*, *cardinality*, and *numeric* as provided by the W3C Recommendation for XML data types [BM04b]. The fundamental facet *date-time* was added to indicate dates and times.

Every value space supports the notion of *equality*, with the following rules. For any  $a$  and  $b$  in the value space, either  $a$  is equal to  $b$ , denoted  $a = b$ , or  $a$  is not equal to  $b$ , denoted  $a \neq b$ . There is no pair  $a$  and  $b$  from the value space such that both  $a = b$  and  $a \neq b$ . For all  $a$  in the value space,  $a = a$ . For any  $a$  and  $b$  in the value space,  $a = b$  if and only if  $b = a$ .

For any  $a$ ,  $b$ , and  $c$  in the value space, if  $a = b$  and  $b = c$ , then  $a = c$ . For any  $a$  and  $b$  in the value space, if  $a = b$ , then  $a$  and  $b$  cannot be distinguished (i.e. equality is identity). In spite of the fact that some value spaces of built-in XML Schema Definition data types such as  $P_{\text{xsd}\#\text{float}}$  and  $P_{\text{xsd}\#\text{double}}$  may intuitively appear to be compatible, they will be treated as pairwise disjoint for primitive data types  $P_i, P_j \in \mathcal{P}$  where  $i \neq j$  and  $i, j \in 1..n$  (i.e.  $P_i$  and  $P_j$  do not share any values).

Value spaces may either be *ordered* or *partially ordered*. A value space, and hence a data type, is said to be ordered if there exists an order-relation defined for that value space. Data types where the order relation is irreflexive, asymmetric, and transitive are said to be partially ordered. Data types whose value spaces have upper and lower bounds defined are said to be *bounded*; *unbounded* if no such bounds exists. Value spaces may be finite or countably infinite. A data type is said to have the *cardinality* of its value space. A data type is said to be *numeric* if its values are conceptually quantities in some mathematical number system. A data type is said to be a *date-time* data type if its values denote dates and times as described in ISO 8601 [Int04].

### 3.1.2 Constraining Facets

A *constraining facet* is an optional property that can be applied to a data type to constrain its value space. In the  $\lambda_C$ -calculus, constrained atomic types are computational structures. The only operations on atomic types are constraint applications. Table 3.1 summarizes the notations that are used for atomic types and constraints.

In more detail, a value space  $v(T)$  is the set of values for a given data type  $T$ . The value spaces of primitive XML Schema Definition data types  $\mathcal{P}_{\text{xsd}}$  such as  $P_{\text{xsd}\#\text{duration}}$ ,  $P_{\text{xsd}\#\text{dateTime}}$ , and  $P_{\text{xsd}\#\text{decimal}}$  are given by the respective type definitions of XML Schema Definition built-in data types [BM04b].

The value space of a base type  $T$  can be restricted by the application of one or more constraints  $c_i = \phi(TV)b_i$   $i \in 1..n$ . Each constraint  $c_i$  has a type variable  $TV$ , which is to be

bound to a base type  $T$ , and a body  $b$  that possibly constrains the value space  $v(T)$ . The letter  $\phi$  is used as a binder for the base type parameter  $TV$ . A constraint  $c = \phi(TV)b$  where the type variable  $TV$  is bound to a base type  $T$  (denoted by  $c\{\{TV \leftarrow T\}\}$ ) has a value space  $v(c\{\{TV \leftarrow T\}\})$  comprising all elements of the value space of the base type  $T$  that satisfy the comparison operations defined in the constraint body  $b$ . A constraint body  $b = \{x|x \in v(TV)\} \cap_{k \in 1..m} \{x|x \prec literal_k\}$  defines the intersection of the value space of  $TV$  and those values that satisfy the properties  $x \prec literal_k$   $k \in 1..m$ . Depending on the constraining facet ' $\prec$ ',  $literal_k$  is interpreted as a lexical representation of an element of  $v(TV)$  or of another value space that is implicitly effective for the constraining facet according to, for example, the W3C Recommendation for XML Schema Definition (e.g., if  $TV$  is bound to the XML data type  $P_{xsd\#gYear}$ ,  $literal_k$  is interpreted as a lexical representation of a Gregorian calendar year while it is taken as an integer number for the constraining facets  $c_{xsd:length}$  and  $c_{xsd:totalDigits}$ , which define the length of string data types and the maximum number of digits of integer data types, respectively).

Table 3.2: Comparison operators for XSD

Zhi# comparison operator	XSD constraining facet
?=	$c_{xsd:length} (c_{?=})$
?>	$c_{xsd:minLength} (c_{?>})$
?<	$c_{xsd:maxLength} (c_{?<})$
??	$c_{xsd:pattern} (c_{??})$
\$=	$c_{xsd:enumeration} (c_{\$=})$
<=	$c_{xsd:maxInclusive} (c_{\leq})$
<	$c_{xsd:maxExclusive} (c_{<})$
>	$c_{xsd:minExclusive} (c_{>})$
>=	$c_{xsd:minInclusive} (c_{\geq})$
%%	$c_{xsd:totalDigits} (c_{\%\%})$
%.	$c_{xsd:fractionDigits} (c_{\%.})$

In order to capture the constraining facets as defined in the W3C Recommendation for XML Schema Definition [BM04b] the comparison operators given in Table 3.2 can be substituted for the operator placeholder ' $\prec$ ' in constraint bodies. The given operators were also included in the Zhi# language specification in order to cope particularly well with constrained XML data types (see Chapter 5). Particular constraints where both the comparison operator and the (single) literal of the constraint body are given are denoted by a  $c$  subscripted with the operator symbol or the name of the constraining facet followed by the literal (e.g.,  $c_{[\prec 5]}$  and  $c_{[\text{xsd:maxExclusive } 5]}$  denote the exclusive upper bound 'five').

A constraint  $c_1 = \phi(TV_1)b_1$  is a sub-constraint of a constraint  $c_2 = \phi(TV_2)b_2$  if the base type parameters  $TV_1$  and  $TV_2$  are bound to the same base type  $T$  and the value space  $v(c_1\{\{TV_1 \leftarrow T\}\})$  is subsumed by  $v(c_2\{\{TV_2 \leftarrow T\}\})$  as shown in Table 3.3.

Table 3.3: S-CSTRVSPACE

$$\frac{v(c_1\{\{TV \leftarrow T\}\}) \subseteq v(c_2\{\{TV \leftarrow T\}\})}{c_1 \prec:: c_2}$$

Rule S-CSTRVSPACE subsumes the width and depth subconstraint rules S-CSTRWIDTH and S-CSTRDEPTH. The *width subconstraint* rule S-CSTRWIDTH as shown in Table 3.4 captures the intuition that one wants to consider the constraint  $c_1 = \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n+k} \{x|x \prec literal_i\}$  to be a subconstraint of the less restrictive rule  $c_2 = \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec literal_i\}$ .

Table 3.4: S-CSTRWIDTH

$$\phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n+k} \{x|x \prec y_i\} \prec:: \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec y_i\}$$

Rule S-CSTRWIDTH applies only to constraints where the common properties  $x \prec literal_i$   $i \in 1..n$  are identical and fewer properties are defined in the body of constraint  $c_2$  than in the body of  $c_1$ . It is also safe to allow the comparison operands  $literal_i$   $i \in 1..n$  to vary as long as the value spaces of each corresponding comparison operation are in

the subset relation. The *depth subconstraint* rule S-CSTRDEPTH as shown in Table 3.5 captures this notion.

Table 3.5: S-CSTRDEPTH

$$\frac{\text{for each } i \quad \{x|x \prec y_i\} \subseteq \{x|x \prec z_i\}}{\phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec y_i\} <:: \phi(TV)\{x|x \in v(TV)\} \bigcap_{i \in 1..n} \{x|x \prec z_i\}}$$

As shown below, a more restrictive constraint  $\text{succ}(c)$  can be computed by replacing the literal  $y$  in the body of constraint  $c$  by its successor  $\text{succ}(y)$  or predecessor  $\text{pred}(y)$  depending on the used comparison operator and bound base type such that, the resulting value space of the constrained type becomes more specific. Note that here the variable  $y$  stands for literal values while the variable  $c$  denotes a constraint (i.e. the definition of  $\text{succ}(c)$  is not circular).

$$c = \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \prec y\}$$

$$c_{\text{pred}} \equiv \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \prec \text{pred}(y)\}$$

$$c_{\text{succ}} \equiv \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \prec \text{succ}(y)\}$$

$$\text{succ}(c) = \begin{cases} c_{\text{pred}} & \text{iff } v(c_{\text{pred}}\{\{TV \leftarrow T\}\}) \subset v(c\{\{TV \leftarrow T\}\}) \\ c_{\text{succ}} & \text{iff } v(c_{\text{succ}}\{\{TV \leftarrow T\}\}) \subset v(c\{\{TV \leftarrow T\}\}) \end{cases}$$

The successor  $\text{succ}(c)$  of a constraint  $c$  is undefined if there are no constraints  $c_{\text{pred}}$  and  $c_{\text{succ}}$  whose values spaces – when applied on a base type  $T$  – are subsumed by  $v(c\{\{TV \leftarrow T\}\})$ . In this case, it is not possible to derive further subtypes using constraint  $c$ . As a result, the constraint’s host type to which the base type parameter  $TV$  is bound is *final over constraint*  $c$  (e.g., a constraint that defines the length of a string data type cannot be used to derive subtypes since strings must be of exactly the given length).



## 3.2 Type Derivation

In the  $\lambda_C$ -calculus, atomic types are inductively defined by their value spaces. Atomic types are derived through the application of value space constraints according to rule TD-CSTRAPP.

Table 3.6: TD-CSTRAPP

$$\frac{T' = T.c}{v(T') = v(c\{\{TV \leftarrow T\}\})}$$

Constraint applications on atomic types can be reduced to set operations on their value spaces. Let  $T_n \equiv \bigcap_{i \in 1..n}^{T_0} c_i$  be an atomic type, which shall be derived from a given base type  $T_0$  based on a number of constraints  $c_1, \dots, c_n$ . Using rule TD-CSTRAPP the effective value space  $v(T_n)$  of the derived type  $T_n$  can be computed by reducing the sequence of constraint applications  $T_0.c_1 \dots c_n$  to the successive application of constraints  $c_k$  on type  $T_{k-1}$  for  $k = 1, \dots, n$  such that,  $T_k = T_{k-1}.c_k$ . Accordingly,  $v(T_k) = v(c_k\{\{TV \leftarrow T_{k-1}\}\})$ .

A value space expression  $v(c_k\{\{TV \leftarrow T_{k-2}.c_{k-1}\}\})$  may be rewritten as  $v(c_k\{\{TV \leftarrow T_{k-2}\}\}) \cap v(c_{k-1}\{\{TV \leftarrow T_{k-2}\}\})$  as long as type definitions within a derivation chain are not subject to change. If in a derivation tree a type definition  $T$  is substituted by a more specific type definition  $T'$  (denoted by  $T \rightarrow T'$ ), all dependent type definitions change accordingly as defined by rule TD-SUBS (note that  $T$  and  $T'$  must be derived from the same primitive base type). For example, if in a derivation tree  $T_3 <: T_2 <: T_1$ , where  $T_2 \equiv T_1.c_{11}$  and  $T_3 \equiv T_2.c_2$ ,  $T_2 \rightarrow T_2.c_{12}$ , then  $T_3 \rightarrow T_3.c_{12}$ .

Table 3.7: TD-SUBS

$$\frac{T \rightarrow T'}{T.c \rightarrow T'.c}$$

Using constraint-based type derivation, an atomic data type *age* as defined in the XSD listing shown below can be derived from the XML data type  $A_{\text{xsd}\#\text{int}}$  through the

application of the two constraints  $c_{[\geq 0]} = \phi(TV)\{x|x \in v(TV)\} \cap \{x|x \geq 0\}$  and  $c_{[< 110]} = \phi(TV)\{x|x \in v(TV)\} \cap \{x|x < 110\}$  as follows:  $age = A_{xsd\#int} \cdot c_{[\geq 0]} \cdot c_{[< 110]}$ .

```

1 <xsd:schema xmlns:xsd="...">
2   <xsd:simpleType name="age">
3     <xsd:restriction base="xsd:int">
4       <xsd:minInclusive value="0"/>
5       <xsd:maxExclusive value="110"/>
6     </xsd:restriction>
7   </xsd:simpleType>
8 </xsd:schema>

```

The value space of the derived data type *age* comprises all elements of  $v(A_{xsd\#int})$ , which are greater than or equal to zero and less than 110.

The effective value space of a derived data type can be resolved by expanding all constraints that were applied to the primitive base type. The constrained type *age* is effectively derived from the XSD built-in primitive type  $P_{xsd\#decimal}$ , whose axiomatically defined value space is the subset of the real numbers that can be represented by decimal numerals, through the application of the following constraints:  $age = P_{xsd\#decimal} \cdot c_{[ \% . 0 ]} \cdot c_{[\geq -9223372036854775808]} \cdot c_{[\leq 9223372036854775807]} \cdot c_{[\geq -2147483648]} \cdot c_{[\leq 2147483647]} \cdot c_{[\geq 0]} \cdot c_{[< 110]}$

The above type definition is given in *normal form* (i.e. all constraints that are used for the type derivation are explicitly applied on a primitive base type with a predefined value space). Using the rules S-CSTRVSPACE, S-CSTRWIDTH, and S-CSTRDEPTH a normal form can be reduced to a *minimal normal form* where each constraining facet such as, for example,  $c_{\leq}$  occurs only once. The definition of the derived type *age* can be reduced to the following minimal normal form:  $age = P_{xsd\#decimal} \cdot c_{[ \% . 0 ]} \cdot c_{[\geq 0]} \cdot c_{[\leq 2147483647]} \cdot c_{[< 110]}$

Even though the constraint  $c_{[< 110]}$  is intuitively more specific than  $c_{[\leq 2147483647]}$  for the domain of the primitive base type  $P_{xsd\#decimal}$  there is no universally valid rule to merge two different constraining facets. Still, implementations of the  $\lambda_C$ -calculus as in

the XSD plug-in for the Zhi# compiler framework may treat constraining facets such as *xsd:maxInclusive* and *xsd:maxExclusive* as being compatible (i.e. reducible) when applied on the same primitive base type.

Also, for the sake of efficiency, our implementation of the  $\lambda_C$ -calculus reduces all type definitions to their minimal normal forms. Thus, for a given number of different constraining facets, the evaluation of the subtyping rules as described in the next section takes the same number of steps independent of the type derivation depth.

### 3.3 Subtyping

A main characteristic of object-oriented languages is that an object can emulate another object that has fewer methods, since the former supports the entire protocol of the latter. Analogously, for constrained data types as defined in this work, the basic rules of subtyping are effective: reflexivity, transitivity, and subsumption. Additionally, width and depth subtyping rules apply to atomic data types that are constrained by one or more constraining facets.

A type  $S$  is considered to be a subtype of  $T$  (denoted by  $S <: T$ ) if the value space of  $S$  is a subset of the value space of  $T$  as shown in Table 3.8. In particular,  $S$  and  $T$  must be derived from the same primitive base type since otherwise their value spaces would be disjoint. The rule S-VSPACE subsumes the width and depth subtyping rules S-WIDTH and S-DEPTH.

A type  $S$  is considered to be a subtype of  $U$  if both types are derived from the same base type  $T$  through the application of constraints and fewer constraints are defined for type  $U$  than for  $S$ . The intuition that it is safe to add constraints to an atomic type is captured by the *width subtyping* rule S-WIDTH for constrained atomic types as shown in Table 3.9 (the notation  $\bigcap_{i \in 1..n}^T c_i$  is a shorthand form for  $T.c_1 \dots c_n$ , i.e. the subsequent application of constraints  $c_i$   $_{i \in 1..n}$  on base type  $T$ ).

Constraints that are defined for atomic types may vary as long as the value spaces of each corresponding constraint are in the subset relation (i.e. the constraints are in the sub-constraint relation denoted by  $c <:: d$ ). The *depth subtyping* rule S-DEPTH for constrained atomic types as shown in Table 3.10 expresses this notion.

Finally, the subtyping rule S-APP captures the notion that a constraint application always makes a type more specific (i.e. constraint applications can only reduce the value space of a type), which is a required property for the soundness of the  $\lambda_C$ -type system.

Table 3.8: S-VSPACE

$$\frac{v(S) \subseteq v(T)}{S <: T}$$

Table 3.9: S-WIDTH

$$\frac{S = \bigcap_{i \in 1..n+k}^T c_i \quad U = \bigcap_{i \in 1..n}^T c_i}{S <: U}$$

Table 3.10: S-DEPTH

$$\frac{\text{for each } i \quad c_i <:: d_i \quad S = \bigcap_{i \in 1..n}^T c_i \quad U = \bigcap_{i \in 1..n}^T d_i}{S <: U}$$

In the  $\lambda_C$ -calculus, the subtyping rule for function types is the standard one with covariant return types and contravariant function parameters (i.e. a function type is contravariant in its domain and covariant in its range). For constrained atomic data types that are embedded with a conventional object-oriented programming language, the subtyping rule for function types could follow the same scheme. However, while covariant return types and contravariant formal parameter types are generally safe, different programming languages support different variance policies. For example, return type covariance is implemented in the Java programming language version 1.5 while it is not in version 1.0

Table 3.11: S-APP

$$\frac{S <: T}{S.c <: T}$$

of the C# programming language where both return and parameter types have to be invariant. While the calculus presented in this work uses a specific implementation of the subtyping rule for function types, different off-the-shelf behaviors of host programming languages may be adopted. Unlike the current version of C#, in the Zhi# programming language, constrained atomic data types as described in this section can be used as co-variant return types and contravariant formal parameter types. Also, it is possible to use source objects of more specific (i.e. more constrained) types at any time (i.e. read-only substitutability, see [ZW97]).

### 3.4 Properties of the $\lambda_C$ -Type System

The most fundamental property of type systems is *safety* (also called *soundness*). The soundness of a type system can be shown by proving the *progress* and *preservation* theorems, which say that a well-typed term is either a value or it can take a step of evaluation and if a well-typed term takes a step of evaluation then the resulting term is also well-typed, respectively. In [Pie02], proofs of the *progress* and *preservation* theorems for the simply typed lambda-calculus with subtyping ( $\lambda_{<}$ ) are given. Both proofs are based on straightforward induction on a derivation of  $t : T$ . Both properties are preserved in the  $\lambda_C$ -calculus because the type construction mechanisms do not pertain to the original  $\lambda_{<}$ -typing and evaluation rules. What remains is to prove the subsumption property for constrained types.

Intuitively,  $A <: B$  can be read “every value described by A is also described by B”. According to rule S-TRANS, type derivations must only yield types whose value spaces are more specific than those of their base types. The proof of the following theorem will show

that this subsumption property is always satisfied for type derivations in the  $\lambda_C$ -calculus.

**Theorem 3.1 (Subsumption)** *If  $A' <: A$  and  $A <: B$  then  $A' <: B$ .*

*Proof:* There is almost nothing to show since in the  $\lambda_C$ -calculus atomic types can only be derived through constraint applications.

*Case (TD-CSTRAPP):*  $A' = A.c$ ,  $c = \phi(TV)\{x|x \in v(TV)\} \cap \{\dots\}$

For a type derivation  $A' = A.c$ , where  $c = \phi(TV)\{x|x \in v(TV)\} \cap \{\dots\}$  is an arbitrary constraint, the value space  $v(A') = v(c\{\{TV \leftarrow A\}\})$  reduces to  $\{x|x \in v(A)\} \cap \{\dots\}$ . Consequently, all elements of  $v(A')$  are also elements of  $v(A)$  and the rule S-VSPACE can be applied to conclude that  $A' <: B$ .  $\square$

### 3.5 Type Inference

Section 3.2 described how atomic types can be derived through explicit applications of value space constraints. This form of type construction mimics the semantics of XML Schema Definition. Using the  $\lambda_C$ -type system it is also possible to infer transient constraints that hold for the instances of constrained types within only a limited scope of a program. A scope within a program shall be denoted by  $\diamond$ , a sub-scope of  $\diamond$  shall be denoted by  $\diamond\diamond$ . Considering the *then*-branch of an *if*-statement of the form *if* ( $a \prec literal$ ) *then*  $\diamond$  it is safe to add the constraint  $c_{[\prec literal]}$  to the type of variable  $a$ . The constraint  $c_{[\prec literal]}$  holds for the instance  $a$  within scope  $\diamond$  until  $a$  is assigned a value (i.e. in a programming language with side effects  $a$  must also not be referenced by method invocations). These two intuitions of adding constraints to the type of a variable for a limited scope and eventually removing them upon an assignment to the variable are captured by the rules TI-IFADD and TI-ASSIGNREM, where  $\Gamma$  and  $\Gamma_\diamond$  are a typing context and a transient typing context for a limited scope  $\diamond$ , respectively (i.e.  $\Gamma_{\diamond\diamond}$  is the typing context of the sub-scope  $\diamond\diamond$ ).

Table 3.12: TI-IFADD

$$\frac{\Gamma \vdash a : A \quad \text{if} \left( \bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i) \right) \text{ then } \diamond}{\Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x \mid x \in v(A)\} \cap \{x \mid x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

Table 3.13: TI-ASSIGNREM

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : A \quad \Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad a := t}{\Gamma_{\infty} \vdash a : A}$$

The following example illustrates the effect of the TI-IFADD rule on the type checking of  $\lambda_C$ -programs. Under the assumption  $\vdash a : A_{\text{xsd}\#\text{int}}$  the following  $\lambda_C$ -application, where the value space of type *age* includes integer numbers from 0 to 110, will fail to type-check according to rule S-VSPACE since  $v(A_{\text{xsd}\#\text{int}}) \not\subseteq v(\text{age})$ . The application does, however, type-check if it occurs within the *then*-branch of an *if*-statement as shown below.

```
(λ x : age . x)(a)
```

```
if ((a >= 0) && (a < 110)) then (λ x : age . x)(a) endif
```

For the scope of the *then*-branch the type of *a* can be inferred to be  $A_{\text{xsd}\#\text{int}} \cdot c_{[\geq 0]} \cdot c_{[< 110]}$ . Using the rule S-VSPACE one can conclude that  $A_{\text{xsd}\#\text{int}} \cdot c_{[\geq 0]} \cdot c_{[< 110]}$  is a subtype of *age*.

In the implementation of the  $\lambda_C$ -calculus in the XSD plug-in for the Zhi# compiler framework only ANDed top-level expressions are used for type inference. In this way, the type inference algorithm is rather conservative (i.e. incomplete), which proved to be sufficient in practice since the effects of type inference should still be comprehensible for the programmer. For fundamental reasons, the proposed kind of type inference will always be incomplete since otherwise the inference algorithm would solve the Halting Problem (for the same reason, there is no “optimal” compiler).

## 3.6 Implementation

The  $\lambda_C$ -type system as described in this chapter was fully implemented in C# and Java for the XML Schema Definition type system. Fig. 3.1 depicts the architecture of the  $\lambda_C$ -type system implementation on a class level. A *TypePool* aggregates a number of *TypeSystem* implementations such as the *XSDTypeSystem*, which contains the logic to load and parse XML Schema Definition files and to create *CTSimpleType* objects. Each loaded XSD type is represented by one *CTSimpleType* instance, which is aggregated by the singleton class *TypePool*. There are 19 implementations of the *CTSimpleType* interface in order to represent the 19 built-in primitive base types of XML Schema Definition (for the sake of conciseness, Fig. 3.1 comprises only *CTAnyURI* and *CTTime*). Note that the devised architecture makes it possible to use not only types of one particular type system but to join an arbitrary number of *TypeSystem* and *CTSimpleType* implementations in one single *TypePool* in order to support the semantics of several different type systems (e.g., atomic SQL data types such as *money* and *time*). Each *CTSimpleType* aggregates a *LexicalSpace* and a *ValueSpace*, which represent the lexical space and fundamental value space facets (e.g., *order*, *boundedness*, *cardinality*) of a particular type definition, respectively. Types can be derived from built-in primitive *CTSimpleTypes* by adding *ConstrainingFacets* to the *ValueSpace* of a *CTSimpleType*. In XML Schema Definition, there are 12 constraining facets to derive from the 19 built-in primitive types. Only the *Enumeration* and *TotalDigits* facets are shown in Fig. 3.1.

Constraining facets can be *modifying* (i.e. string literals that are assigned to instances of constrained types are implicitly modified upon assignment) or *enforcing* (i.e. certain value space constraints must hold for the interpretations of string literals that are assigned to instances of constrained types). The devised architecture makes it possible to implement arbitrary application specific constraints. For example, one may define modifying constraints “nonnull” or “translate” that automatically replace *null* values by empty strings and replace string values by localized translations, respectively.



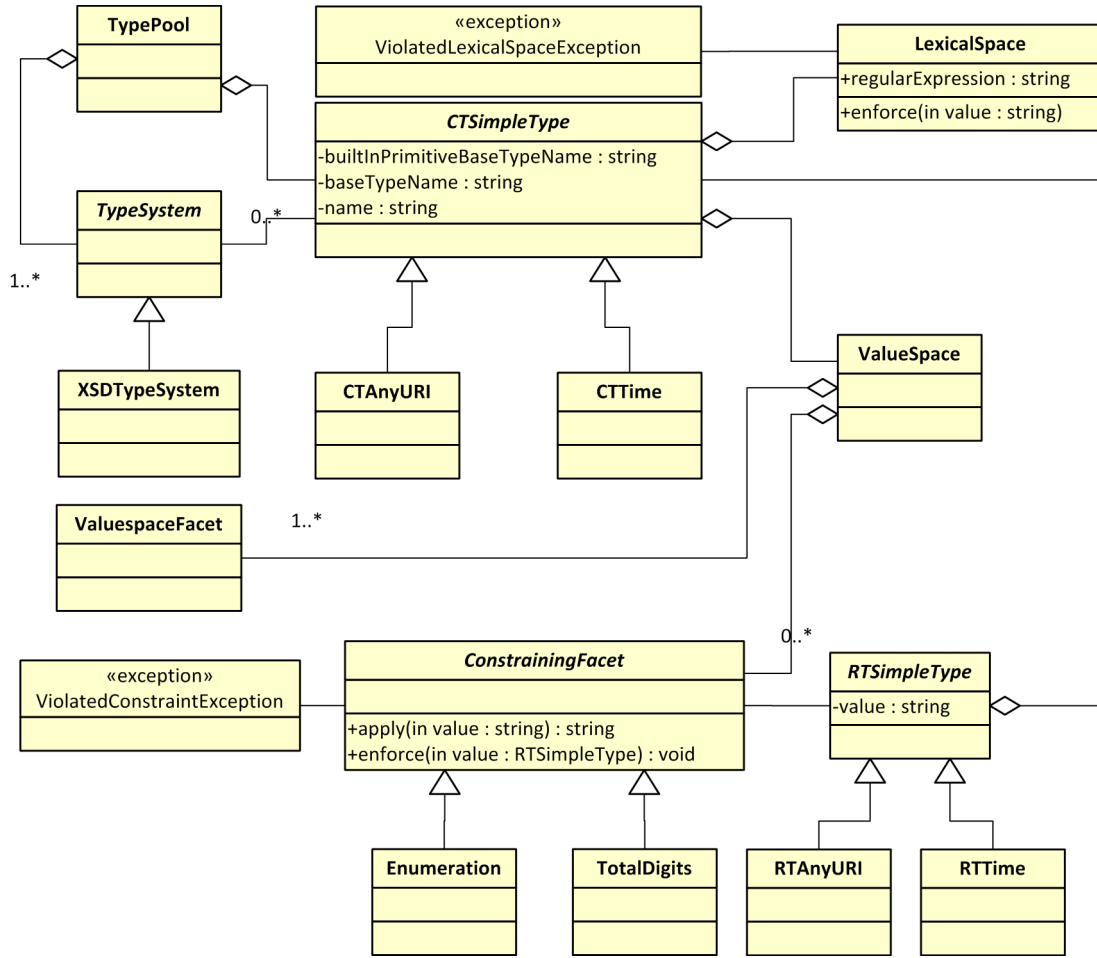


Figure 3.1: Architecture of the  $\lambda_C$ -type system implementation

*RTSimpleType* objects represent runtime occurrences of XML data types in, for example, Zhi# programs. Each implementation of *RTSimpleType* such as *RTAnyURI* or *RTTime* aggregates a corresponding *CTSimpleType* metaobject that carries the actual type information. At runtime, assignments to the *value* member of *RTSimpleType* objects are guarded by the lexical space that is defined for the primitive base type of the particular type (i.e. the *LexicalSpace* of the *CTSimpleType* metaobject) and by the constraining facets that are defined for the particular type (i.e. the *ConstrainingFacets* that are aggregated by the *ValueSpace* of the *CTSimpleType* metaobject). A *ViolatedLexicalSpaceException* and a *ViolatedConstraintException* is thrown if the enforcement of the lexical space and value space constraints fails, respectively.

Internal data type specific representations of the *RTSimpleType value* member are used in the XSD implementation of the  $\lambda_C$ -type system in order to facilitate comparisons and arithmetic operations involving XML data type values. For example, components of the *xsd#dateTime* data type such as year, month etc. are stored as separate integer values. Also, XML Schema Definition allows (countable) infinite value spaces for numeric data types. Big number arithmetic [Knu97] was used for the XSD implementation of the  $\lambda_C$ -type system in order to resolve value space limitations of .NET numeric data types. The Zhi# compiler can be configured to support numeric value spaces of almost arbitrary cardinalities.

Fig. 3.2 depicts an object diagram of a  $\lambda_C$ -type pool that contains the type definition of an XML data type named *chokyColaURI*, which is derived from the built-in primitive type *xs#anyURI* based on the enumeration of three valid URIs. Note how the lexical and value space definitions of the XSD code snippet are represented by objects in the diagram. Also, there is a named instance *aChokyColaURI* of the XSD data type *http://www.chokycola.com#chokyColaURI* represented by an *RTAnyURI* object.

The  $\lambda_C$ -type system is not only used for static typing and dynamic checking in the Zhi# programming language as described in Section 5.1; it has also been embedded with the CHIL Knowledge Base Server [PRS09], which is an adapter for OWL ontology management systems (see Section 4.1). The implementation of the  $\lambda_C$ -type system made it particularly easy to augment the OWL API of the CHIL Knowledge Base Server with type checking for OWL datatype properties, whose ranges are atomic XML Schema Definition data types. For experimental purposes a  $\lambda_C$ -interpreter, which can be used to load XSD type definitions and to evaluate  $\lambda_C$ -terms, was implemented as a .NET Windows Forms application. An ANTLR [Par05] grammar of the  $\lambda_C$ -syntax was developed in order to automatically generate a lexer and parser for  $\lambda_C$ -terms whose abstract syntax trees are eventually transformed into a  $\lambda_C$ -Code DOM in order to facilitate the semantic analysis and execution (i.e. reduction) of  $\lambda_C$ -applications.

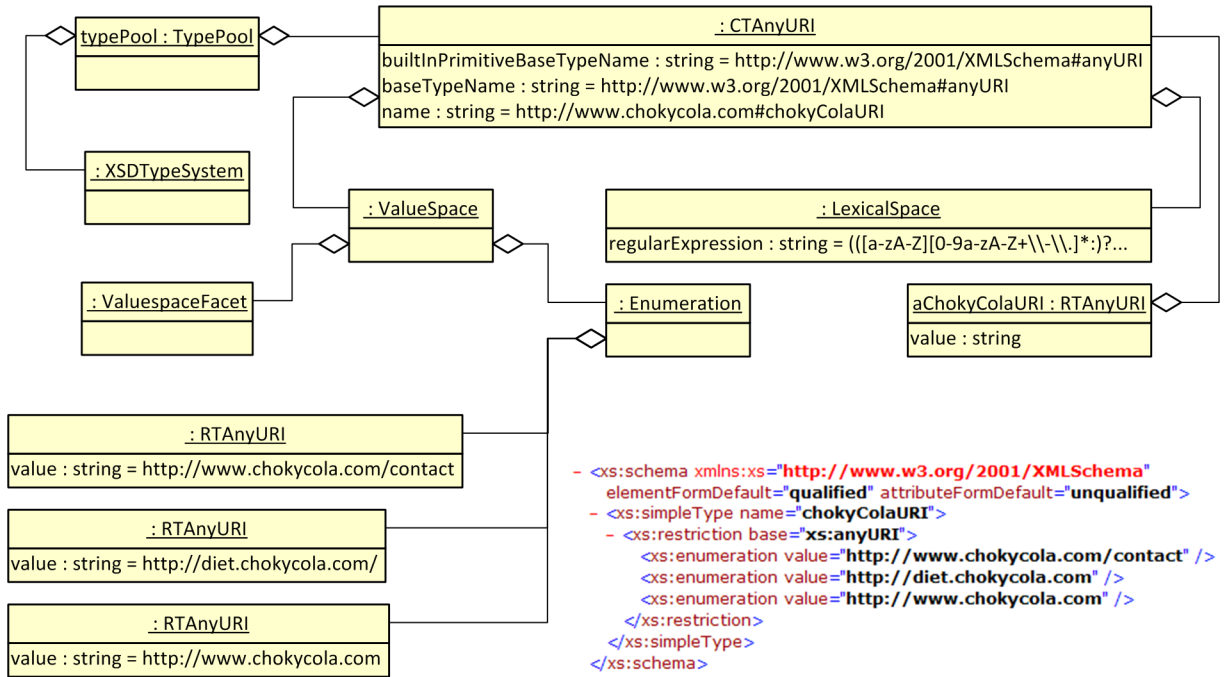


Figure 3.2:  $\lambda_C$ -type pool

### 3.7 Constraint Arithmetic

The evaluation and subtyping rules of the  $\lambda_C$ -calculus guarantee that in  $\lambda_C$ -abstractions typed bound variables do only accept instances of types whose value spaces are in the subset relation with the value space of the given type. In programming languages that embed the type system of the  $\lambda_C$ -calculus instances of constrained data types may not only be combined using  $\lambda_C$ -applications. Instead, typed terms may be combined in arbitrary binary arithmetic expressions.

Section 3.5 already indicated that the  $\lambda_C$ -calculus includes type inference features that allow the type of a variable to change within the scope of embracing *if*-statements. For the scope within an *if*-statement constraints are transiently added to the type of an object whose constrained value space is considered by subsequently applicable evaluation rules.

Similarly, the types of binary arithmetic expressions that include at least one instance of a constrained data type are inferred based on *constraint arithmetic*. Here, the objective

is to preserve as many constraints of the two operands as possible in order to compute the most specific constrained type of the binary expression. Let

$$\begin{aligned} A_{[< 50]} &= A_{\text{xsd}\#\text{positiveInteger}} \cdot C_{[< 50]}, \\ A_{[< 100]} &= A_{\text{xsd}\#\text{positiveInteger}} \cdot C_{[< 100]}, \text{ and} \\ A_{[< 200]} &= A_{\text{xsd}\#\text{positiveInteger}} \cdot C_{[< 200]}, \end{aligned}$$

be XML Schema Definition type definitions to represent positive integer numbers less than 50, 100, and 200, respectively. Also, let  $\Gamma$  be a typing context where

$$t_1 : A_{[< 50]}, t_2 : A_{[< 100]}, \text{ and } t_3 : A_{[< 200]}.$$

Using a naïve constraint arithmetic that simply discards all constraints that may be defined for the operands' types, the type of the binary expression  $t_1 + t_2$  would simply be inferred to be the most specific unconstrained common supertype. For XML Schema Definition data types, this strategy always yields an unconstrained primitive base type. In this case,  $\Gamma \vdash (t_1 + t_2) : P_{\text{xsd}\#\text{decimal}}$ . Consequently, the assignment  $t_3 = t_1 + t_2$  would fail to type-check since  $P_{\text{xsd}\#\text{decimal}}$  is not a subtype of  $A_{[< 200]}$ .

The architectural model of the  $\lambda_C$ -type system as depicted in Fig. 3.1 facilitates the implementation of a constraint arithmetic that follows general interval arithmetic rules. Fig. 3.3 shows the algorithm – given in pseudo code – that is executed by the XSD type checker of the Zhi# programming language in order to compute the type of the arithmetic expression  $t = t_1 + t_2$  where  $t : T$ ,  $t_1 : T_1$ , and  $t_2 : T_2$ . According to the class model depicted in Fig. 3.1 XML data types are represented by instances of the *CTSimpleType* class.

First, the types  $T_1$  and  $T_2$  are checked to be compatible (line 2). Following the stipulation of the  $\lambda_C$ -calculus XSD types are considered compatible only if they are derived from the same primitive base type. Still, implementations of the  $\lambda_C$ -type system may add functionality to relate instances of types with disjoint values spaces (e.g., XSD and .NET integers). Table E.1 lists how XML data types can be related in arithmetic operations. Zhi#'s XSD compiler plug-in adds functionality to relate XSD and .NET data

types. Next, an anonymous unconstrained clone is created of the first operand type (line 3). Note that all *Clone()* methods in the implementation of the  $\lambda_C$ -calculus return deep copies. In the cloned type object, the type name is set to the first operand's primitive base type name with all constraints removed from its value space. In the materialized type object clones *T1'* and *T2'* (lines 4 and 5) the set of defined constraining facets is materialized (lines 11–19) such that, redundant constraints are removed from the value space (e.g.,  $c_{[< a]} \wedge c_{[< b]} \wedge a < b \Rightarrow c_{[< a]}$ ), related constraints are homogenized (e.g.,  $c_{[< a]} \wedge c_{[\leq b]} \wedge a < b \Rightarrow c_{[< a]} \wedge c_{[\leq a]}$ ), and resulting constraints are inferred (e.g.,  $c_{[< a]} \wedge \overline{c_{[\leq x]}} \Rightarrow c_{[< a]} \wedge c_{[\leq a]}$ ). Implementations of the  $\lambda_C$ -type system may refine the constraint materialization rules for particular constraints that are applied on particular primitive base types. For example, for the numeric XSD data type  $P_{\text{xsd}\#\text{decimal}}$  the maximum number of fraction digits is taken into account for the homogenization of the *xsd:maxExclusive* and *xsd:maxInclusive* constraints (i.e.  $c_{[< a]} \wedge c_{[\leq b]} \wedge c_{[\% . 0]} \wedge a < b \Rightarrow c_{[< a]} \wedge c_{[\leq (a-1)]}$ ). Table E.2 lists the materialization rules that are implemented by the XSD plug-in for the Zhi# compiler framework. Finally, the resulting set of constraints is generically computed using a brute-force approach where every valid constraint combination is added to the resulting list of constraining facets; exceptions are thrown and silently discarded for incompatible constraint combinations (lines 20–27).

Pairwise combinations of constraints are sufficient for interval addition and subtraction where  $[x_1, x_2] + [y_1, y_2] = [x_1 + y_1, x_2 + y_2]$  and  $[x_1, x_2] - [y_1, y_2] = [x_1 - y_2, x_2 - y_1]$ , respectively, while multiplication ( $[x_1, x_2] \cdot [y_1, y_2] = \min(x_1 \cdot y_1, x_1 \cdot y_2, x_2 \cdot y_1, x_2 \cdot y_2), \max(x_1 \cdot y_1, x_1 \cdot y_2, x_2 \cdot y_1, x_2 \cdot y_2)$ ) and division ( $[x_1, x_2] / [y_1, y_2] = [x_1, x_2] \cdot (1/[y_1, y_2])$ , where  $1/[y_1, y_2] = [1/y_2, 1/y_1]$  if  $0 \notin [y_1, y_2]$ ) would require constraint and type awareness in the *getAdditionConstraints(T1, T2)* method. The architectural model of the  $\lambda_C$ -type system allows such type and constraint specific refinements of the constraint arithmetic. These refinements, however, were not implemented since the resulting complexity of such refinements is unlikely to be tractable by programmers. Implemented XSD constraint combinations are listed in Table E.3.

```

1 CTSimpleType add(T1:CTSimpleType, T2:CTSimpleType) {
2   if (areCompatible(T1, T2)) {
3     CTSimpleType T = makeAnonymousUnconstrained(clone(T1));
4     CTSimpleType T1' = materialize(T1);
5     CTSimpleType T2' = materialize(T2);
6     T.constraints = getAdditionConstraints(T1', T2');
7     return materialize(T);
8   } else {
9     throw new IncompatibleTypeException();
10  }}
11 CTSimpleType materialize(T:CTSimpleType) {
12   List<ConstrainingFacet> constraints =                               ↵
13   removeRedundantConstraints(T.constraints);
14   constraints = T.homogenizeRelatedConstraints(constraints);
15   constraints = T.inferResultingConstraints(constraints);
16   CTSimpleType T' = T.clone();
17   T'.constraints = constraints;
18   return T';
19 }
20 List<ConstrainingFacet> getAdditionConstraints(T1, T2) {
21   List<ConstrainingFacet> constraints = new List<ConstrainingFacet>();
22   foreach(ConstrainingFacet c1 in T1.constraints) {
23     foreach(ConstrainingFacet c2 in T2.constraints) {
24       try {
25         constraints += c1.add(c2);
26       } catch { }
27   }}}

```

Figure 3.3:  $\lambda_C$ -constraint arithmetic algorithm

### 3.8 Related Work

Foster et al. [FTA02] developed the CQUAL tool that can be used to extend standard types with flow-sensitive type qualifiers. The formal foundation of CQUAL is constituted by the type system of the call-by-value lambda calculus that was extended with pointers and type qualifier annotations. Type inference checks that given annotations are correct, where CQUAL's flow sensitivity is restricted to the decorating type qualifiers. In the  $\lambda_C$ -calculus, types are explicitly constructed using value space constraints, which may be modified entirely by also flow-sensitive type inference. CQUAL's type qualifiers are rather complementary to the actual type information in order to, for example, annotate objects with required state information (e.g., to describe an *open File*) and there are only ad hoc user-defined subtype relations between type qualifiers while there are formal subtyping rules for value space constraints in the  $\lambda_C$ -calculus.

Brian Chin et al. introduced semantic type qualifiers [CMM05] to support user-defined type refinements, which ensure additional invariants of interest (e.g., *nonnull*, *nonzero*). Just like in the  $\lambda_C$ -calculus, value-qualified types are always considered to be subtypes of their associated unqualified types (cf. the width subtyping rules in the  $\lambda_C$ -calculus). Still, in contrast to the  $\lambda_C$ -type system there is no support for explicit subtype declarations between user-defined qualifiers. An important difference is that in the work of Brian Chin et al. associated type rules of type qualifiers are defined for particular code patterns (i.e. knowledge is assumed about the syntax of the programming language), which is not required for constrained type definitions in Zhi#. On the other hand, Brian Chin et al. provide an automatic soundness checker to prove the declared invariants for type qualifier definitions. In their subsequent CLARITY approach [CMM06], qualifier rules can be automatically inferred from given invariants. Both CQUAL and CLARITY were instantiated in frameworks for user-defined type qualifiers in C programs.

The JQUAL tool [GF07] adds user-defined type qualifiers to Java. JQUAL supports subtyping orders of type qualifiers, which, however, must be supplied manually.

### 3.9 Summary

The author devised an extension of the simply typed lambda calculus with subtyping ( $\lambda_{<}$ ) for constrained atomic data types. Atomic data types can only have atomic values, which are not allowed to be further fractionalized. A primitive atomic data type is a three-tuple consisting of a set of distinct values called its value space, a set of lexical representations called its lexical space, and a set of fundamental facets. The value space is the set of values for a given data type. Each value in a value space is denoted by at least one literal of its lexical space. Fundamental facets semantically characterize abstract properties of the value space. The  $\lambda_C$ -calculus includes the fundamental facets equality, order, boundedness, cardinality, numeric, and date-time.

Constrained data types are computational structures where the only operations are constraint applications. Constraining facets are optional properties that can be applied to a data type to restrict its value space. Constraining facets can be modifying (i.e. string literals that are assigned to instances of constrained types are implicitly modified upon assignment) or enforcing (i.e. certain value space constraints must hold for the interpretations of string literals that are assigned to instances of constrained types). Three subsumption rules were defined for constraints based on their application on atomic data types. A constraint is a sub-constraint of another constraint if the value space that results from the application of the former constraint on a data type is a subset of the value space that results from the application of the latter constraint on the same data type. Constraints can be more restrictive based on the widths and depths of their definitions.

In the type system of the  $\lambda_C$ -calculus, data types are inductively defined by their value spaces (i.e. the set of values for a data type). Starting from a set of built-in primitive types with axiomatically defined value spaces, atomic types are derived through the application of value space constraints. Constraint applications on atomic types can be reduced to set operations on the types' value spaces. The form of type construction in the  $\lambda_C$ -calculus can be used to mimic type derivation as in XML Schema Definition.



An atomic type is a subtype of another type if the value space of the former is a subset of the value space of the latter. In this way the basic rules of subtyping are effective: reflexivity, transitivity, and subsumption. Subtyping rules were defined that capture the intuition that it is safe to add constraints to an atomic type and to use more specific constraints for type definition. Constraint applications always make types more specific. The fact that constraint applications can only reduce the value space of a type is a required property for the soundness of the  $\lambda_C$ -type system, which was proved based on a straightforward induction on a derivation of a typed term in the  $\lambda_C$ -calculus.

It is possible to infer transient constraints that may hold for the instances of constrained types within only a limited scope of a program. Type inference rules were defined that capture the two intuitions of adding constraints to the type of a variable for a limited scope and eventually removing them upon modifications of the variable. The proposed type inference algorithm is conservative to keep it comprehensible for programmers. The implementation of the  $\lambda_C$ -calculus uses an extensive constraint arithmetic that can be used to infer the types of binary expressions that involve constrained types. Without loss of the soundness property of the  $\lambda_C$ -type system, the implementation of the  $\lambda_C$ -calculus introduces reasonable compatibilities between intuitively compatible constraining facets and primitive base types.

The  $\lambda_C$ -type system as described in this chapter was fully implemented in Java and C# for the XML Schema Definition type system. These implementations can be used to load type definitions from XSD files and classify these types in a hierarchy. The Java implementation is used in the CHIL Knowledge Base Server to facilitate the handling of OWL datatype properties (see Chapter 4). The C# implementation is used in the Zhi# compiler plug-in for static typing and dynamic checking of XML Schema Definition data types (see Chapter 5). The architectural model of the  $\lambda_C$ -implementation makes it possible to join an arbitrary number of particular type system implementations in one single type pool. It is conceivable to implement further type systems that are similar to XML Schema Definition such as, for example, the SQL or OCL type systems.



## CHAPTER 4

### The CHIL OWL API

As the underlying Semantic Web standards such as RDF(S) [MM04, BG04b], DAML+OIL [HHP01], and their common Description Logics (DL) [BCM03] based successor OWL DL [MH04a] have matured, tools for ontology engineering have emerged both in commercial as well as in academic fields. Knowledge acquisition systems such as Protégé [Sta06] make it particularly easy to construct domain ontologies and to enter data. Ontology management systems such as HP Labs' Jena [HP 04] can be used for loading OWL DL ontologies from files and via the Internet and for creating, modifying, querying, and storing ontologies. Inference engines such as RACER [MH04b] and Pellet [Pel06] provide support for query answering. There is a growing set of tools, projects, and applications for the *SHOIN(D)* Description Logic based Web Ontology Language (OWL DL). However, processing ontological information using existing off-the-shelf ontology management systems is still laborious and error-prone. From the author's experience, this is inter alia caused by two main problems.

Firstly, there are no formal specifications that fully define the semantics of ontology management APIs. This is particularly problematic since typical interface methods of OWL APIs (e.g., *listSubclasses*, *addIndividual*) are closely related to the semantics of the formal foundations (i.e. Description Logics) of the Web Ontology Language.

Secondly, existing off-the-shelf ontology management systems provide only limited connectivity with respect to native support for programming languages and remoting protocols. Hence, it is particularly difficult to use ontology management systems remotely or along with a variety of different programming languages (i.e. in heterogeneous

distributed computing environments). More severely, it may be unfeasible to replace an ontology management system by alternative products without rewriting significant parts of client code.

This chapter describes a pluggable architectural model for an ontological knowledge base server, which can expose the ontology management functionality of arbitrary off-the-shelf OWL DL systems via a well defined API. A combination of Description Logics terminology and Floyd-Hoare logic [Hoa69] was used to formally specify the result sets and side effects of each method.

The architectural model of the developed knowledge base server facilitates adding and replacing remoting protocol hosts and replacing the adapter code for arbitrary off-the-shelf OWL DL management systems. Implementations of the knowledge base server API can automatically be tested for adherence to the formal specification using a regression test framework. In order to make it particularly easy for programmers to use the developed OWL API, a code generation tool was devised that automatically generates client libraries for a number of programming languages.

The ontological knowledge base server along with the code generation tool as described in this chapter is actively used in the CHIL research project [Inf04]. This is why it will be referred to as the *CHIL Knowledge Base Server*; the OWL API that was defined in this work will be called the *CHIL OWL API*. The CHIL research project aims to introduce computers into a loop of humans interacting with humans, rather than condemning humans to operate in a loop of computers. In order to implement unobtrusive user friendly services a semantic middleware has been developed that fusions information provided by so called *perceptual components* in meaningful ways. Each perceptual component (e.g., image and speech recognizers, body trackers, etc.) contributes to the common domain of discourse. The Web Ontology Language OWL DL was used to replace previous domain models that had been based on particular programming languages. The CHIL Knowledge Base Server is used as the back-end of an extensive semantic middleware [PRS06].

Whereas there are several approaches that aim to provide programming language independent APIs for processing ontological knowledge bases, to the best of the author's knowledge, the CHIL Knowledge Base Server is first to combine the following features.

- The CHIL OWL API is solely based on primitive built-in XML Schema Definition data types and is remotely accessible by virtually every programming language capable of parsing strings and of communicating via TCP sockets.
- A combination of *SHOIN(D)* Description Logic semantics with Floyd-Hoare logic was used to formally specify the result sets and side effects of each interface method.
- A regression test framework was developed for automatically validating implementations of the CHIL OWL API for adherence to the formal specification.

The outline of this chapter is as follows. Section 4.1 elucidates the architectural model of the CHIL Knowledge Base Server and the design of the CHIL OWL API. The formal specification of this API and a testing framework, which can be used to automatically validate particular implementations, are described in Section 4.2. Section 4.3 describes the implementation of an example scenario. Section 4.4 gives an overview of the connectivity capabilities and API specifications of some of the most widely used ontology management systems.

## 4.1 The CHIL Knowledge Base Server

The CHIL Knowledge Base Server is an adapter written in Java for off-the-shelf ontology management systems that implements a formally specified and well defined OWL DL API. It supports the interchange of adapted reasoners and ontology management systems. Moreover, client libraries can automatically be generated for a number of different programming languages.

### 4.1.1 Architectural model

For the CHIL Knowledge Base Server, the following most crucial requirements had to be met. Since the server is targeted for a distributed system, it must be accessible both locally and remotely through a single interface. Since client components in the CHIL research project may be written in a variety of different languages (e.g., Java, C#, C++, Python), the remote interface must be programming language independent. Data representation must be architecture independent such that, mixed use of architectures with little-endian and big-endian byte order does not lead to interoperability issues. The CHIL Knowledge Base Server must be capable of handling multiple, potentially competing incoming requests in parallel without corrupting the underlying database (i.e. thread-safe server design). Since in the CHIL research project the Web Ontology Language OWL DL is used, the CHIL OWL API should be tailored specifically to OWL DL, rather than providing a more verbose and potentially error-prone interface to a more general ontology model. Finally, it must be possible to replace adapted ontology management systems.

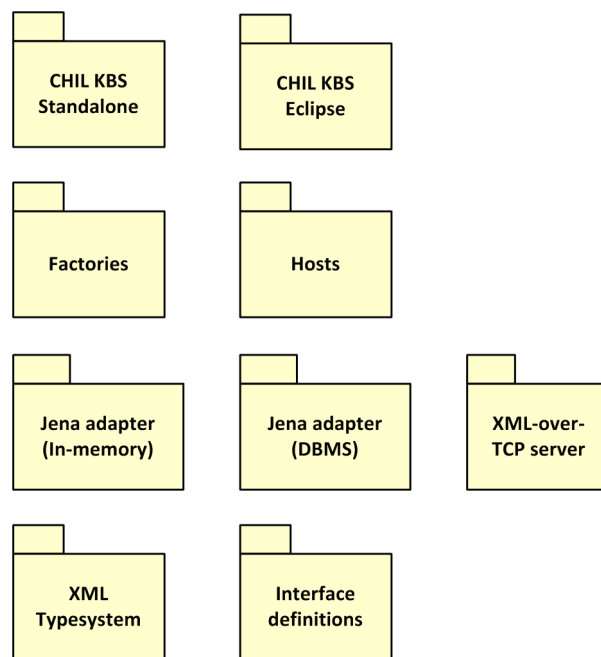


Figure 4.1: CHIL Knowledge Base Server components

The CHIL Knowledge Base Server software comprises several Java components as depicted in Fig. 4.1. Excluding the automatically generated client libraries, the entire software amounts to 15,000 lines of code (LOC).

Since the Web Ontology Language specification allows atomic XML data types as the range for OWL datatype properties, a full-fledged type system for constrained atomic data types following the formal foundations of the  $\lambda_C$ -calculus as described in Chapter 3 was implemented. Thus, it is possible to register XML Schema Definitions with the CHIL Knowledge Base Server in order to properly consider range restrictions of OWL datatype properties. This validation component enables improved type safety compared to other existing ontology management systems. Still, it is the responsibility of the adapted inference engine to which extent the type information of RDF literals are used for reasoning. The implementation of the XML type system component comprises 4,000 LOC. The interfaces of the CHIL Knowledge Base Server application are defined in a separate component, which totals 1,000 LOC. Up to now adapters are available for the Jena Semantic Web Framework [HP 04], which can be configured to use a transient in-memory ontology model (2000 LOC) or a persistent RDF database. The DBMS adapter (500 LOC) is an extension of the in-memory model adapter and supports Microsoft SQL Server [Mic07b], MySQL [MyS07], and PostgreSQL [Pos07] as possible database back-ends. The CHIL OWL API is exposed via an XML-over-TCP interface (3000 LOC). While it is conceivable to support further remoting protocols in the future, the XML-over-TCP port appeared to be most compatible in order to be accessible from a variety of programming languages. The factories (500 LOC) and hosts packages (500 LOC) implement functionality to dynamically load ontology management system adapters and remoting protocol hosts. Every implementation of these interfaces is loaded in a separate class loader in order to avoid library version conflicts and dependency issues. The CHIL Knowledge Base Server can be run as a standalone application (500 LOC) and as an Eclipse plug-in (3000 LOC) as shown in Fig. 4.2 and 4.3, respectively. The Eclipse plug-in provides a GUI that can be used to control the CHIL Knowledge Base Server and to browse managed ontologies.

The architectural model on a class level of the CHIL Knowledge Base Server is partly depicted in Fig. 4.4. It makes extensive use of the Factory Design Pattern [GHJ94] and reflection capabilities of the Java programming language in order to be able to plug in hosts for arbitrary remoting protocols dynamically at runtime.

The *IOWLAPI* interface was automatically generated from the XML-based API definition as described in the following subsection. Up to now two implementations of this interface exist that adapt the Jena Semantic Web Framework to the CHIL OWL API. The *OWLAPIJena* adapter configures Jena to use an in-memory ontology model. The *OWLAPIJenaDBMS* adapter provides for a persistent database back-end using Microsoft SQL Server, MySQL, or PostgreSQL. The adapter classes are dynamically loaded in a separate class loader by the respective implementations of the *IOWLAPIFactory* interface. Implementations of the *OWLAPIServer* interface make the CHIL OWL API available via remoting protocols. Again, particular implementations of the *OWLAPIServer* interface such as *XMLoverTCPSTerver*, which provides an XML-over-TCP port, are loaded dynamically in separate class loaders by implementations of the *IOWLAPIServerFactory*. Both the singleton instance of *IOWLAPI* as well as the arbitrary number of *OWLAPIServers* are aggregated by the *OWLAPIHost*. The *IOWLAPIHost* and *IOWLAPI* interfaces are referenced by server applications such as the stand-alone and Eclipse-based implementations of the CHIL Knowledge Base Server.

The author decided to implement a TCP/IP via the socket interface in the first place because in state-of-the-art operating systems, local TCP/IP connections are usually routed via loopback or similar devices that bypass most of the TCP/IP stack. The advantage of having a single interface for local and remote communication when using the socket interface even for local communication therefore fully outweighs any marginal performance slowdown or latency imposed by socket communication.

Language independence was achieved by a two-step approach. Firstly, all communication is performed by transmitting and receiving XML messages over TCP/IP, rather



```

C:\WINDOWS\system32\cmd.exe - kbs.bat chilkbsconfig.xml ontology.owl tt.xsd
Successfully loaded XML schema definition from file tt.xsd (q: Quit, r: Reset, h
: help)

Starting remoting protocol servers...
Status of XML-over-TCP server: STARTED
CHIL Knowledge Base Server STARTED (q: Quit, r: Reset, h: help)
h
i: print configuration information
f: load ontology from file
u: load ontology from URL
l: load XML schema definition from file
w: load XML schema definition from URL
s: print statements
c: print classes
t: print datatypes
o: print object properties
d: print datatype properties
x: start/stop servers
r: reset ontology
v: validate ontology
q: quit

```

Figure 4.2: CHIL Knowledge Base Server stand-alone application

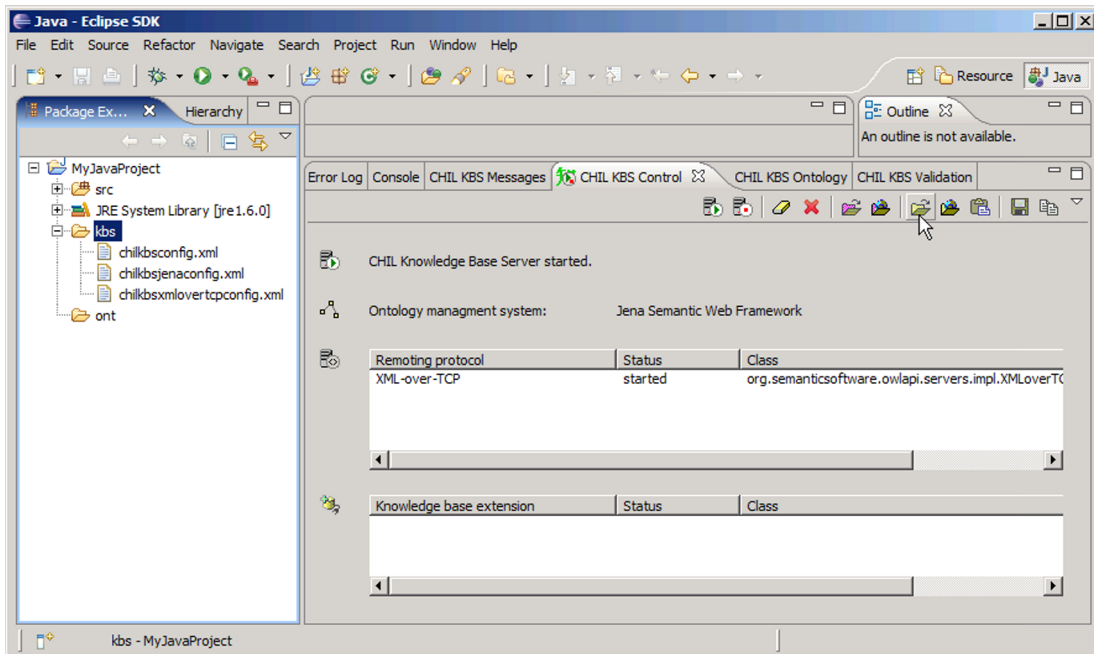


Figure 4.3: CHIL Knowledge Base Server Eclipse plug-in

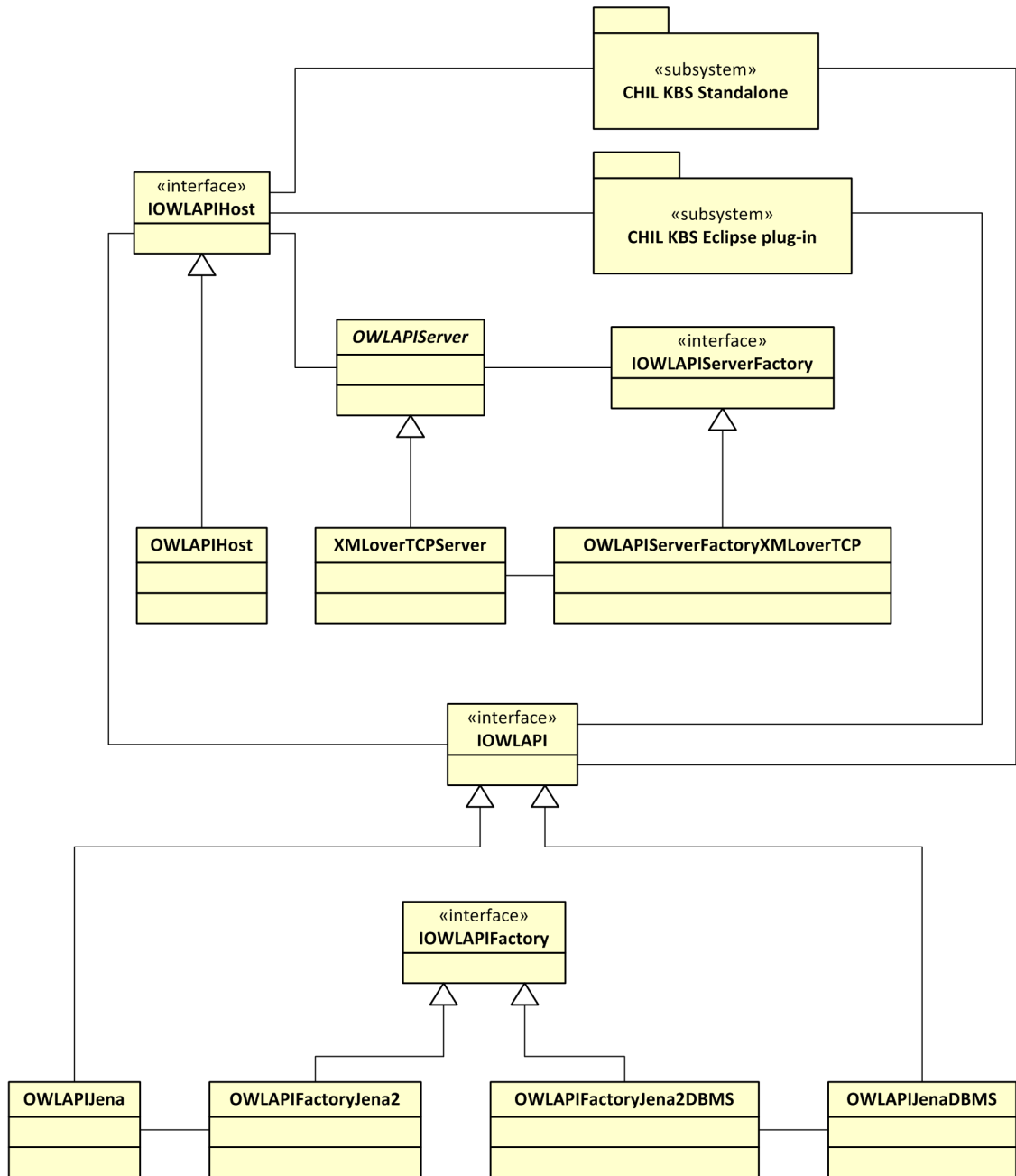


Figure 4.4: CHIL Knowledge Base Server architectural model

than building upon some language dependent RPC-based mechanism. Data values are encoded with XML data types, rather than using programming language-specific data types. While with XML messages based on XML data types one achieves a highly programming language independent communication mechanism, one does not want to put the burden of generating and parsing XML messages on client programmers. Therefore, as a complementary step of the presented approach, for a selected set of programming languages, client libraries are provided that handle all XML-related work. Up to now client libraries are available for Java (3000 LOC), C# (3500 LOC), and C++ (4500 LOC).

The CHIL Knowledge Base Server handles multiple incoming requests with standard socket calls (i.e. listen on server port, accept request, rebind to different port). Each request is rebound to a different port and delegated to a thread of its own. In this way, another connection on the server port can be accepted while the previous one is still being processed. In order to avoid corruption of the managed knowledge base data by concurrent access, without relying on the capabilities of the underlying ontology management system, all incoming requests are strictly serialized before they are executed on live data.

#### **4.1.2 The CHIL OWL API**

The CHIL Knowledge Base Server was designed to locally and remotely store, manage, and retrieve arbitrary ontological data that meets OWL DL. Originally designed for use in the highly distributed, heterogeneous environment of the CHIL research project, special emphasis was put on good connectivity and compatibility with the most widely used programming languages and runtime environments.

The CHIL OWL API definition is given as an XML instance document. Its XML Schema Definition is twofold. The first part as depicted in Fig. 4.5 provides a schema for defining interfaces and methods on the meta-level. Interfaces and methods are given names. The optional descriptions can be used to generate source code comments. A method signature comprises an arbitrary number of named formal parameters. For formal

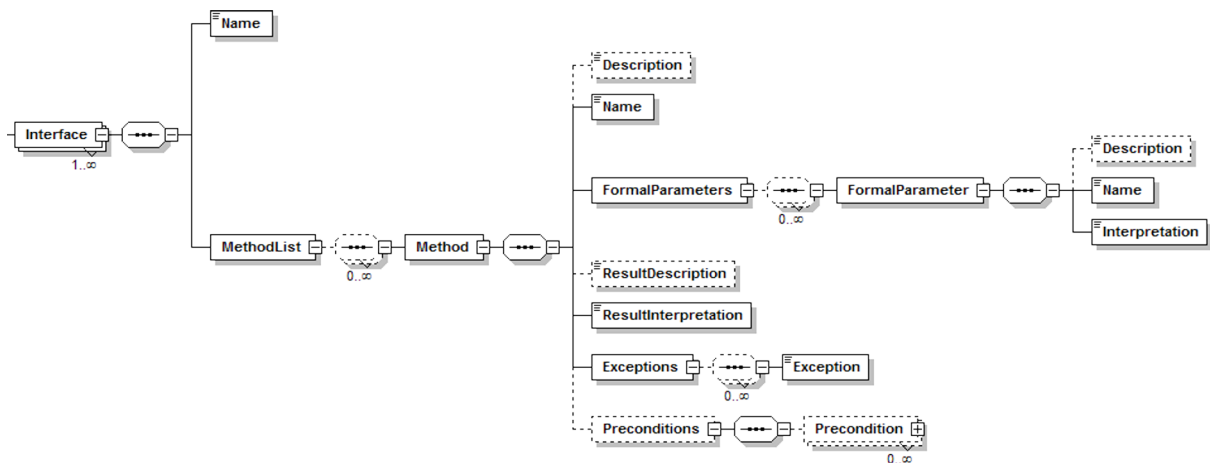


Figure 4.5: CHIL OWL API meta-level schema

parameters and the return value of each method *interpretation* information are provided, which give meaning to them. Formal parameters and return values can be interpreted as one of the OWL DL ontology elements listed in Table 4.1. These interpretation information are used in two ways. First, they are translated into programming language data types during code generation. Second, they are used in the formal specification of the CHIL OWL API elucidated in Section 4.2.

Note that the meta-level schema does not impose the usage of particular programming language data types to represent elements of an ontology. Instead, CHIL OWL API client libraries may use arbitrary data types that are most appropriate in a particular programming language to represent particular elements of an ontology (as long as these data types are correctly marshalled over the wire). As such, it is conceivable to use, for example, an array of strings or dynamically growing container classes such as lists or vectors to represent a list of ontological individuals.

CHIL OWL API methods may require certain preconditions to hold in order to execute properly. The precondition names used in the CHIL OWL API meta-level schema refer to the formally specified preconditions given in Appendix F, which were used to automatically generate regression test cases as explained in Subsection 4.2.3.

Table 4.1: CHIL OWL API element interpretations

annotationProperty	boolean	class
comment	datatype	dataValuedProperty
individual	individualValuedProperty	label
language	listOfAnnotationProperties	listOfClasses
listOfDatatypes	listOfDataValuedProperties	listOfIndividuals
listOfIndividualValuedProperties	listOfLiteralValues	listOfOntologies
listOfOntologyProperties	listOfStatements	literalValue
naturalNumber	numberOfIndividuals	numberOfLiteralValues
ontology	ontologyProperty	property
rdfXML	resource	URIReference
URL	validityReport	versionInfo
void	xsdXML	

The CHIL OWL API comprises 37 tell operations to modify the TBox, 16 tell operations to modify the ABox, 26 ask operations to query the TBox, and 12 ask operations to query the ABox of OWL DL knowledge bases. In addition, there are 15 methods to query and modify the annotations of ontology elements and 17 auxiliary methods to, for example, load an ontology from a file or to serialize an ontology model to its RDF/XML syntax. These methods are not included in the formal specification of the CHIL OWL API since their semantics cannot be founded on Descriptions Logics.

The following fragment of the XML-based CHIL OWL API definition describes the *listSubClasses* method, which takes as input an ontological concept (line 4–7) and returns a list of subsumed concepts (line 9). The XML-based method definition does not make any assumptions about the programming language data types that may eventually be used to denote the elements of an ontology (e.g., the name of the concept). Instead, interpretation information are given that identify the input value as an ontological concept (line 6) and the return value as a list of concepts (line 9). An *OntologyManagementException* (line 11) and *UndeclaredConceptException* (line 12) is thrown if the adapted

ontology management systems fails and the input concept is undefined in the ontology, respectively. The exceptions that may be thrown by CHIL OWL API methods are listed in Table 4.2. Again, this list is used both for code generation and in the formal API specification.

```

1 <Method>
2   <Name>listSubClasses</Name>
3   <FormalParameters>
4     <FormalParameter>
5       <Name>owlClass</Name>
6       <Interpretation>class</Interpretation>
7     </FormalParameter>
8   </FormalParameters>
9   <ResultInterpretation>listOfClasses</ResultInterpretation>
10  <Exceptions>
11    <Exception>OntologyManagementException</Exception>
12    <Exception>UndeclaredConceptException</Exception>
13  </Exceptions>
14  <Preconditions>
15    <Precondition>
16      <DeclaredConcept>
17        <Concept>owlClass</Concept>
18      </DeclaredConcept>
19    </Precondition>
20  </Preconditions>
21 </Method>

```

The two listings below show the automatically generated Java code for the XML-over-TCP server and the C# client code for the *listSubClasses* method, respectively. In Java and C#, the *java.lang.String* and *System.String* data types are used, respectively, to denote names of ontological concepts. The mapping from the ontology element interpretations in Table 4.1 to programming language data types is defined by the code generator for the particular programming language. In contrast to other OWL APIs, the CHIL

OWL API does not depend on programming language specific data types. Moreover, the entire interface definition is service oriented, which also facilitates remote access and integration with non object-oriented client programming languages.

Table 4.2: CHIL OWL API exceptions

InconsistentOntologyException	InvalidConceptNameException
InvalidDataValueException	InvalidIndividualNameException
InvalidLanguageException	InvalidNamespaceException
InvalidPropertyDomainException	InvalidPropertyNameException
InvalidPropertyRangeException	InvalidPropertyValueException
InvalidXMLSchemaDefinitionException	NotANaturalNumberException
OntologyException	OntologyIOException
OntologyManagementException	OntologySerializationException
UncomparableTypesException	UndeclaredConceptException
UndeclaredDatatypeException	UndeclaredDatatypePropertyException
UndeclaredIndividualException	UndeclaredNamespaceException
UndeclaredObjectPropertyException	UndeclaredPropertyValueException
UndeclaredResourceException	UndefinedCommentException
UndefinedIsDefinedByException	UndefinedLabelException
UndefinedSeeAlsoException	UndefinedVersionInfoException

```

1  [...]
2  // Handling of incoming 'listSubClasses' request message
3  else if ("listSubClasses".equals(strMethodName)) {
4  try {
5      String strP0V = nParameter.getFirstChild().getNextSibling().      ↪
6      getFirstChild().getNodeValue();
7      nParameter = nParameter.getNextSibling();
8      String[] arr = owlapi.listSubClasses(strP0V);
9      Element elResult = elResponse.getOwnerDocument().createElementNS(      ↪
10     "http://www.semantic-software.org/CHILKBSOWLAPI", "Result");
11     elResponse.appendChild(elResult);

```

```

12 Element elInterpretation = elResponse.getOwnerDocument(). ↵
13     createElementNS("http://www.semantic-software.org/CHILKBSOWLAPI", ↵
14     "Interpretation");
15 elInterpretation.appendChild(elResponse.getOwnerDocument(). ↵
16     createTextNode("listOfClasses"));
17 elResult.appendChild(elInterpretation);
18 for (int i = 0; i < arr.length; i++)
19 {
20     Element elValue = elResponse.getOwnerDocument(). ↵
21     createElementNS("http://www.semantic-software.org/CHILKBSOWLAPI", ↵
22     "Value");
23     elValue.appendChild(elResponse.getOwnerDocument(). ↵
24     createTextNode(arr[i].toString()));
25     elResult.appendChild(elValue);
26 }
27 sendResponseDocument(elResponse.getOwnerDocument());
28 } catch (Exception ex) {
29     sendResponseDocument(makeExceptionResponseDocument(ex));
30 }
31 }

```

```

1 public string[] ListSubClasses(string owlClass) {
2     XmlElement elRequest = MakeRequestElement();
3     XmlElement elMethodName = elRequest.OwnerDocument. ↵
4     CreateElement("owlapi", "MethodName", ↵
5     "http://www.semantic-software.org/CHILKBSOWLAPI");
6     elMethodName.AppendChild(elRequest.OwnerDocument. ↵
7     CreateTextNode("listSubClasses"));
8     elRequest.AppendChild(elMethodName);
9     XmlElement elParameters = elRequest.OwnerDocument. ↵
10    CreateElement("owlapi", "Parameters", ↵
11    "http://www.semantic-software.org/CHILKBSOWLAPI");
12    elRequest.AppendChild(elParameters);
13    XmlElement elParameter0 = elRequest.OwnerDocument. ↵

```



```

14 CreateElement("owlapi", "Parameter", ↵
15 "http://www.semantic-software.org/CHILKBSOWLAPI");
16 elParameters.AppendChild(elParameter0);
17 XmlElement elInterpretation0 = elRequest.OwnerDocument. ↵
18 CreateElement("owlapi", "Interpretation", ↵
19 "http://www.semantic-software.org/CHILKBSOWLAPI");
20 elInterpretation0.AppendChild(elRequest.OwnerDocument. ↵
21 CreateTextNode("class"));
22 elParameter0.AppendChild(elInterpretation0);
23 XmlElement elValue0 = elRequest.OwnerDocument. ↵
24 CreateElement("owlapi", "Value", ↵
25 "http://www.semantic-software.org/CHILKBSOWLAPI");
26 elValue0.AppendChild(elRequest.OwnerDocument.CreateTextNode(owlClass));
27 elParameter0.AppendChild(elValue0);
28 XmlDocument domResponse = SendReceive(elRequest.OwnerDocument);
29 TestForExceptions(domResponse);
30 XmlNamespaceManager nsmgr = ↵
31 new XmlNamespaceManager(domResponse.NameTable);
32 nsmgr.AddNamespace("oa", ↵
33 "http://www.semantic-software.org/CHILKBSOWLAPI");
34 XmlNodeList nlValues = domResponse. ↵
35 SelectNodes("/oa:CHILKBSOWLAPI/oa:Response/oa:Result/oa:Value", nsmgr);
36 if (nlValues == null) {
37 return null;
38 } else {
39 string[] values = new string[nlValues.Count];
40 for (int i = 0; i < nlValues.Count; i++)
41 {
42 values[i] = nlValues[i].FirstChild.Value;
43 }
44 return values;
45 }
46 }

```

The second part of the CHIL OWL API definition as shown in Fig. 4.6 describes the format of request and response messages on the object level (i.e. the format of the messages that are sent over the wire). In order to make sure that XML messages sent between the CHIL Knowledge Base Server and XML-over-TCP clients adhere to this schema definition both the XML-over-TCP server component as well as the client libraries are generated automatically as shown in the two preceding Java and C# listings. The code generation approached helped a great deal during the development phase of the CHIL OWL API. Client libraries could be automatically updated to instantaneously reflect changes in the API definition.

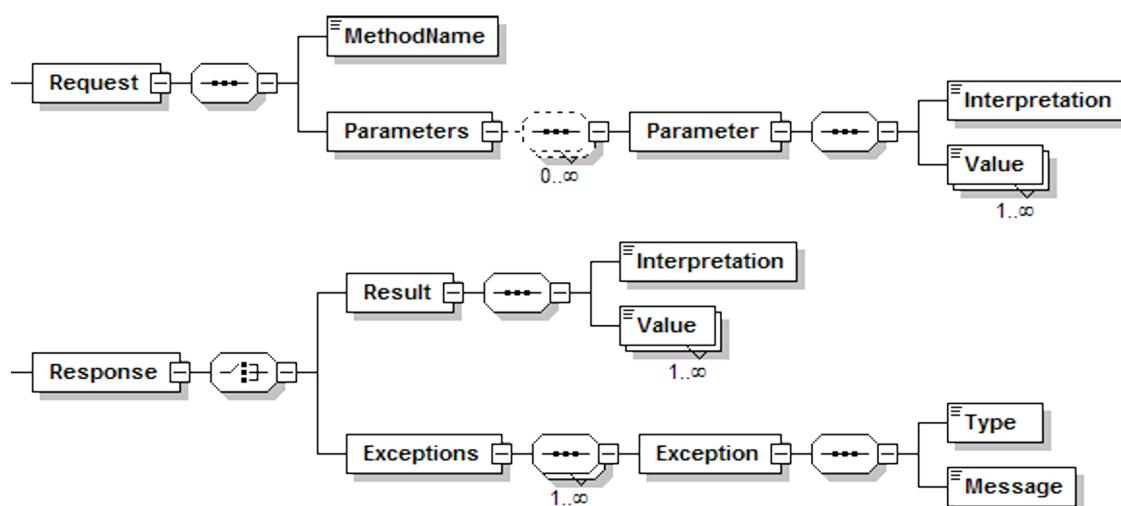


Figure 4.6: CHIL OWL API object-level schema

## 4.2 A Formally Specified OWL API

A formal specification of the CHIL OWL API was devised in order to make it possible to consistently adapt off-the-shelf ontology management systems and to provide knowledge base clients with well defined semantics of the OWL API methods. In particular, ambiguities had to be resolved that may be caused by informal specifications such as “Add a property to this resource. A statement with this resource as the subject,  $p$  as the predicate

and  $o$  as the object is added to the model associated with this resource.” (see HP Lab’s Jena’s documentation of the `com.hp.hpl.jena.rdf.model.Resource.addProperty(Property p, boolean o)` method). In this case, it remains totally unclear what the preconditions are and under what circumstances the method will execute properly (i.e. what happens when the host individual is undefined in the ontology or when the property value’s type is different from the range description of the referred property). Also, with only informal documentation there is no way to know how an ontology looks after a sequence of method invocations, which makes it impractical to validate implementations of textually specified OWL APIs since there is no practical easy way to predict the results and side effects of method calls in advance and on a meta-level.

In a first approach on formalizing the semantics of the CHIL OWL API [PRS06], the author had utilized the Z notation language [Spi01] as a formal framework. In order to accommodate the Z notation better to the needs of OWL, we had extended the Z notation syntax by Description Logics notation. With this extended ontological Z notation, we were able to formally specify the semantics of the OWL API of the CHIL Knowledge Base Server in a clear and straightforward way. While this initial approach of using an extended Z notation proved to be handy for specifying the CHIL OWL API’s operational semantics, it turned out to have only limited usefulness for practical exploitation of our API specification. In particular, the lack of Z notation tools that support our extended notation made our specification itself useful only in theory. We could have tried to develop a mapping from our extended syntax to pure Z notation syntax in a pre-processing manner in order to work around the lack of such tools. However, it turned out that such a mapping would have gotten at least as complicated as the formal specification of OWL DL itself, thus losing the advantage of the handy syntax.

In this section, an alternative approach is presented to formalize the semantics of the CHIL OWL API, using Floyd-Hoare logic [Hoa69] as a formal framework. Floyd-Hoare logic provides a set of logical rules in order to reason about the correctness of computer programs with the rigor of mathematical logic. This section concludes with an example

of how one can exploit the Floyd-Hoare logic based formal specification to predict the effects of a sequence of CHIL OWL API method calls by applying standard Floyd-Hoare logic inference rules.

### 4.2.1 Notational framework

In Floyd-Hoare logic, theorems are of the form  $\{P\}Q\{R\}$ , where  $P$  and  $R$  are *assertions* and  $Q$  is a *program*.  $P$  is called the *precondition* and  $R$  the *postcondition*. Standard Floyd-Hoare logic proves only partial correctness, while termination would have to be proved separately. Thus, the intuitive reading of a Hoare triple is: Whenever  $P$  holds of the state before the execution of  $Q$ , then  $R$  will hold afterwards, or  $Q$  does not terminate. In particular, if  $Q$  does not terminate the state of the computation is undefined, so  $R$  can be any statement at all. In Hoare's original paper [Hoa69] six axioms are given as shown in Table 4.3. There are two general logical rules, an assignment axiom and three rules for control constructs. In the assignment axiom HR-ASSIGN,  $P[x/E]$  denotes that in expression  $P$  all free occurrences of variable  $x$  have been replaced by expression  $E$ . In rule HR-WHILE,  $P$  is the *loop invariant*.

Table 4.3: Hoare logic axioms and rules

<i>Axiom</i>	<i>Name</i>
$\frac{P' \Rightarrow P, \{P\}Q\{R\}}{\{P'\}Q\{R\}}$	(HR-PRESTRENGTH)
$\frac{\{P\}Q\{R\}, R \Rightarrow R'}{\{P\}Q\{R'\}}$	(HR-POSTWEAK)
$\frac{}{\{P[x/E]\} x := E \{P\}}$	(HR-ASSIGN)

Table 4.3: Hoare logic axioms and rules

<i>Axiom</i>	<i>Name</i>
$\frac{\{P\}Q_1\{R\}, \{R\}Q_2\{S\}}{\{P\}Q_1; Q_2\{S\}}$	(HR-SEQUENCE)
$\frac{\{B \wedge P\} Q_1 \{R\}, \{\neg B \wedge P\} Q_2 \{R\}}{\{P\} \text{ if } B \text{ then } Q_1 \text{ else } Q_2 \text{ endif } \{R\}}$	(HR-COND)
$\frac{\{P \wedge B\} Q \{P\}}{\{P\} \text{ while } B \text{ do } Q \text{ done } \{\neg B \wedge P\}}$	(HR-WHILE)

The CHIL OWL API methods are specified as Hoare triples following the schema in Fig. 4.7 shown below.

$$\begin{aligned}
 \{P\}: & \quad \{\eta(\mathcal{O}) \wedge (\bigwedge_{i \in 0..n} \text{PC-OWLAPI-}X_i)\} \\
 Q: & \quad \mathcal{O}, m(p_1, \dots, p_n) \\
 \{R\}: & \quad \{(\mathcal{O} \vdash (\mathit{SHOIN}(\mathbf{D}) \text{ notation})) \wedge (\mathcal{R} = \mathit{SHOIN}(\mathbf{D}) \text{ notation})\}
 \end{aligned}$$

Figure 4.7: CHIL OWL API Hoare triple schema

In precondition  $P$ , the function  $\eta(\mathcal{O})$  returns true for consistent ontologies and false for inconsistent ontologies. Thus, the function  $\eta(\mathcal{O})$  is used to assert the consistency of ontology  $\mathcal{O}$ . Further assertions are given as a conjunction of properties of ontology  $\mathcal{O}$  and of the input parameters of the method under consideration. The Hoare triples of the CHIL OWL API specification reference 18 recurring basic preconditions that are described in Appendix F. These basic preconditions are given as conjunctions of Boolean expressions. Each clause in the Boolean expressions is a statement about, for example, the ontology under consideration. In practice, all conjoined basic preconditions must hold

before the execution of a method of the CHIL OWL API. The basic preconditions are named PC-OWLAPI-... and are parameterized with formal parameters, which can be used to devise the Boolean expressions in the *Requires* field of the precondition description (see the schema of the exemplary precondition specification in Fig. 4.8).

In program  $Q$ , the comma operator “,” applies the given method  $m$  – whose semantics one wants to specify – with formal parameters  $p_1, \dots, p_n$  on ontology  $\mathcal{O}$ .

In postcondition  $R$ ,  $\mathcal{SHOIN}(\mathbf{D})$  notation is used to describe 1) properties of ontology  $\mathcal{O}$  after the application of the specified method  $m$  and 2) the return value  $\mathcal{R}$  of method  $m$ . The symbol  $\emptyset$  denotes an empty result set, which in practice corresponds to the programming language return type *void*.  $\mathcal{SHOIN}(\mathbf{D})$  terminology is used because the  $\mathcal{SHOIN}(\mathbf{D})$  Description Logic constitutes the formal foundation of the Web Ontology Language OWL DL. Variables are used with respective semantics as given in Table 4.4. The (possibly subscripted) variables  $C$ ,  $D$ ,  $R$ ,  $U$ ,  $o$ , and  $v$  stand for concepts, data types, abstract roles, datatype roles, individuals, and data values, respectively. They may also be used as meta-variables ranging over variables; the context will make clear which is which. The complete specification of the CHIL OWL API is given in Appendix G.

Table 4.4:  $\mathcal{SHOIN}(\mathbf{D})$  metavariables

Constructor Name	Syntax	Semantics
atomic concept	$C$	$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
data type	$D$	$D^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
abstract role	$R$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
datatype role	$U$	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
individual	$o$	$o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
data value	$v$	$v^{\mathcal{I}} = v^{\mathbf{D}}$

The order of the conjoined basic preconditions in the Hoare triples of the specification of the CHIL OWL API is irrelevant for applying the Hoare logic axioms and rules in order to reason about the effects of subsequent method applications on an ontology. The

notation of the preconditions (i.e. the consistency check  $\eta(\mathcal{O})$  and the basic preconditions PC-OWLAPI- $X_i$   $i \in \{0..n\}$ ) is, however, positional when the specification of the CHIL OWL API is used for automatic code generation and the validation of the deterministic behavior of implementations of the CHIL OWL API specification. An ordered list of preconditions facilitates the automatic generation of regression test cases because for a particular basic precondition PC-OWLAPI- $X_i$  in the precondition  $P$  of a Hoare triple the ontology on which the test code operates can be assumed to be in a certain state.

Fig. 4.8 shows the specification of the basic precondition PC-OWLAPI-DECLARED-CONCEPT. The specifications of the basic preconditions comprise protocol information to create elements in the ontology under consideration (fields *Creates*) and to bind the precondition parameters to these elements (fields *Binds*) such that, the specified conditions initially evaluate to false and eventually to true. The specified exception (field *Expects*) is expected if the required properties (field *Requires*) do not hold for the ontology under consideration. Required predecessors of a basic precondition (field *Required predecessor*) are given to allow for chained up preconditions whose Boolean expressions in the *Requires* field assume a particular context (e.g., an individual with a particular name in the ontology). The provided information proved to be sufficient to automatically generate regression test code for the formally specified methods of the CHIL OWL API (see Subsection 4.2.3). The 18 basic preconditions are described in Appendix F.

PC-OWLAPI-DECLARED-CONCEPT( $C$ )	
Requires:	$C \in N_C$
Required predecessor:	—
Creates:	—
Binds:	$C = C_{\text{fresh}}$
Expects:	<i>UndeclaredConceptException</i>
Creates:	$C_A \sqsubseteq \top$
Binds:	$C = C_A$

Figure 4.8: Basic precondition PC-OWLAPI-DECLARED-CONCEPT

### 4.2.2 Examples

Following the Hoare triple schema in Fig. 4.7 and the notation of Table 4.4, the Hoare triples HT-OWLAPI-DECLARESUBCLASS and HT-OWLAPI-ADDINDIVIDUAL specify the CHIL OWL API methods *declareSubClass*( $C_1, C_2$ ) and *addIndividual*( $o, C$ ), respectively, as shown in Fig. 4.9.

(HT-OWLAPI-DECLARESUBCLASS)

$$\begin{aligned} & \text{declareSubClass}(C_1, C_2) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareSubClass}(C_1, C_2) \\ \{R\}: & \left\{ (\mathcal{O} \vdash C_1 \sqsubseteq C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

(HT-OWLAPI-ADDINDIVIDUAL)

$$\begin{aligned} & \text{addIndividual}(o, C) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{addIndividual}(o, C) \\ \{R\}: & \left\{ (\mathcal{O} \vdash o : C) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

Figure 4.9: Formally specified CHIL OWL API methods

The Hoare triple HT-OWLAPI-DECLARESUBCLASS should be read as follows. Given a consistent ontology  $\mathcal{O}$  with declared named concept descriptions  $C_1$  and  $C_2$ , the method call  $\mathcal{O}, \text{declareSubClass}(C_1, C_2)$  declares  $C_1$  to be a sub-concept of  $C_2$ , where  $C_1$  and  $C_2$  are meta-variables. The semantics of this sub-concept relation is that the interpretation  $\mathcal{I}$  of  $C_1$  (i.e. for consistent ontologies the set of individuals in the extension of the concept description  $C_1$ ) is a subset of  $C_2^{\mathcal{I}}$ . The return type of  $\mathcal{O}, \text{declareSubClass}(C_1, C_2)$  is the empty set, which in practice corresponds to the programming language return



type *void*. According to the Hoare triple HT-OWLAPI-ADDINDIVIDUAL, the method call  $\mathcal{O}, addIndividual(o, C)$  requires the concept  $C$  to be declared in the ontology. The postcondition asserts that the individual  $o$  is in the extension of the concept description  $C$ . Again,  $o$  and  $C$  are meta-variables.

With such Hoare triple based specifications available, Floyd-Hoare logic rules can be used to reason on a meta-level about the effect of subsequent method calls on an ontology. For example, assuming a previous method call  $\mathcal{O}, addSubclass(BodyTracker, Tracker)$ <sup>1</sup>, the Hoare triple HT-OWLAPI-DECLARESUBCLASS asserts only that in ontology  $\mathcal{O}$   $BodyTracker \sqsubseteq Tracker$  while a method call  $\mathcal{O}, addIndividual(BT, BodyTracker)$  requires the concept  $BodyTracker$  to be declared in the ontology (i.e. the named concept description  $BodyTracker$  is an element of the set of concept names of the ontology:  $\mathcal{O} \vdash BodyTracker \in N_C$ ). Using the Floyd-Hoare logic rule HR-PRESTRENGTH one can infer that the basic precondition PC-OWLAPI-DECLARED-CONCEPT( $BodyTracker$ ) is met by the postcondition  $\mathcal{O} \vdash BodyTracker \sqsubseteq Tracker$  of the previous method call  $\mathcal{O}, declareSubClass(BodyTracker, Tracker)$ .

$$\begin{array}{c}
 \text{HR-PRESTRENGTH:} \\
 \mathcal{O} \vdash C_1 \sqsubseteq C_2 \Rightarrow \mathcal{O} \vdash (C_1 \in N_C), \text{ HT-OWLAPI-ADDINDIVIDUAL} \\
 \hline
 \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash C_1 \sqsubseteq C_2 \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, addIndividual(o, C_1) \\
 \{R\}: \quad \left\{ (\mathcal{O} \vdash o : C_1) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

Figure 4.10: Application of the Floyd-Hoare pre-strengthening rule

The Hoare logic rule HR-SEQUENCE can be used to infer that, given a consistent ontology  $\mathcal{O}$ , the two subsequent method calls  $\mathcal{O}, declareClass(Tracker)$  and  $\mathcal{O}, addIndividual(T1, Tracker)$  result in an ontology that contains an individual T1, which is in the extension of the named concept description  $Tracker$ .

<sup>1</sup>In CHIL, a body tracker is a perceptual component that tracks the positions of persons as they move around in smart room environments.

$$\begin{array}{c}
\text{HR-SEQUENCE:} \\
\text{HT-OWLAPI-DECLARECLASS, HT-OWLAPI-ADDINDIVIDUAL} \\
\hline
\{P\}: \quad \left\{ \eta(\mathcal{O}) \right\} \\
\{Q\}: \quad \mathcal{O}, \text{declareClass}(C) ; \mathcal{O}, \text{addIndividual}(o, C) \\
\{R\}: \quad \left\{ (\mathcal{O} \vdash o : C) \wedge (\mathcal{R} = \emptyset) \right\}
\end{array}$$

Figure 4.11: Application of the Floyd-Hoare sequencing rule

Note that the postcondition  $R$  of no method of the CHIL OWL API asserts an ontology  $\mathcal{O}$  to be consistent after the application of a method on  $\mathcal{O}$ . In fact, this would require to intermingle the meta-level of the specification with the object level of particular ontologies. In order to be able to apply standard Hoare logic rules, the following assumption is made. After each method invocation on an ontology  $\mathcal{O}$  the consistency of  $\mathcal{O}$  is validated. If  $\mathcal{O}$  is inconsistent, the program terminates with an exception. Thus, for subsequent method calls on  $\mathcal{O}$  one can assume  $\mathcal{O}$  to be consistent since otherwise the program would not have executed until that point.

### 4.2.3 The CHIL OWL API testing framework

The CHIL Knowledge Base Server explicitly supports the replacement of adapted off-the-shelf ontology management systems. In order to make these changes transparent to knowledge base clients it is crucial for implementations of the CHIL OWL API to obey the operational semantics according to the specifications as elucidated in the previous subsection. In particular, an ordered sequence of basic preconditions is part of the formal specifications of CHIL OWL API methods. Implementations of the CHIL OWL API are required to check these preconditions in the given order. For example, the implementation of the *listObjectPropertyValuesOfIndividual*( $o, R$ ) method (see Appendix G.2) must examine the presence of individual  $o$  and object property  $R$  in the ontology. Regression test code must call the method in the context of an ontology such that, with the given parameters 1) the method fails once because of every violated basic precondition and 2) the

method succeeds. Manually devising such test code would be particularly tedious since for every basic precondition of every method one would have to create ontology contexts that let a method fail and continue to execute. For the 91 formally specified methods of the CHIL OWL API 176 basic preconditions were defined. Still, only 18 different preconditions as given in Appendix F were needed to describe all the properties of an ontology under which the CHIL OWL API methods execute properly. This 18:176 ratio was reason enough to automate the generation of test cases as depicted in Fig. 4.12.

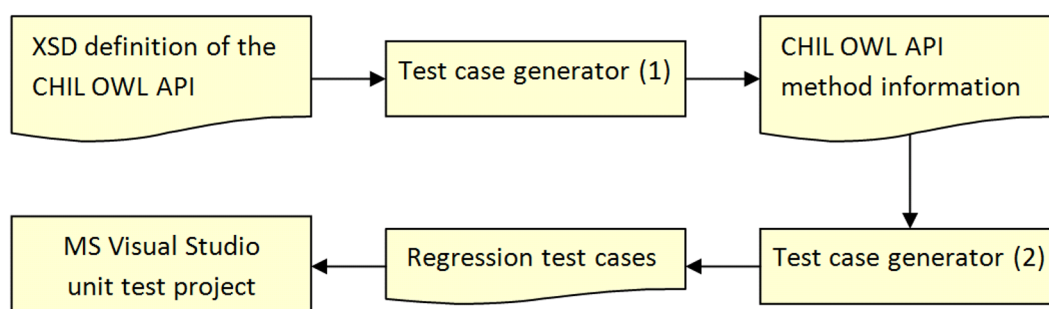


Figure 4.12: CHIL OWL API test case generation

The specification of the CHIL OWL API and the protocol information of the 18 basic preconditions were used to provide a test case generator application with sufficient information to generate regression test code, which utilizes the CHIL OWL API C# client library. The test generator code for the *listObjectPropertyValuesOfIndividual(o, R)* method is shown below followed by an excerpt of the generated regression test code.

```

1 internal static Protocol listObjectPropertyValuesOfIndividual() {
2     Method m = new Method();
3     m.Interface = "IAskingABox";
4     m.Name = "listObjectPropertyValuesOfIndividual";
5     m.ResultInterpretation = "listOfIndividuals";
6     FormalParameterIndividual owlIndividual =           ↪
7     new FormalParameterIndividual("owlIndividual");
8     m.FormalParameters.Add(owlIndividual);
9     FormalParameterObjectProperty owlObjectProperty = ↪
10    new FormalParameterObjectProperty("owlObjectProperty");
  
```

```

11 m.FormalParameters.Add(owlObjectProperty);
12 m.Preconditions.Add(new DeclaredIndividual(owlIndividual, m, false));
13 m.Preconditions.Add(new DeclaredObjectProperty(owlObjectProperty,      ↵
14                                     m, false));
15 m.Preconditions.Add(new HasObjectProperty(owlIndividual,                ↵
16                                     owlObjectProperty, m, true));
17 return new Protocol(m);
18 }

```

```

1 [TestMethod]
2 [Description("IAskingABox")]
3 public void ListObjectPropertyValuesOfIndividual() {
4     object obj = null;
5     // Precondition: Declared individual 'owlIndividual'
6     try {
7         obj = o.ListObjectPropertyValuesOfIndividual(                        ↵
8                                     "#UndefinedIndividual",                ↵
9                                     "#UndefinedObjectProperty");
10     Assert.Fail("Expected 'UndeclaredIndividualException'");
11 }
12 catch (UndeclaredIndividualException) {
13     o.DeclareClass("#A");
14     o.AddIndividual("#oa", "#A");
15     try {
16         obj = o.ListObjectPropertyValuesOfIndividual(                        ↵
17                                     "#oa",                                  ↵
18                                     "#UndefinedObjectProperty");
19     } catch {}
20 }
21 catch (Exception ex2) {
22     Assert.Fail(ex2.Message);
23 }
24 [...]
25 }

```

The generated test cases are executed by the regression test framework of Microsoft Visual Studio [Mic07c]. The CHIL OWL API adapter code and the Jena Semantic Web Framework [HP 04] were instrumented using the code coverage tool Emma [Rou07]. In the adapter component, the measured method-, block-, and line coverage based on the purely automatically generated test code was 100%, 88%, and 86%, respectively. For the ten mostly used Jena Semantic Web Framework 2.5.5 Java packages a class-, method-, block-, and line coverage of 65%, 42%, 43%, and 44%, respectively, could be achieved. For the five mostly used packages, the class-, method-, block-, and line coverage was 69%, 47%, 53%, and 55%, respectively. The code coverage of the Jena system was several percentage points better for previous releases of the Jena Semantic Web Framework.

The automatically generated test code puts emphasis on the behavior of the CHIL OWL API implementation with respect to the defined preconditions for each method. This is why it is conceivable that even though a method passes the automatically generated test cases, it may fail under certain circumstances. In particular, there is no explicit testing yet of the results of ontological reasoning. For example, for the *list ObjectProperty Values-Of Individual( $o$ ,  $R$ )* method it can be imagined that there is a sub-property of  $R$ , which relates  $o$  to further individuals. In order to cover test cases that depend on such inferred entailments in the ontology (i.e. proper reasoning in the adapted ontology management system), a regression test framework as depicted in Fig. 4.13 was used. Together with a reference ontology, the formal specification of the CHIL OWL API was taken to manually compute the result sets and side effects of each interface method. The same kind of output is computed by the adapted off-the-shelf ontology management system. Both results are automatically compared with each other in order to validate the implementation.

The design of the reference ontology and the set of test cases were crucial in order to validate the CHIL OWL API as completely as possible. The reference ontology had to comprise both a TBox and an ABox. Moreover, special effort was put in covering ontology features that are specific to the *SHOIN(D)* Description Logic on which the Web Ontology Language OWL DL is based (e.g., transitive roles, transitivity of the

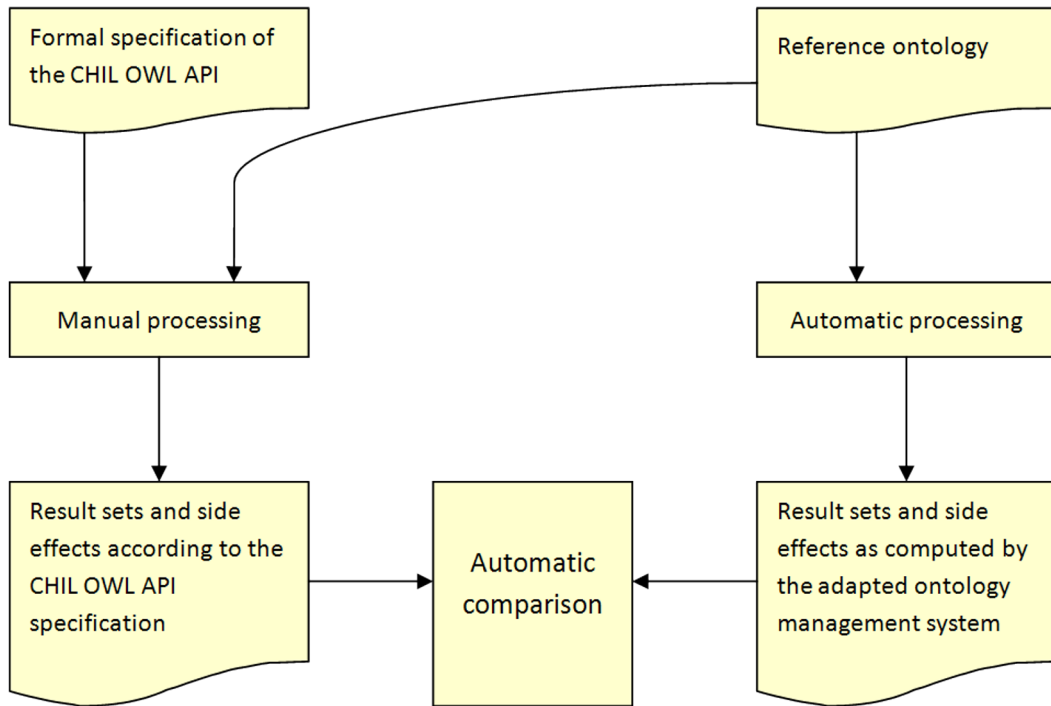


Figure 4.13: CHIL OWL API testing framework

subsumption relation, and nominals). Test cases were chosen in a similar way such that, with a presumably minimal number of method calls, a maximum of reasoning and ontology management features could be covered. For example, for a TBox  $C_1 \sqsubseteq C_2 \sqsubseteq C_3$  the method  $listSubClasses(C)$  is called with both concepts  $C_2$  and  $C_3$  in order to be able to check for the indirectly subsumed subclass  $C_1$  as well.

The combination of the automatically generated regression test cases with 98 manually devised test cases increased the block- and line coverage of the adapter code by eight and six percentage points, respectively. The code coverage of the Jena Semantic Web Framework did not change significantly, which indicates the applicability of the automatic test case generation approach. However, the manually devised test cases explicitly check for compliance with particular OWL DL language features, which may make it easier to adapt further ontology management systems to the CHIL OWL API. Note that the presented testing framework is not to validate the behavior of off-the-shelf ontology management

systems against the specification of the Web Ontology Language. Rather, adapters of such systems are tested to comply with the specification of the CHIL OWL API with respect to the given reference ontology.

### 4.3 Example Scenario Implementation

Using the CHIL OWL API, knowledge base queries and modifications as they occur in the example scenario described in Subsection 1.3.7 can be implemented. The project meeting of the example scenario can be scheduled and the set starting times queried in C# as follows (object *o* refers to an instance of the *ICHILOWLAPI* interface).

```
1 string ns = "http://chil.server.de/ontology#";
2 string xs = "http://www.w3.org/2001/XMLSchema#";
3 o.AddDatatypePropertyValue(ns + "PROJECTMEETING",           ↪
4   ns + "scheduledAt", "2008-06-27T13:00:00Z", xs + "dateTime");
5 string[] startingTimes = o.ListDatatypePropertyValuesOfIndividual( ↪
6   ns + "PROJECTMEETING", ns + "scheduledAt");
```

The CHIL Knowledge Base Server ships with a full-fledged XSD validator. Still, using an API to modify external data is inherently unsafe and error-prone. For the code snippet above a conventional C# compiler does not 1) check that the referenced OWL datatype property actually exists in the ontology and 2) type-check that the string literal in line 4 denotes a valid *xsd#dateTime* value. Hence, a well typed C# program may be invalid with respect to a particular ontology and XML schema. In line 5, the burden to parse the returned *xsd#dateTime* string is put on the programmer.

In programming languages such as C#, the *is*-operator can be used to check the runtime type of an object. For ontological individuals such a runtime type check has to be devised manually as shown below, where the list of RDF types of the individual under consideration is checked to include the required type. Again, undetected syntax errors may lead to program failure at runtime.

```

1 string ns = "http://chil.server.de/ontology#";
2 string [] arrTypes = ↔
3   o.ListRDFTypesOfIndividual(ns + "PROJECTMEETING");
4 if (arrTypes.Contains(ns + "ActiveMeeting")) { [...] }

```

In statically typed programming languages, the structure of input objects that can substitute a formal method parameter can be restricted by its declared type. This feature is not applicable for external types since there is no isomorphic mapping from ontological concept descriptions and XSD type definitions to C# classes. As shown below, a method that returns the start time of an example scenario *Event* object can only be declared with a string parameter that references the individual in the knowledge base. There is, however, no guarantee that at runtime the referenced individual actually is in the extension of the *Event* concept. A manual “type check” as shown above would be necessary.

```

1 public int getStartTime(string Event) { [...] }

```

The author developed the Zhi# programming language on request by CHIL project partners in order to make OWL concept descriptions and XSD type definitions first class citizens of an object-oriented programming language.

## 4.4 Related Work

This section gives an overview of some widely used ontology management systems and reasoning engines along with a bird’s eye view of some generic interface specifications for DL systems and related attempts to improve the remoting capabilities of such APIs. Available ontology management systems at the time the CHIL research project started were KAON, the Jena Semantic Web Framework, Snobase, the Protégé knowledge acquisition system, Sesame, and RACER. Available Description Logic APIs at this time include the DIG protocol and former frame-oriented knowledge representation systems such as the Generic Frame Protocol and OKBC.



#### 4.4.1 Off-the-shelf ontology management systems

This subsection gives an overview of existing off-the-shelf ontology management systems that can be used to manage RDF(S) data. In particular, the following three dimensions are considered. Firstly, remoting capabilities are assessed based on the number of supported remoting protocols (e.g., Java RMI [Sun06a], SOAP [Mit07], and CORBA [Obj09c]). Secondly, the extent of native support for programming languages such as Java or C++ is considered. Thirdly, the way how the API specifications were devised is listed.

Table 4.5: Off-the-shelf ontology management systems

Ont. mgt. system	Remoting support	Prog. languages	API specification
KAON2	limited to Java	Java	Java Docs
Jena	no	Java	Java Docs
Snobase	no	Java	Java Docs
Protégé	no	Java	Java Docs
Sesame	HTTP	Java, Python	Java Docs
RACER	HTTP, Sockets	Java, Lisp	DIG

KAON2 [Mot06] is an open source ontology management infrastructure targeted for business applications. It ships as a Java library file. The core of the KAON2 Java library are two APIs for RDF and the KAON ontology language. These APIs are represented by Java interfaces for which several implementations exist. Remote access is supported but limited to the Java programming language. Moreover, additional application server software is required, which may be impractical in practice. KAON2 comes with natural language descriptions of its RDF and KAON ontology language APIs.

HP Labs' Jena Semantic Web Framework [HP 04] provides an ontological framework for the Java language environment. The internal representation of ontological data in Jena is tightly bound to the RDF model of triples. Originally designed for DAML+OIL, but later adopted to OWL, Jena ships with a layered API. On the upper layers, it offers

a unified view onto the features of the DAML+OIL and OWL languages, while providing access to specific ontology language dependent constructs via specific ontology models. Jena makes it possible to configure and to replace the underlying reasoning engine. This is why there are only informal specifications of the exposed Jena API since the actual behavior depends on the used reasoner. Remote access is not supported.

The IBM Ontology Management System (also known as SNOBASE, for Semantic Network Ontology Base) [LGA03] is a Java framework that provides a mechanism for querying ontologies and a programming interface for interacting with vocabularies of standard ontology specification languages such as OWL. Applications can query against the created ontology models and the inference engine deduces the answers and returns result sets similar to JDBC (Java Data Base Connectivity) result sets [Sun06b]. In theory, Java-based remote access is possible but not available yet.

Stanford University School of Medicine's Protégé [Sta06] is an interactive Java application with a GUI that focuses on creating and editing ontologies. Having started as a project before OWL was available, it was designed to support a variety of different ontology languages. For ontology management Protégé focuses on user input through the graphical user interface. It also supports an API for plug-ins that essentially can be used as a management API. However, there is no support for remote access and the Protégé OWL API is specified only by example.

Sesame [Ope06] is an open source RDF database with support for RDF(S) inferencing and querying. It can be deployed on top of a variety of storage systems and offers a significant number of wrappers that facilitate HTTP-based access to the Sesame system for a number of programming languages. However, the semantics of the Sesame API for processing OWL data are defined by third party extensions, which up to now only implement fragments of the OWL specification.

RACER [MH04b] is a Semantic Web inference engine for query answering over RDF documents, and, with respect to specified RDF(S)/DAML ontologies, registering perma-

ment queries. RACER implements a Description Logics reasoning system with support for TBoxes with generalized concept inclusions, ABoxes, and concrete domains. It supports native access from Java and Lisp and implements the DIG protocol [Bec02] via XML-over-HTTP.

#### 4.4.2 Knowledge base interface specifications

The DIG protocol [Bec02], which is a simple API for a general Description Logics system, is one representative of a class of interface definitions that consist of simple mechanisms to tell and ask DL knowledge bases. These mechanisms follow foundational aspects that have been well-studied over time [Lev84]. Many previous frame-oriented knowledge representation systems such as the Generic Frame Protocol [CFF97] and OKBC (Open Knowledge Base Connectivity) [CFF98] also embody such distinctions.

Although well defined, the DIG specification merely defines an XML schema that has to be used along with HTTP as the underlying communication protocol. There is no specific support for a particular programming language. In contrast, the KRSS specification [PS93], which is an earlier approach to define a number of DL tell and ask operations, was tightly bound to the LISP [Gra95] syntax, which may not be adequate for programmers who prefer other languages such as Java or C#.

In addition to RACER, the FaCT reasoner [Hor98, Hor99] from the University of Manchester is another implementation of the DIG 1.0 interface specification, which also requires further application server software to facilitate remote access.

Bechhofer et al. proposed a CORBA interface to the FaCT system [BHP99]. Beyond the fact that CORBA may not be an appropriate remoting technology in today's service-oriented and XML-based computing environments, Bechhofer et al. note that "the CORBA IDL does not support the definition of the kinds of recursive data types that may be required for the representation of DL concepts and roles". This is why an XML-based workaround was devised to pass ontological concepts and roles as single data items.

Previous approaches to augment DL knowledge base interfaces with remoting capabilities include the wines [BMP91] and stereo [MRJ95] configuration demonstration systems.

Common to all mentioned DL knowledge base interface specifications is the lack of support for arbitrary state-of-the-art remoting protocols and adequate error and exception handling. In particular, there are no detailed error messages presented to knowledge base clients in case invalid requests are passed to the ontology management system. Furthermore, none of the discussed API definitions is formally specified in the sense of a foundation on formal Description Logic semantics. Their partially unintelligible specifications make it particularly difficult to develop applications and to interchange the underlying OWL API. Moreover, support for XML Schema Definition data types is throughout very limited.

## 4.5 Summary

The author devised a pluggable architectural model of an ontological knowledge base server. The CHIL Knowledge Base Server can be used to adapt off-the-shelf ontology management systems. In the current implementation, the Jena Semantic Web Framework [HP 04] is adapted and configured to use the Pellet OWL DL reasoner [Pel06] with in-memory and database backed ontology models. The CHIL Knowledge Base Server can be started as a standalone application and as an Eclipse plug-in as shown in Fig. 4.2 and 4.3, respectively. The Eclipse plug-in provides a GUI that can be used to control the CHIL Knowledge Base Server and to browse managed ontologies.

The CHIL Knowledge Base Server implements the formally specified CHIL OWL API for  $\mathcal{SHOIN}(\mathbf{D})$  knowledge bases. The CHIL OWL API was defined based on a combination of Floyd-Hoare logic and formal Description Logics terminology. The formal specification was devised in order to make it possible to consistently adapt off-the-shelf ontology management systems and to provide knowledge base clients with well defined programming language independent semantics of the CHIL OWL API.

Complementary to the formal specification, the CHIL OWL API definition is given as an XML instance document. Its XML Schema Definition is twofold. The first part as depicted in Fig. 4.5 provides a schema how interfaces and methods can be defined on the meta-level. The second part as shown in Fig. 4.6 defines the format of request and response messages on the object level (i.e. the format of the messages that are sent over the wire). Both the XML-over-TCP server component of the CHIL Knowledge Base Server as well as the client libraries were generated automatically from the XML-based API definition. The code generation approach helped a great deal during the development phase of the CHIL OWL API. Client libraries could be automatically updated to instantaneously reflect changes in the API definition. Regression test code was automatically generated from the formal specification. The CHIL OWL API adapter code and the Jena Semantic Web Framework were instrumented using the code coverage tool Emma [Rou07]. In the adapter component, the measured method-, block-, and line coverage based on the purely automatically generated test code was 100%, 88%, and 86%, respectively. For the ten mostly used Jena Semantic Web Framework Java packages a class-, method-, block-, and line coverage of 65% (68%), 42% (51%), 43% (48%), and 44% (49%), respectively, could be achieved. For the five mostly used packages, the class-, method-, block-, and line coverage was 69% (75%), 47% (58%), 53% (57%), and 55% (55%), respectively. Numbers in brackets were effective for a previous Jena release.

The formally specified CHIL OWL API facilitates access to the CHIL Knowledge Base Server from a variety of heterogeneous client applications. A case study [PRS09], conducted in course of the CHIL research project, showed that in order to benefit from common conceptualizations as defined by OWL DL ontologies, it is crucial to improve the connectivity of ontology management systems in order to fully exploit their potential application scope and to support a variety of different programming languages. The CHIL Knowledge Base Server proved to be a reliable back-end for a semantic middleware that incorporates more than fifty image and speech recognition based perceptual components used in the CHIL research project.



## CHAPTER 5

### XSD Aware Compilation – Types and... Constraints

Since their standardizations by the W3C, the Extensible Markup Language (XML) [BPS06] and XML Schema Definition (XSD) [FW04] have been widely adopted as a format to describe data and to define programming language agnostic data types and content models. Several other W3C standards such as the Resource Description Framework (RDF) [MM04] and the Web Ontology Language (OWL) [MH04a] are based on XML and XSD. In RDF, it is possible to have typed literals whose types are declared in an XML schema definition. Processing OWL data implicitly requires the use of constrained atomic XML Schema Definition data types, which may be the range of OWL datatype properties. For example, an ontological concept *Person* may have defined a property *hasAge* of type *xsd#unsignedInt*. This type could be mapped to the C# data type *System.UInt32*. If, however, the XML schema defined a further constrained atomic data type such as *unsignedIntLessThan110* in order to constrain possible *hasAge* values of a *Person* to reasonable values less than 110, there would be no appropriate C# data type with a value space that comprises integer values between 0 and 110. Instead, assignments to objects of type *System.UInt32* would have to be explicitly checked to be *schema valid*.

An XML instance document – or in this case an instance of a constrained atomic data type – is said to be a valid instance of a schema if there is an XML schema given and the content of the XML instance document – or of the data type value – conforms to the content model as defined in the schema. Up to now, schema validation has been particularly error-prone since there is no isomorphic mapping between XML schema definitions and programmatic data types.

The objective of this work is to incorporate XML Schema Definition data types into a conventional object-oriented programming language and to automate tedious validation and type checking tasks. Except for conjoint type derivation along with different type systems, it should be possible to use XML data types and values of these types in any context of Zhi# programs that is valid for built-in .NET value types. In particular, programming language features such as method overriding, user-defined operators, and runtime type checks should abide by XML Schema Definition type system rules. In the Zhi# programming language, the constrained types of the  $\lambda_C$ -calculus are used in order to implement programming language inherent support for constrained atomic XML Schema Definition data types. In this chapter, the term “constrained types” refers to atomic data types that represent a value space, which may be constrained by explicitly defined constraining facets (e.g., *xsd:minExclusive*, *xsd:maxExclusive*) as described in Chapter 3. Note that this is different to constraint-based type inference algorithms found in the literature where constraints are not checked but rather recorded for later consideration.

The outline of this chapter is as follows. Section 5.1 describes the application of  $\lambda_C$ -calculus type system features in order to embed atomic XML Schema Definition data types with the C# programming language. Section 5.2 alludes to related work in the field of domain specific languages for XML. A summary is given in Section 5.3.

## 5.1 Integrating XSD with the C# Programming Language

The XML Schema Definition type system was implemented based on the formal foundation of the  $\lambda_C$ -calculus as described in Chapter 3. This implementation of XML Schema Definition data types was integrated with the Zhi# programming language by means of a plug-in for the Zhi# compiler framework that facilitates loading of XML schema definitions, static typing of constrained atomic types in Zhi# programs, and the transformation of XSD type references in Zhi# programs into C# code. Dynamic checking of XML data types is facilitated by the accompanying XSD plug-in for the Zhi# runtime library.



### 5.1.1 Referencing XML Schema Definitions

The Zhi# programming language introduces a feature henceforth called *XSD aware compilation*. The Zhi# compiler takes as input both Zhi# source files as well as XML schema definitions as depicted in Fig. 5.1.

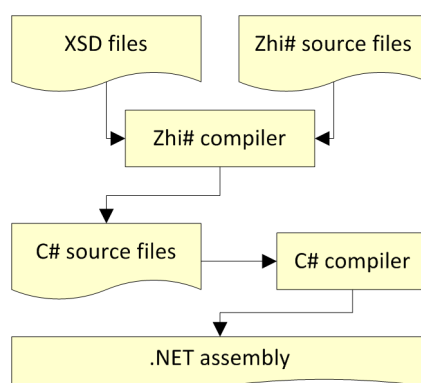


Figure 5.1: XSD aware compilation

Atomic XML data types defined in XML schemas are made public as native data types in Zhi# programs using the keyword *import*, which works analogously for XML data types like the C# *using* keyword for .NET programming language type definitions. It permits the use of atomic XML data types in a Zhi# namespace such that, one does not have to qualify the use of a type in that namespace. Thus, the *productID* data type, which is defined in the XML namespace *http://www.chokycola.com* in the following schema, can be used in a Zhi# program as shown below.

```
1 <xsd:schema targetNamespace="http://www.chokycola.com" [...] >
2 <xsd:simpleType name="productID">
3   <xsd:restriction base="xsd:int">
4     <xsd:minInclusive value="4000"/>
5     <xsd:maxExclusive value="8000"/>
6   </xsd:restriction >
7 </xsd:simpleType>
8 </xsd:schema>
```

```

1 // Import XML namespace 'http://www.chokycola.com' and bind it to alias 'coke'
2 import XML coke = http://www.chokycola.com;
3 namespace MyBusinessApplication {
4   class MyClass {
5     #coke#productID pID;
6   }}

```

In the given example, the value space of the XML data type *productID* is limited to integer values greater than or equal to 4000 and less than 8000.

For each imported XML namespace (e.g., *http://www.chokycola.com* in line 2 in the code snippet above) the Zhi# XSD compiler plug-in loads the respective type definitions from the referenced library files. The set of library files that is made available to each Zhi# compiler plug-in comprises all referenced non-source code files (e.g., .xsd-, .owl-, .dll files). From these files XML Schema Definitions (i.e. .xsd-files) are considered by the XSD compiler plug-in. An error is raised if in the referenced schema files no XSD type definitions exist in the imported namespaces. Types that are defined in the imported XSD namespaces may be used in the code following the *import* statement; they are syntactically highlighted in the Eclipse-based Zhi# editor and are available in form of autocompletion proposals as shown in Fig. 5.2. Imported XSD types are subject to concise static typing.

In the current implementation of the Zhi# compiler, external type references must be fully qualified using an alias that is bound to the namespace in which the external type is defined. In particular, it is not possible to 1) use external namespace references at arbitrary positions in Zhi# programs and 2) use unqualified external type references.

The decision to not support external namespace references at arbitrary positions is due to the fact that different external type systems may use arbitrary schemes to denote a namespace. However, without syntactical restrictions that are known beforehand it is not feasible to allow for such arbitrary naming schemes in the Zhi# language grammar. Even for XSD and OWL namespaces, which follow the generic syntax rules for URIs

[BFM98], it may not be desirable to repeatedly use lengthy namespaces instead of handy aliases. Both the external namespace alias and the local type name must be preceded by a '#'-symbol (i.e. type references must be of the form `#alias#local_name`, see line 5 in the Zhi# code snippet above and line 15 in Fig. 5.2 below).

```

5 import XML xsd = http://www.w3.org/2001/XMLSchema;
6 import XML coke = http://www.choky-cola.com/business;
7 import OWL ont = http://www.choky-cola.com/ontology;
8
9 namespace ProgramNamespace
10 {
11     public class Demo1
12     {
13         public void f ()
14         {
15             #coke#
16         }
17     }
18 }

```

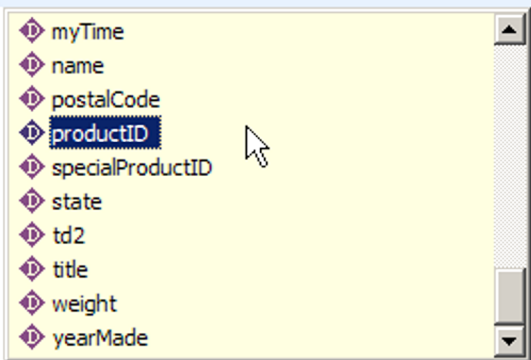


Figure 5.2: Autocompletion of XSD types

### 5.1.2 Static typing

The XSD compiler plug-in provides exhaustive static type checking of XML Schema Definition type references. At compile time, the types of XSD objects are checked according to the constraint-based (sub-)typing rules of the  $\lambda_C$ -calculus (see Chapter 3). The generated C# code eventually contains dynamic type checks in order to allow for a safe usage of Zhi# assemblies from conventional .NET programs. The dynamic checking code for XML data types is provided by the XSD plug-in for the Zhi# runtime library.

XML Schema Definition incorporates features from both nominal and structural type systems. XML data types are given names and are explicitly derived from a named base type. Each type eventually refines one of the 19 built-in primitive base types. Still, a subtype relation between two XML data types exists not only based on the explicitly defined derivation tree. Instead, a type  $S$  can also be considered to be a subtype of  $T$  based on a more specific value space. Type  $S$  is a subtype of  $T$  if the set of schema valid elements for type  $S$  is a subset of the set of schema valid elements for type  $T$ .

In the following schema definition, the XML data types *lessThan100* and *lessThan10* are explicitly derived from the built-in primitive type *xs#integer*. Accordingly, there is no explicit subtype relation between *lessThan10* and *lessThan100*.

```

1 <xsd:schema xmlns:xsd="...">
2   <xs:simpleType name="lessThan100">
3     <xs:restriction base="xs:integer">
4       <xs:maxExclusive value="100"/>
5     </xs:restriction>
6   </xs:simpleType>
7   <xs:simpleType name="lessThan10">
8     <xs:restriction base="xs:integer">
9       <xs:maxExclusive value="10"/>
10    </xs:restriction>
11  </xs:simpleType>
12 </xsd:schema>

```

The classified type hierarchy as shown in Fig. 5.3 is revealed by the subtyping rules of the  $\lambda_C$ -calculus. The dashed arrow indicates the implicit subsumption of type *lessThan10* by *lessThan100*. The Zhi# type checker always uses the classified derivation tree that includes both explicit and implicit “is-a” relationships between XSD type definitions. Note that throughout this work only atomic XML data types are considered while there is no support for complex XML Schema Definition content models.

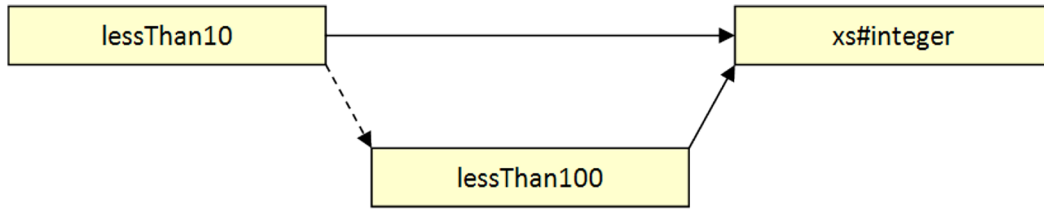


Figure 5.3: Implicit subtype relationship between XSD types

The Common Language Specification [Mic06] defines several value types for which isomorphic mappings exist to XML Schema Definition types as shown in Table 5.1.

Table 5.1: Isomorphic mappings between .NET and XSD types

.NET value type	XSD type	.NET value type	XSD type
<i>System.String</i>	<i>xsd#string</i>	<i>System.Int16</i>	<i>xsd#short</i>
<i>System.Boolean</i>	<i>xsd#boolean</i>	<i>System.UInt16</i>	<i>xsd#unsignedShort</i>
<i>System.Single</i>	<i>xsd#float</i>	<i>System.Int32</i>	<i>xsd#int</i>
<i>System.Double</i>	<i>xsd#double</i>	<i>System.UInt32</i>	<i>xsd#unsignedInt</i>
<i>System.SByte</i>	<i>xsd#byte</i>	<i>System.Int64</i>	<i>xsd#long</i>
<i>System.Byte</i>	<i>xsd#unsignedByte</i>	<i>System.UInt64</i>	<i>xsd#unsignedLong</i>

In spite of the obvious subsumption relationships between several of these types, there is no explicit subtype relation between them (e.g., *System.Int16* is not derived from *System.Int32*). Since there is no structural typing in C# 1.0 (except for some questionable subtleties) one cannot infer implicit compatibilities between these types. Instead, the C# programming language defines a number of implicit conversion operators between the built-in value types. These operators are not defined in the mscorlib.dll but are provided extra by the C# compiler. Similarly, the Zhi# compiler allows for implicit conversions between the built-in .NET value types listed in Table 5.1 and XML data types for which a value space based subtype relation exists. For example, an *xsd#int* value can be assigned to a *System.Int32* variable and vice versa. The following subsections will elaborate on the use of XML data types in Zhi# programs, which is illustrated in Fig. 5.4.

```

1 import XML cc = http://www.chokycola.com/business;
2 class Product {
3   public static implicit operator #cc#itemPrice(Product p) { [...] }
4 }
5 class BillingApp {
6   public int getPostageOfSpecialProducts(#cc#specialProductID pID) { [...] }
7   public #cc#shippingAndHandling getPostage(#cc#productID pID) {
8     if (pID is #cc#specialProductID) {
9       int i = getPostageOfSpecialProducts(pID);
10      if ((i > 0) && (i < 20)) {
11        return i;
12      }
13      else {
14        throw Exception("Invalid postage value");
15      }
16    }
17    else {
18      return 7.0F;
19    }
20  }
21  public void prepareInvoice(Product p, #cc#shippingAndHandling SaH) {
22    #cc#itemPrice price = p;
23    #cc#invoiceTotal = price + SaH;
24    [...]
25  }
26 }
27 class DerivedBillingApp : BillingApp {
28   public override #cc#shippingAndHandling
29     getPostage(#cc#extendedProductID pID) { [...] }
30 }

```

Figure 5.4: XML data types in Zhi#programs

### 5.1.2.1 Method overriding

Method overriding allows the new implementation of a method inherited from a base class. The implementation in the subclass replaces the implementation in the superclass. It is resolved at runtime which of these methods is used. Different programming languages use different variance rules for the return types and formal parameter types of method declarations involved in overriding. Assuming the following method declarations, the *covariance rule* for return types stipulates that  $B$  must be a descendant type of  $A$ ; the *contravariant rule* for formal parameters imposes that  $P$  must be a descendant type of  $Q$ .

```
1 class T {
2   public A f(P p) { [...] }
3 }
4 class S : T {
5   public B f(Q q) { [...] }
6 }
```

In C# 1.0, both the return types as well as the formal parameter types of methods involved in method overriding must be *invariant* (i.e. identical). In Zhi#, methods can be declared using XSD input and output parameters. The Zhi# compiler provides for covariant XSD return types and contravariant XSD formal parameter types as follows.

```
1 class T {
2   public #xsd#long f(#xsd#byte p) { [...] }
3 }
4 class S : T {
5   public #xsd#int f(#xsd#short q) { [...] }
6 }
```

This change to the standard behavior of C# 1.0 was made since covariant output parameter types and contravariant input parameter types are generally safe and recent versions of Java and C# are no longer restricted to invariant parameter types either.

The *getPostage* Method in line 7 in the *Zhi#* example program shown in Fig. 5.4 yields the shipping and handling costs that apply for particular products, which are distinguished by their product ID. The input parameter of the overriding *getPostage* method method in line 28 must be at least as general as the original parameter type *productID*, which is true for the used *extendedProductID* type.

### 5.1.2.2 Method overloading

Method overloading is a type of polymorphism usually found in statically-typed programming languages that allows the declaration of a number of methods that share the same name but have different method signatures. It is resolved at compile time which of these methods is used.

In *Zhi#*, methods must not be overloaded based on XML data types. This current restriction is a result of the compilation of XML data types to *C#* code where the single proxy type *RTSimpleType* (see Section 3.6) is substituted for all XML data types that are used in *Zhi#* programs. This restriction is similar to method and operator overloading using generic types in Java and is inherent to programming language features compared to features of the runtime environment. The *Zhi#* compiler reports an error for the following method definitions. Still, XML data types may occur in signatures of overloaded methods where overloading stems from conventional formal parameter types as shown below.

```
1 class C {
2     public void f(#xsd#short p) { [...] }
3     public void f(#xsd#byte p) { [...] } // Error!
4 }
```

```
1 class C {
2     public void f(#xsd#short p, short q) { [...] }
3     public void f(#xsd#byte p, byte q) { [...] } // OK!
4 }
```



### 5.1.2.3 *is*-Operator

The *is*-operator is used to check whether the runtime type of an object is compatible with a given type. For constrained value types this corresponds to checking whether the *value* of an object is an element of the *value space* of the given type. For the scope within the checking statement the type of the given object is inferred to be at least as specific as the given type. In line 8 in the Zhi# program shown in Fig. 5.4, the *is*-operator is used to check whether the *productID* value *pID* is in the value space of type *specialProductID*. According to the  $\lambda_C$ -subtyping rules, both types do not necessarily need to be in the same derivation chain. A compile-time error occurs if both types are derived from two different (i.e. incompatible) primitive base types in order to avoid unnecessary type checks at runtime. Runtime type checks using the *is*-operator adhere to changes in the schema definition. The type check in line 8 will still function properly even if the type definition of *specialProductID* changes after the compilation of the Zhi# program. However, the type definition must not change during the execution of a compiled Zhi# program since XML schema files are loaded and parsed only once at application startup.

### 5.1.2.4 User-defined operators

In line 22 in the Zhi# program shown in Fig. 5.4, an instance of class *Product* is assigned to an *itemPrice* variable. This assignment utilizes the user-defined implicit conversion operator from *Products* to *itemPrices* that is defined in line 3.

In Zhi#, user-defined conversion operators and user-defined binary operators may contain XSD types as long as one type in the parameter list is the containing .NET type, which is a limitation imposed by the C# language specification (i.e. Zhi# does not restrict the set of definable operators). Note that in the output C# code all XML data types are represented by the single proxy type *RTSimpleType*. As a consequence, it is not possible yet to define a number of operators whose signatures vary only in the used XML data types at the same positions in the signatures (cf. method overloading).

User-defined operators that relate XSD types are complementary to Zhi#'s built-in operators for XSD types since the built-in operators do only relate XSD types with XSD types and .NET primitive types (XSD type definitions cannot contain operator definitions; .NET primitive type definitions are *sealed* (i.e. cannot be inherited) and cannot contain additional operator definitions either).

### 5.1.2.5 Binary operators

Zhi#'s XSD compiler plug-in (i.e. the implementation of the  $\lambda_C$ -calculus for XML Schema Definition) defines a number of built-in binary operators for XML data types as listed in Table E.1. Instances of XML data types can be compared with instances of compatible types using the comparison operators '>=', '>', '==', '<=', and '<' and combined through addition, subtraction, multiplication, division, modulo, bitwise AND, OR and XOR, and logical AND and OR using the '+', '-', '\*', '/', '%', '&', '|', '^', '&&', and '||' operators, respectively. The comparison operators return a .NET *bool* value instead of an *xsd#boolean* in order to facilitate their usage in, for example, *if*-statements.

### 5.1.2.6 Conversion operators

The Zhi# XSD compiler plug-in defines a set of conversion operators for all XSD and primitive .NET value types that are compatible with each other according to the subtyping rules of the  $\lambda_C$ -type system. Compatible .NET value types such as *System.Int32* ( $\mapsto$  *xsd#int*) or *System.Double* ( $\mapsto$  *xsd#double*) are implicitly converted to their XSD counterparts and may thus be used conjointly with XML data types. Also, XML data type values can be assigned to .NET *string* variables. The built-in implicit *ToString*-conversion operators serialize the values of XSD objects to .NET *strings*. Moreover, in Zhi#, the XSD type *xsd#boolean* implements the .NET *true* and *false* operators, which facilitates the use of *xsd#boolean* objects in controlling expressions in, for example, *if*-statements and in conditional expressions.

### 5.1.2.7 Assignments to instances of XSD array types

The Zhi# programming language extensions were devised to most closely adhere to the standard behavior of conventional C#. The most remarkable deviation pertains to the typing rule of arrays. Interestingly, C# (like Java) actually permits covariant subtyping of arrays. According to [Pie02], this feature was “originally introduced [in Java] to compensate for the lack of parametric polymorphism in the typing of some basic operations such as copying parts of arrays”. In C#, the following two lines are well-typed. However, the assignment in line 2 causes an *ArrayTypeMismatchException* at runtime.

```
1 object [] arr = new string [1]; // OK for C# compiler
2 arr [0] = 23; // ArrayTypeMismatchException at runtime
```

Nowadays, covariant subtyping of arrays is generally considered a flaw in the language design and was decidedly modified for constrained types in Zhi#. The XSD compiler plug-in enforces invariant subtyping for arrays of XSD types (it does allow covariant subtyping for “references”, though, see below). For arrays of constrained types, the XSD plug-in emits a compile-time error if the target and source array type are different.

```
#xsd#int [] arr = new #xsd#byte [1]; // Rejected by Zhi# compiler
```

### 5.1.2.8 Assignments to instances of XSD non-array types

Assignments to instances of constrained non-array types would implicate the same issues as for array types since Zhi#'s constrained value types are in fact represented by .NET reference types in the generated C# code (cf. Fig. 3.2). As a consequence, if assignment expressions that comprise constrained types in Zhi# were straightforwardly translated into assignment statements of their corresponding C# proxy types, not only the *value* of an object but also its *type* (i.e. the constraints) would be assigned to the target object. That would lead to incorrect program behavior in the presence of the desired covariant subtyping for constrained types since for subsequent assignments the target object may

be guarded by too restrictive constraints of the previous source object. This is why declarations and assignments that involve constrained types are translated into object creation expressions and setter-function calls on the Zhi# runtime library, respectively. The following two lines of Zhi# code are translated into C# as shown below. For the sake of brevity, methods and types of the Zhi# runtime library are not fully qualified in code snippets in this chapter. Also, the namespace aliases from the input Zhi# program are used in the output C# code for XML namespaces.

```
1 #cc#productID pID = 4000;  
2 pID = 4001;  
  
1 RTSimpleType pID = NewXSD("cc#productID", 4000);  
2 SetValue("cc#productID", out pID, 4001);
```

The definition of variable *pID* is translated into a declaration of an instance of its abstract C# proxy type that is assigned the result of the invocation of the static *NewXSD* method, which creates an instance of the concrete *productID* proxy type (i.e. *RTDecimal*) and initializes it with the value 4000. The invocation of the static setter-function in line 2 in the C# program creates a new instance of the *productID* proxy type, initializes it with the value 4001, and assigns it to variable *pID*. The *out* parameter keyword causes the *SetValue* method to refer to the same variable that was passed into it, i.e. any changes made to *pID* in the method will be reflected in that variable when control passes back to the calling method. Thus, even assignments to *null*-objects are handled properly (at the cost of the creation of a fresh instance).

Assignments to instances of constrained XML data types are statically type checked by the Zhi# compiler. As a consequence, an assignment of a *System.Int32* value to a *productID* variable as shown in the code snippet below results in “(Line 3) Error: Violation of constraint xsd:minInclusive = 4000” and “(Line 3) Error: Violation of constraint xsd:maxExclusive = 8000” compile-time errors. In contrast, in plain C#, where one may use *System.Int32* or *System.Int16* integer types instead, the violation of the value space

constraints of the *productID* data type would result in error messages during costly schema validations at runtime. Even worse, invalid assignments may as well remain undiscovered and lead to application failure at runtime.

```
1 int i;  
2 #coke#productID pID;  
3 pID = i; // Rejected by Zhi# compiler
```

In Zhi#, for assignments to and from instances of constrained XML data types four different cases must be considered. Firstly, the type of the source operand may have a defined value space that is subsumed by the value space of the type of the target object. In addition to the  $\lambda_C$ -subtyping rules .NET data types such as *System.Int32* and *System.Double* are by definition compatible with their XSD counterparts (see Table 5.1 and Subsection 5.1.2.6). Such assignments are generally safe.

Secondly, the type of the source operand may have an appropriate value space that was inferred based on control and data flow analysis as described in the following subsection. Such assignments are safe, too.

Thirdly, an implicit user-defined conversion operator may be defined for assignments to and from instances of a constrained XML data type. In this case no Zhi# compiler warnings are emitted. Still, the conversion may fail in the implementation of the user-defined conversion operator. This may of course also happen with any conventional user-defined .NET conversion operator.

Eventually, source operands may be downcasted to the type of the target object. Unsafe downcasts (i.e. type casts to subtypes of the type of the casted expression) defer type checking from compile time to runtime (see Subsection 5.1.5). The Zhi# compiler emits warnings for value space constraints of the target type that are not known at compile time to hold for the value space of the source type and may thus cause the downcast to fail at runtime. Unsafe downcasts should therefore only be used rarely in cases where the type inference mechanisms of the XSD compiler plug-in are not sufficient.

### 5.1.3 Type inference

The Zhi# compiler framework supports the implementation of static type inference rules by external compiler plug-ins. The Zhi# compiler plug-in for XML Schema Definition infers types of variables based on control and data flow analysis. Types of literals are inferred based on the particular values of the literal expressions. Types of binary arithmetic expressions are inferred using the constraint arithmetic of the  $\lambda_C$ -type system as described in Section 3.7.

In the  $\lambda_C$ -calculus, the type inference rule TI-IFADD can be used to infer the value of a variable within the scope of an embracing *if*-statement. The rule TI-ASSIGNREM can be used to remove transiently added constraints from the type of a variable.

Table 5.2: TI-IFADD

$$\frac{\Gamma \vdash a : A \quad \text{if } \left( \bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i) \right) \text{ then } \diamond}{\Gamma_\diamond \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x \mid x \in v(A)\} \cap \{x \mid x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

Table 5.3: TI-ASSIGNREM

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : A \quad \Gamma_\diamond \vdash a : \bigcap_{i \in 1..n}^A c_i \quad a := t}{\Gamma_\infty \vdash a : A}$$

A scope within a program shall be denoted by  $\diamond$ , a sub-scope of  $\diamond$  shall be denoted by  $\diamond\diamond$ . Considering the *then*-branch of an *if*-statement of the form *if* ( $a \prec \text{literal}$ ) *then*  $\diamond$  it is safe to add the constraint  $c_{[\prec \text{literal}]}$  to the type of variable  $a$ . The constraint  $c_{[\prec \text{literal}]}$  holds for the instance  $a$  within scope  $\diamond$  until  $a$  is assigned a value (i.e. in a programming language with side effects  $a$  must also not be referenced by method invocations).  $\Gamma$  and  $\Gamma_\diamond$  are a typing context and a transient typing context for a limited scope  $\diamond$ , respectively (i.e.  $\Gamma_\infty$  is the typing context of the sub-scope  $\diamond\diamond$ ). Both type inference rules are implemented by Zhi#'s XML Schema Definition compiler plug-in.

In the following Zhi# program, the type of variable  $xpi$  is inferred to be  $xsd\#positiveInteger\{< 10\}$  for the assignment expression in line 3. Accordingly, the type of the lvalue  $x$  can be as specific as  $xsd\#positiveInteger\{< 10\}$ , too. The transiently added constraint  $\{< 10\}$  is removed from the type of  $xpi$  upon the assignment in line 4. Therefore,  $xpi$  is again of the explicitly declared type  $xsd\#positiveInteger$  for the assignment expression in line 5.

```

1 #xsd#positiveInteger xpi = [...];
2 if (xpi < 10) {
3   #xsd#int x = xpi; // xpi is an #xsd#positiveInteger{< 10}
4   xpi = [...];
5   #xsd#int xi = xpi; // xpi is an #xsd#positiveInteger
6 }

```

In the current implementation of the Zhi# compiler plug-in for constrained XML Schema Definition data types, only ANDed top-level expressions are used for type inference. In this way, the type inference algorithm is rather conservative (i.e. incomplete), which proved to be sufficient in practice since the effects of type inference should still be comprehensible for the programmer. For fundamental reasons, the proposed kind of type inference will always be incomplete since otherwise the inference algorithm would solve the Halting Problem (for the same reason, there is no “optimal” compiler).

In C#, it is possible to create loops by using the *for*, *while*, *do*, and *foreach*-statements. The *for* and *while*-loops execute a block of statements repeatedly until a specified expression evaluates to false. Because the test of the expression takes place before the execution of the loop (i.e. *for* and *while*-loops execute zero or more times) the loop termination criteria can be exploited for type inference. In contrast, a *do*-statement cannot be used for type inference since the loop body of a *do*-statement is executed at least once regardless of the value of the controlling expression. The *foreach*-statement repeats a group of embedded statements for each element in an array. These elements are not subject to any tests, which makes the *foreach*-statement useless for type inference.

In order to take advantage of the constraining effects of *for* and *while*-loops the two additional type inference rules TI-FORADD and TI-WHILEADD are implemented by the XSD compiler plug-in.

Table 5.4: TI-FORADD

$$\frac{\Gamma \vdash a : A \quad \text{for } ([initializers]; \bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i); [iterators]) \text{ then } \diamond}{\Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x | x \in v(A)\} \cap \{x | x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

Table 5.5: TI-WHILEADD

$$\frac{\Gamma \vdash a : A \quad \text{while } (\bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i)) \text{ then } \diamond}{\Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x | x \in v(A)\} \cap \{x | x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

All described type inference rules are applied to XML data type variables and variables of primitive .NET value types such as *System.Int32* for which an isomorphic mapping exists to an XSD type (see Table 5.1).

In the following Zhi# program the type of variable *i* is inferred to be *System.Int32*{< 10} for the assignment in line 3 using the type inference rule TI-FORADD. Similarly to the previous code snippet the transiently added constraint {< 10} is removed from the type of *i* upon the assignment in line 4 and *i* is again of the explicitly declared type *System.Int32* for the assignment expression in line 5.

```

1 int i = [...];
2 for (i = 0; i < 10; i++) {
3   #xsd#int x = i; // i is a System.Int32{< 10}
4   i = [...];
5   #xsd#int xi = i; // i is a System.Int32
6 }

```



Note that in Zhi# only local variables can be constrained. Especially, properties of .NET objects that are used along with relational operators are not subject to type inference since the values that are yielded by .NET properties may be different for each single property access.

In the preceding code snippet, the expression  $i < 10$  can be used for type inference since the C# “less than” relational operator ( $<$ ) corresponds to the XML Schema Definition constraining facet *xsd:maxExclusive*. In order to cover all XSD constraining facets seven additional comparison operators as listed in Table 3.2 were added to the Zhi# language grammar. The effects of these operators on type inference are described in the following subsections.

### 5.1.3.1 $?>$ , $?=$ , and $?<$ Operator

In Zhi#, the string data types *System.String* and *xsd:string* define “minimum length” ( $?>$ ), “length” ( $?=$ ), and “maximum length” ( $?<$ ) operators that return true if the first operand has the given minimum length, exact length, and maximum length, respectively. The second operand can be a .NET or an XSD integer type or a *typeof*-expression used with an XSD string type. In the following Zhi# code snippet, the string length of the .NET string variable *s* is tested to not exceed the maximum string length that is defined for the XSD string type *streetAddress*. An error is emitted by the Zhi# compiler if no *xsd:maxLength* constraint is defined for type *streetAddress*.

```
1 string s = [...];  
2 if (s ?< typeof(#cc#streetAddress)) { [...] }
```

Additionally, the *Length* property that is defined for the .NET *System.String* data type is used for type inference along with the “greater than or equal” ( $>=$ ), “greater than” ( $>$ ), “less than” ( $<$ ), and “less than or equal” ( $<=$ ) operators. For the block of statements within the *if*-statement in the following code snippet the type of the .NET string variable *s* is inferred to be *System.String*{ $?< 10$ }.

```

1 string s = [...];
2 if (s.Length < 10) { [...] }

```

### 5.1.3.2 ?? Operator

The *xsd:pattern* constraint restricts the value space of a data type by constraining its lexical space to literals that match a specific pattern. In Zhi#, the relational “pattern” operator (??)<sup>1</sup> can be used to test the string representation of an object to match a regular expression. For .NET objects the string representation corresponds with the return value of the ubiquitous *ToString()*-method. In Zhi#, XSD objects define an implicit conversion operator to *System.String*. The second operand of the “pattern” operator must be either a string literal, which is statically checked to denote a valid regular expression, or a *typeof*-expression used with an XML data type for which a “pattern” constraint is defined. In the following code snippet, the .NET *System.String* variable *s* is checked to hold a word of the type 3 language defined by the regular expression “[0-9]5” (i.e. a number with five digits).

```

1 string s = [...];
2 if (s ?? "[0-9]{5}") { [...] }

```

Because regular languages are closed under intersection, union, and complementation the word problem can be reduced to the emptiness problem. The efficiency of this procedure heavily depends on how a type 3 language is defined. If both languages are given as DFAs the complexity is  $O(n^2)$ . Unfortunately, if both languages are given as NFAs or – as is the case here – as regular expressions, the equivalence problem is NP-hard. In the current implementation of the Zhi# XSD compiler plug-in, two XML data types for which *xsd:pattern* constraints are defined are compatible only if both patterns (i.e. regular expressions) are syntactically identical.

---

<sup>1</sup>In C# 3.0 the “null coalescing” operator (??) was introduced, which checks whether the value provided on the left side of the expression is *null*, and if so it returns an alternative value indicated by the right side of the expression.

### 5.1.3.3 ?\$ Operator

The *xsd:enumeration* pattern constrains the value space of an XML data type to a specified set of values. In Zhi#, the relational “enumeration” operator (?\$) checks whether the value provided on the left side of the expression is an element of the value space of the type provided on the right side. The second operand of the “enumeration” operator must be a *typeof*-expression used with an XML data type for which an *xsd:enumeration* constraint is defined. The set of values cannot be explicitly specified yet in order to keep the Zhi# language grammar simple. It is conceivable, though, to allow for literal set notations of the form  $\{x, y, z\}$  or container objects such as instances of *System.List*. In the following Zhi# program, the value of the .NET *System.String* variable *s* is checked to be a valid title according to the *title* type definition.

```
1 string s = [...];  
2 if (s ?$ typeof(#cc#title)) { [...] }
```

### 5.1.3.4 %. Operator

The *xsd:fractionDigits* pattern controls the size of the minimum difference between values in the value space of data types derived from *xsd#decimal* by restricting the value space to numbers that are expressible as  $i \times 10^{-n}$  where *i* and *n* are integers and  $0 \leq n \leq \textit{fractionDigits}$ . The value of *fractionDigits* must be a non-negative integer. Note that *xsd:fractionDigits* does not restrict the lexical space directly; a non-canonical lexical representation that adds additional leading zero digits or trailing fractional zero digits is still permitted. In Zhi#, the relational “fraction digits” operator (%) checks whether the *xsd#decimal* number on the left side of the expression has at most a given number of fractional digits. The second operand of the “fraction digits” operator must either be a .NET or an XSD integer object or a *typeof*-expression used with an XML data type for which an *xsd:fractionDigits* constraint is defined. An *xsd#decimal* variable *d* can be checked to have at most two fractional digits as follows.

```

1 #xsd#decimal d = [...];
2 if (d %. 2) { [...] }

```

### 5.1.3.5 %% Operator

The *xsd:totalDigits* pattern controls the maximum number of values in the value space of data types derived from *xsd#decimal* by restricting it to numbers that are expressible as  $i \times 10^{-n}$  where  $i$  and  $n$  are integers such that  $|i| < 10^{\text{totalDigits}}$  and  $0 \leq n \leq \text{totalDigits}$ . The value of *totalDigits* must be a positive integer. Note that *xsd:totalDigits* does not restrict the lexical space directly; a lexical representation that adds additional leading zero digits or trailing fractional zero digits is still permitted. In *Zhi#*, the relational “total digits” operator (%%) checks whether the *xsd#decimal* number on the left side of the expression has at most a given number of total digits. The second operand of the “total digits” operator must either be a .NET or an XSD integer object or a *typeof*-expression used with an XML data type for which an *xsd:totalDigits* constraint is defined. An *xsd#decimal* variable  $d$  can be checked to have at most four total digits as follows.

```

1 #xsd#decimal d = [...];
2 if (d %% 4) { [...] }

```

Constraints that are transiently added to the types of variables are preserved when the same variable is again subject to type inference. In this way, the type of an object can be accumulatively inferred to eventually have the required value space. An *xsd#decimal* variable  $d$  can be constrained to have at most two fractional digits and at most four total digits as shown in the following code snippet. In line 3 the type of  $d$  is inferred to be *xsd#decimal*{%. 2}. For the block of embedded statements in line 4 the type of  $d$  is inferred to be *xsd#decimal*{%. 2}{%% 4}.

Note how the accumulation of constraints with each controlling expression in *if*, *for*, and *while*-statements directly corresponds to the construction of types in the  $\lambda_C$ -calculus using the rule TD-CSTRAPP (see Section 3.2).

```

1 #xsd#decimal d = [...];
2 if (d %. 2) {
3   if (d %% 4) { // d is an xsd#decimal{%. 2}
4     [...]      // d is an xsd#decimal{%. 2}{%% 4}
5   }
6 }

```

### 5.1.3.6 *is*-Operator

Instead of considering every single constraint separately, which requires manual lookup in the XML schema and using a number of nested *if*-statements, value spaces may be checked as a whole as shown in the code snippet below. In Zhi#, the *is*-operator (*is*) can be used to check that the value space of the type of the operand on the left side is subsumed by the value space of the type reference on the right side of the expression. The first operand of the *is*-operator can be an instance of an XML data type or a primitive .NET value type from the list in Table 5.1. The second operand must be a type reference of an XML data type. The Zhi# compiler emits an error if the given expression is never of the provided type. In the current implementation of the Zhi# programming language the second operand must be a type reference that follows the syntactic requirements stipulated in Subsection 5.1.1.

```

1 string s = [...];
2 if (s is #cc#streetAddress) { [...] }

```

For type systems that are known beforehand it is conceivable to extend the Zhi# language grammar to allow for anonymous type definitions as the second operand of the *is*-operator. Anonymous type definitions that are embedded in Zhi# programs would be agnostic to changes in the referenced XML schema definitions while for referenced named types the definitions in the schema may change and the value space check still functions properly with respect to the modified type definition (cf. Subsection 5.1.2.3).

### 5.1.3.7 Literals

In Zhi#, it is also possible for compiler plug-ins to infer more specific types of .NET literals. A literal is a source code representation of a value. In C#, there are boolean-literals, integer-literals, real-literals, character-literals, string-literals, and *null*-literals.

For the types of integer-literals it is possible to statically add the XSD constraints *xsd:minExclusive*, *xsd:minInclusive*, *xsd:maxExclusive*, *xsd:maxInclusive*, *xsd:totalDigits*, and *xsd:fractionDigits*. In the following variable declaration, the type of integer *i* is inferred by the XSD compiler plug-in to be *System.Int32*{> 22}{>= 23}{<= 23}{< 24}{%% 2}{%.0}. The constraints {> 22} and {< 24} are the result of the application of the constraint inference rules described in Section 3.7.

```
int i = 23;
```

The XSD constraints *xsd:minInclusive* and *xsd:maxInclusive* are added to the types of real-literals. In the following variable declaration, the type of the real number *r* is inferred to be *System.Double*{>= 23}{<= 23}.

```
double r = 23;
```

The XSD constraints *xsd:minLength*, *xsd:length*, and *xsd:maxLength* are added to the types of string-literals. In the following variable declaration, the type of the *string* variable *s* is inferred to be *System.String*{?> 3}{?= 3}{?< 3}.

```
string s = "abc";
```

Note that additional compiler plug-ins may cooperatively add further non-XSD constraints to the types of both literals and variables. In particular, constraints from different external types systems are not mutually exclusive. For literals and binary expressions in Zhi# programs, all activated compiler plug-ins are consulted one after the other in order to add constraint information. Hence, additional plug-ins should be aware of the syntax of the constraining facets from existing compiler plug-ins and should not reuse the same constraint symbols.

### 5.1.4 Compilation to C#

Zhi# program text is compiled into conventional C#. External type and member references as well as binary expressions that relate instances of external types are replaced by .NET proxy types and multi-argument function calls, respectively.

The abstract  $\lambda_C$ -type system class *RTSimpleType* (see Fig. 3.1) is substituted for XML Schema Definition type references in compiled Zhi# programs (i.e. *all* referenced XSD types are represented by *one* single proxy class). The  $\lambda_C$ -type system interfaces were implemented as described in Section 3.6 in order to implement XSD type definitions and constraint-based type derivation. In particular, 19 implementations of the abstract *RTSimpleType* class embody the 19 built-in XSD primitive types as listed in Table 5.6. Note that none of these concrete classes is explicitly referenced in compiled Zhi# programs. Instead, C#'s polymorphism is used to make *RTSimpleType* objects respond to method invocations according to the actual XML data type for which they stand.

Table 5.6: XSD built-in primitive types

---

<i>anyURI</i>	<i>base64Binary</i>	<i>boolean</i>	<i>date</i>
<i>dateTime</i>	<i>decimal</i>	<i>double</i>	<i>duration</i>
<i>float</i>	<i>gDay</i>	<i>gMonth</i>	<i>gMonthDay</i>
<i>gYear</i>	<i>gYearMonth</i>	<i>hexBinary</i>	<i>NOTATION</i>
<i>QName</i>	<i>string</i>	<i>time</i>	

---

The following Zhi# code snippet is compiled into the C# program shown below. The XSD types *cc#itemPrice* and *cc#invoiceTotal* are both represented by the same proxy type *RTSimpleType* in the output C# program. The source values 23 and 42 are not directly assigned to the variables *p1* and *p2*. Instead, the *NewXSD* method provided by the *ZhiSharpRuntime* class is used to create *RTSimpleType* objects from the given values. Because expressions, which comprise XML data types, are statically type-checked the creation of the proxy type instances is guaranteed to succeed at runtime for immutable

XML schema definitions; it may fail if the `Zhi#` program is started with modified XSD type definitions (cf. Subsection 5.1.1).

```
1 #cc#itemPrice p1 = 23;
2 #cc#itemPrice p2 = 42;
3 #cc#invoiceTotal total = p1 + p2;

1 RTSimpleType p1 = ↵
2   NewXSD("http://www.chokycola.com/business#itemPrice", 23);
3 RTSimpleType p2 = ↵
4   NewXSD("http://www.chokycola.com/business#itemPrice", 42);
5 RTSimpleType total = ↵
6   NewXSD("http://www.chokycola.com/business#invoiceTotal", ↵
7     Addition("http://www.w3.org/2001/XMLSchema#double", ↵
8               {>= "0"}{< "200"}{<= "200"}", ↵
9               p1, p2));
```

The partial static `ZhiSharpRuntime` class is required at runtime by compiled `Zhi#` programs. Its static constructor loads external type definitions and initializes external type systems. External compiler plug-ins can extend the `ZhiSharpRuntime` class with type system specific functionalities. The XSD compiler plug-in contributes among others the static `NewXSD` and `Addition` methods and the `RTSimpleType` class.

XML schema definitions that are to be loaded by the `Zhi#` runtime must be specified in the `App.config`<sup>2</sup> configuration file as values of the `ZhiSharpXSDFiles` key as follows.

```
1 <configuration>
2   <appSettings>
3     <add key="ZhiSharpXSDFiles" value="chokycola.xsd"/>
4   </appSettings>
5 </configuration>
```

---

<sup>2</sup>.NET programs can be configured using an XML configuration file named `App.config`. Its contents can be conveniently extracted using the .NET `ConfigurationManager` class.



In Zhi# programs, binary expressions that comprise instances of XSD types are translated into multi-argument function calls on the Zhi# runtime library.

$$x = a \ \tau \ b \ \xrightarrow{\text{is translated into}} \ x = f_{\tau}(a, b, \dots)$$

The addition expression  $p1 + p2$  in line 3 the Zhi# code snippet above is translated into an invocation of the *Addition* function provided by the XSD plug-in for the Zhi# runtime library. The *Addition* function adds the *RTSimpleType* objects  $p1$  and  $p2$ . The sum object is then taken as the input for the creation of an *xsd#double*{>= "0"}{< "200"} object. This constrained *xsd#double* data type is statically inferred as the type of the expression  $p1 + p2$ . The result of the addition operation is also dynamically type-checked by the generated C# code to not exceed the statically computed value space. Note that the anonymous constrained *xsd#double* data type is agnostic to schema changes at runtime. Eventually, the output of the *Addition* function is used to initialize a *cc#invoiceTotal* variable. This variable definition is dynamically type-checked such that, the XSD lvalue *total* will never be assigned an invalid value according to the schema definition loaded at runtime. In particular, the assigned value will neither be too general nor too specific. If a too specific object was assigned to an XSD variable subsequent assignments would fail in the presence of the covariant subtyping for constrained data types because *RTSimpleType* objects do not only aggregate the *value* but also the *type* of the represented XML data type value (i.e. the target object may subsequently be guarded by too restrictive constraints of the previous source object). This is why definitions and assignments that involve constrained types are translated into object creation expressions and setter-function calls on the Zhi# runtime library, respectively.

In Subsection 5.1.2.8 an example is given how assignments to XSD variables are compiled into C# in the presence of covariant subtyping. See Appendix B.1 for how Zhi# code that contains XML Schema Definition data types is translated into C# by the XSD compiler plug-in.

### 5.1.5 Dynamic checking

In Zhi#, XML Schema Definition objects can be downcast to XML data types whose value space is a subset of the value space of the original type. This conversion is analogous to downcasting in C# where a variable is bound to a value that is an instance of a class that is a subtype of the variable's original type. Downcasting is useful because it allows symmetric binding of classes. Still, *InvalidCastException*s can occur at runtime if the cast object is not of the provided type. The C# compiler statically checks if the provided type is a subtype of the type of the cast object in order to avoid “unnecessary” runtime errors. Analogously, the Zhi# compiler reports an error if an XSD object is cast to a type that is incompatible with the primitive base type of the object. Other possible constraint and value space violations at runtime dwindle to compiler warnings.

At runtime, invalid downcasts to XML data types are detected by Zhi#'s dynamic type checking<sup>3</sup> for XSD types. In Zhi#, *every* assignment to an XSD variable is dynamically checked. This includes assignments that were statically type-checked and considered to be safe as well as unsafe expressions such as downcasts. For performance reasons dynamic type checks of statically checked expressions may be omitted for code that is not visible to other .NET programming languages than Zhi# if the underlying XML schema definitions are not subject to change after compilation of the Zhi# program. For the sake of simplicity, the current implementation of the Zhi# compiler generates code that contains dynamic type checks for all assignments that comprise instances of XML data types. As a consequence, XML schemata, which contain the referenced type definitions, are allowed to be modified after compilation before startup of a Zhi# application (this may heavily influence the execution of the Zhi# program, though). Moreover, compiled Zhi# assemblies can be used by different .NET programming languages since assignments to the *Value* field of the *RTSimpleType* class are always checked to be valid in respect of the underlying XML Schema Definition.

---

<sup>3</sup>In [Pie02], Pierce notes that “Terms like ‘dynamically typed’ are arguably misnomers and should probably be replaced by ‘dynamically checked’, but the usage is standard.”

For assignments to XSD variables a *ViolatedLexicalSpaceException* and a *ViolatedConstraintException* are thrown if the value that is represented by the source object is not an element of the lexical space and the value space, respectively, of the type of the XSD lvalue. Standard XML Schema Definition validation principles as implemented in the  $\lambda_C$ -type system are employed as depicted in Fig. 5.5. First, modifying facets such as *xsd:whitespace* are applied to the source object's string representation. Next, the lexical space is checked to be valid for the given built-in primitive base type from which the type of target object *t* is derived. Subsequently, all constraining facets that are defined for the XSD target type are enforced. Finally, the *Value* field of the XSD object *t* (i.e. the instance of *RTSimpleType*) is updated.

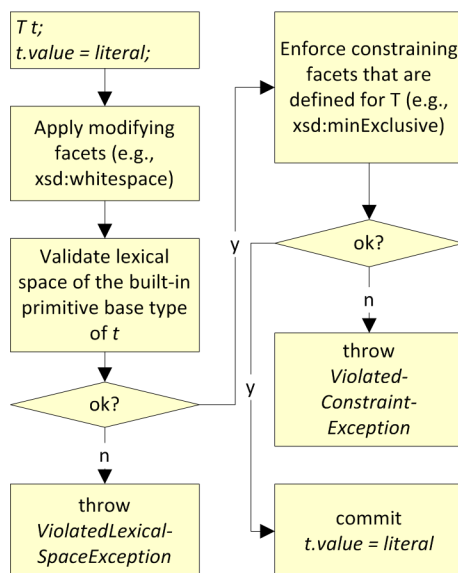


Figure 5.5: XSD validation

Zhi# programs are fully interoperable with standard .NET assemblies. In particular, publicly visible instances of constrained XML data types can be accessed in conventional C# programs via the *Value* property of the *RTSimpleType* proxy class. In this way it is possible to assign *System.String* values to and read *System.String* values from XSD objects with the same dynamic type checking as described above (i.e. it is not possible to bypass stipulated XML Schema Definition constraints).

## 5.2 Related Work

Around ten years ago computers eventually became fast enough to afford the luxury of XML (recall that XML is a subset of SGML, which was invented already in 1969). Since then major software companies have come out strongly in favor of standardizing both on XML and XML Schema Definition. As a result, a number of approaches have emerged to define programming languages specifically for the XML domain.

XDuce [HP03] is a functional programming language that is specifically designed for processing XML data. One can read an XML document as an XDuce value, extract information from it or convert it to another format, and write out the result value as an XML document. The subsequent Xtatic project [GP03] aims to develop theoretical foundations and implementation techniques for a lightweight extension of C# tailored for native XML processing. In contrast to Zhi#, both approaches focus on content models of aggregated XML structures and lack support for atomic XML data types. In this way, they are similar to Xen and C $\omega$ , which are amalgamations of Microsoft's Common Language Runtime (CLR) [Mic06], XML, and SQL programming languages.

The JWIG development system [CMS03] is a Java-based high-level language for the development of interactive Web services. J Wig integrates the central features of the <bigwig> language [BMS02] into Java by providing explicit support for web service sessions and safe XHTML dynamic document construction. In particular, J Wig facilitates the construction of XHTML documents by introducing XML templates, which can contain inlined pieces of code, called code gaps. At runtime, code gaps can be substituted by other templates or literals. In the Xact project [KMS04], J Wig's validation algorithm is extended to implement further compile time guarantees such that, dynamically transformed XML documents are valid according to a given XML schema.

XL [FGK02] adopts to XQuery [BCF07] and combines imperative and declarative programming language features to facilitate the development of Web services. The XML Objects Programming Language [SL05] integrates XML and XPath [CD99] into the Java

programming language with a main emphasis on valid updates of persistent XML objects.

In [BW04], a constraint algebra is proposed in which complex constraint expressions can be built up from primitive constraints using logical connectives like conjunction or disjunction. This algebra, however, has not been embedded with a type system. A constraint model for XML is presented in [FKS01]. Just like the XML domain specific languages mentioned above this model focuses on the content model of aggregated types and does not include value space constraints of atomic data types as they may be defined by XML Schema Definition.

To the best of the author’s knowledge, all of these approaches lack support for constrained atomic XML data types. Rather, the Zhi# approach presented in this work is complementary to some technologies that do only support content models of complex XML Schema Definition types.

Type qualifiers can be used to express properties of objects in addition to their type information. For example, in the C programming language, the type qualifier *const* can be used to declare variables constant in order to allow the compiler to make certain optimizations. Type qualifiers encode a simple form of subtyping, which can be used to guarantee a certain program behavior at runtime or to find bugs.

Foster et al. [FTA02] developed the CQUAL tool that can be used to extend standard types with flow-sensitive type qualifiers. In contrast, Zhi#’s constrained atomic types are not decorated with type qualifiers in the program text but are defined using value space constraints (i.e. value space “qualifiers” are first class citizens of type construction). Unlike CQUAL, flow-sensitivity in Zhi# is not restricted to constrained (i.e. qualified) types but is also used to infer properties of .NET value type objects. In Zhi#, subtype relations of constraints are not defined in an ad hoc manner but by formal subtyping rules. Brian Chin et al. introduced semantic type qualifiers [CMM05]. This work was later augmented with inference of type qualifiers and qualifier rules in a system called CLARITY [CMM06]. Their framework can be used to restrict the values of expressions (e.g., *nonnull*, *nonzero*)

and to restrict the flow of values through a program (e.g., *tainted*, *untainted*). The former class of qualifiers is similar to constrained types in Zhi#. An important difference here is that in the work of Brian Chin et al. associated type rules of type qualifiers are defined for particular code patterns (i.e. knowledge is assumed about the syntax of the programming language), which is not required for constrained type definitions in Zhi#. In both approaches constrained (i.e. qualified) types are considered to be subtypes of their associated base types. In Zhi#, value space constraints are automatically classified based on formal subtyping rules while the language of Brian Chin et al. does not support explicit subtype declarations between two user-defined qualifiers. The CQUAL system supports subtyping relationships among qualifiers, too. The CLARITY framework boasts qualifier rule inference from given qualifier invariants. By contrast, in Zhi#, the type of a term is computed by combining the types for its subterms. Subtyping orders of type qualifiers are supported by the JQUAL tool for Java [GF07]; users have to specify the subtyping order manually, though. In contrast to the Zhi# compiler framework, JQUAL assumes that the input program is correct with respect to the standard Java types.

### 5.3 Summary

The XML Schema Definition type system was implemented based on the formal foundation of the  $\lambda_C$ -calculus as described in Chapter 3. XML Schema Definition data types were integrated with the Zhi# programming language by means of a plug-in for the Zhi# compiler framework that facilitates loading of XML schema definitions, static typing of constrained atomic data types in Zhi# programs, and the transformation of XSD type references in Zhi# programs into C# code. Except for conjoint type derivation along with different type systems, XML data types can be used in any context of Zhi# programs that is valid for built-in .NET value types. In particular, XML Schema Definition type system rules were integrated with programming language features such as method overriding, user-defined operators, and runtime type checks (i.e. *XSD aware compilation*).

The XSD compiler plug-in implements static typing of XML data types that takes into account both nominal and structural aspects of the XML Schema Definition type system. Implicit conversions are provided between primitive .NET value types and XML data types for which a value space based subtype relation exists. The XSD compiler plug-in enforces invariant subtyping for XSD array types, which is a deviation from the covariant subtyping of .NET array types, and allows covariant subtyping for non-array types.

The Zhi# compiler framework supports the implementation of static type inference rules by compiler plug-ins. The compiler plug-in for XML Schema Definition infers types of variables based on control and data flow analysis. The type inference rule for *if*-statements in the  $\lambda_C$ -calculus was complemented with type inference rules for *for* and *while*-statements. The types of literals are inferred based on the literal values. Types of binary arithmetic expressions are inferred based on the constraint arithmetic of the  $\lambda_C$ -type system. In order to cover all XSD constraining facets seven additional comparison operators as listed in Table 3.2 were added to the Zhi# language grammar.

Zhi# program text is compiled into conventional C#. External type and member references as well as binary expressions that relate instances of external types are replaced by .NET proxy types and multi-argument function calls, respectively. The abstract  $\lambda_C$ -type system class *RTSimpleType* (see Fig. 3.1) is substituted for XML Schema Definition type references in compiled Zhi# programs (i.e. *all* referenced XSD types are represented by *one* single proxy class). The  $\lambda_C$ -type system interfaces were implemented as described in Section 3.6 in order to implement XSD type definitions and constraint-based type derivation. In particular, 19 implementations of the abstract *RTSimpleType* class embody the 19 built-in XSD primitive types. The generated C# code facilitates dynamic checking of XML data types, which is provided by the accompanying XSD plug-in for the Zhi# runtime library. The XSD plug-in detects invalid downcasts to XML data types and invalid assignments to the *Value* property of compiled XML data types by conventional C# code. In this way, Zhi# programs are fully interoperable with standard .NET assemblies with full consideration of the stipulated XML Schema Definition value space constraints.





## CHAPTER 6

# OWL Aware Compilation – Complex Data, Simple Code

Widely used object-oriented programming languages such as Java or C# include a built-in static type system. Programming language type systems provide a conceptual framework that makes it particularly easy to design, understand, and maintain object-oriented systems. However, with the emergence of ontology languages such as the Web Ontology Language (OWL) [MH04a] conventional built-in programming language type systems have reached their limits.

In course of the CHIL research project [Inf04] an ontology was deployed in order to provide a formal high level description of the CHIL domain of discourse that can be efficiently used to build intelligent applications. By using the Description Logics [BCM03] based Web Ontology Language (OWL DL) for modeling concepts, properties, and individuals of the CHIL domain of discourse it is possible to use a reasoner for automatically checking the consistency of the knowledge base and making implicit knowledge explicit, which can be considered to be a form of artificial intelligence.

A typical OWL DL knowledge base comprises two components – a *TBox* and an *ABox*. The TBox contains general knowledge about a problem domain in form of a terminology and is built through declarations that describe general properties (roles) of concepts. The ABox contains assertional knowledge, which is specific to the individuals of the domain of discourse. While the TBox is usually thought to be immutable, the ABox may be subject to occasional or even constant change. In particular, modifications may lead to

an ABox that violates terminological definitions such as cardinality constraints or value space restrictions of OWL datatype properties.

Up to now, ontological knowledge bases are modified using APIs, which are provided by a variety of different ontology management systems [HP 04, MH04b, Sta06]. From a software developer’s perspective, there is no support for statically detecting illegal operations based on given terminologies (e.g., undefined concepts, invalid datatype property values) and conveniently integrating ontological concepts, roles, and individuals with the program text of a general purpose programming language.

A common difficulty of widely used OWL APIs and academic approaches to use wrapper classes to represent elements of an ontology are the different conceptual bases of types and instances in a programming language and concepts, roles, individuals, and XML Schema Definition [BM04b] data type values in OWL DL. In particular, the Web Ontology Language reveals the following major differences to object-oriented programming languages and database management systems.

- Ontology entailment for the Web Ontology Language can be reduced to knowledge base satisfiability in the  $\mathcal{SHOIN}(\mathbf{D})$  Description Logic [HP04]. In contrast to object-oriented programming languages and less expressive Description Logics,  $\mathcal{SHOIN}(\mathbf{D})$  provides a rich set of concept constructors (see Table 1.4). For example, concepts can be described via cardinality and value restrictions on properties (e.g., a small meeting is a meeting with at most three participants). As a consequence, the consistency of a knowledge base cannot be checked “statically” on the meta-level since both a particular TBox *and* a particular ABox are required to infer the RDF types of individuals in the knowledge base.
- OWL concept descriptions are automatically classified in a subsumption hierarchy. Imitating this inherent behavior of ontological knowledge bases using a hierarchy of programming language wrapper classes would result in reimplementing a complete OWL DL reasoner.

- Unlike object-oriented programming languages or database management systems, OWL makes the *open world assumption* (OWA), which codifies the informal notion that in general no single observer has complete knowledge. The open world assumption limits the deductions a reasoner can make. In particular, it is not possible to infer from the lack of knowledge of a statement being true, anything that follows from that statement being false. For example, assuming a concept description of a big meeting, which is to have at least 5 participants, a consistency check will not fail if there are big meeting instances with less than five participants. From the absence of information about meeting participants in the available knowledge, a reasoner cannot surmise that these additional participants actually do not exist in the real world; they can only be considered unknown. A consistency check will fail, however, if there are small meetings with too many participants. Note that the OWA is closely related to the monotonic nature of first-order logic (i.e. adding information never falsifies previous conclusions).
- The Web Ontology Language does not make the *unique name assumption* (UNA). In contrast to logics with the unique name assumption, different ontological individuals do not necessarily refer to different entities in the described world. In fact, two individuals can be inferred to be identical (e.g., values of functional object properties can be inferred to be identical because of the property cardinality of ‘1’). In OWL, it is also possible to explicitly declare that two given named individuals have the same identity or different identities.
- Unlike object-oriented programming languages, ontological roles in OWL DL are not defined as part of class definitions but form a hierarchy of their own (i.e., property centric modeling).
- In OWL, property domain and range restrictions are not authoritative. Instead, the defined domain and range of an OWL property is used to infer the types of the subjects and objects of RDF triples, respectively.

With programming language inherent support for constrained atomic XML Schema Definition data types, Zhi# provides the prerequisite for compile-time support and dynamic type checking for Web Ontology Language concept descriptions. The implemented compile-time support includes not only static type checking of expressions that involve OWL DL objects but also features of the Zhi# development environment such as auto-completion of ontological concept and role names.

## 6.1 Integrating OWL DL with the C# Programming Language

Based on the formally specified CHIL OWL API (see Chapter 4) an OWL plug-in [Paa07] for the Zhi# compiler framework was developed that makes it possible to use OWL concepts, roles (properties), and individuals in Zhi# programs. In Zhi#, ontological reasoning is integrated with the conventional .NET type system and C# programming language features. In particular, the OWL compiler plug-in can be used cooperatively with the XSD plug-in described in the previous chapter in order to facilitate the handling of OWL datatype properties. Imported OWL concept descriptions are subject to both static typing and dynamic checking. Zhi# expressions that comprise ontological concepts and roles are transformed into conventional C# code.

### 6.1.1 Referencing OWL DL Ontologies

Zhi#'s *ontology aware compilation* makes it possible to reference ontological concepts and roles as defined in OWL DL ontologies and to use these references along with constrained XML Schema Definition data types and C# programming language features. The Zhi# compiler takes as input Zhi# source files and OWL DL ontologies encoded in RDF/XML syntax as depicted in Fig. 6.1. Both the CHIL OWL API and the Zhi# OWL compiler plug-in support XML data types. The CHIL Knowledge Base server uses an implementation of the  $\lambda_C$ -type system (see Chapter 3). The Zhi# OWL compiler plug-in delegates the evaluation of XSD objects to the XSD compiler plug-in.

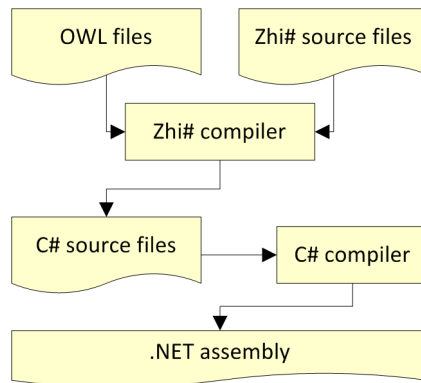


Figure 6.1: OWL aware compilation

The Zhi# keyword *import* works analogously for OWL concepts and roles like the C# *using* keyword for .NET programming language type definitions. It can be used to import concepts and roles that are defined in OWL DL ontologies. OWL concepts and roles can be used in a Zhi# namespace such that, one does not have to qualify the use of a concept or role in that namespace. Concepts and roles of the TBox describing the domain of discourse on which a software application operates are made available right in the application’s program text.

The following OWL DL ontology, encoded in RDF/XML syntax, declares the ontological concepts *Meeting* and *ActivityLevel*, the ontological role *hasActivityLevel*, and the ontological individuals LOW and HIGH in the namespace *http://chil.server.de/ontology*. In the Zhi# program shown below the concepts and roles are referenced by using the alias *cho*, which is bound to the ontology namespace by the *import* statement in line 1 in the Zhi# program.

Even the shown few lines of Zhi# code indicate possible programming language features that are likely to ease the development of knowledge base client applications. First, the imported namespace could be checked to actually contain concept and role definitions. Second, the used concepts and roles could be checked to exist in the imported namespace. Eventually, the range description of the role *hasActivityLevel* can be statically checked to not be disjoint with the concept description *ActivityLevel*.

```

1 <rdf:RDF xmlns="http://chil.server.de/ontology" [...] >
2   <owl:Ontology rdf:about=""/>
3   <owl:Class rdf:ID="ActivityLevel"/>
4   <owl:Class rdf:ID="Meeting"/>
5   <owl:ObjectProperty rdf:ID="hasActivityLevel">
6     <rdfs:domain rdf:resource="#Meeting"/>
7     <rdfs:range rdf:resource="#ActivityLevel"/>
8   </owl:ObjectProperty>
9   <ActivityLevel rdf:ID="Low"/><ActivityLevel rdf:ID="HIGH"/>
10 </rdf:RDF>

```

```

1 import OWL cho = http://chil.server.de/ontology;
2 namespace MyCHILApplication {
3   class MyClass {
4     #cho#Meeting m = [...];
5     m.#cho#hasActivityLevel =
6       new #cho#ActivityLevel("http://chil.server.de/ontology#HIGH");
7   }
8 }

```

For each imported OWL namespace the Zhi# OWL compiler plug-in loads the respective type definitions from the referenced RDF/XML library files. The set of library files that is made available to each Zhi# compiler plug-in comprises all referenced non-source code files (e.g., .xsd-, .owl-, .dll files). From these files RDF/XML encoded ontology definitions (i.e. .owl- and .rdf-files) are considered by the OWL compiler plug-in. A compile-time error occurs if no ontology is defined in imported OWL namespaces. Concept descriptions and ontological roles that are defined in the imported OWL namespaces may be used in the code following the import statement. OWL concepts and roles are subject to static typing and dynamic checking, and are syntactically highlighted in the Eclipse-based Zhi# editor. Also, they are available as autocompletion proposals (cf. Fig. 5.2).

### 6.1.2 Static typing

C# is a statically typed programming language. Type checking is performed during compile time as opposed to runtime. As a consequence, many errors can be caught early at compile time (i.e. fail-fast), which allows for efficient execution at runtime. Unfortunately, the non-contextual property centric data modeling features of the Web Ontology Language render compile-time type checking only a partial test on Zhi# program text. In a well-typed C# program, every expression is guaranteed to be of a certain type at runtime. In Zhi#, the same is not generally true for ontological individuals. Eventually, static type checking of computer programs that use elements of an ontology amounts to ontological reasoning on the meta-level, which cannot be complete for all possible incarnations of an ontology. This subsection describes the static type checks that *can* be performed on ontological expressions in Zhi# programs.

#### 6.1.2.1 Syntax checks

The most fundamental compile-time feature that the OWL plug-in provides is checking the existence of referenced ontology elements in the imported terminology. The C# statements below declare the ontological individuals A and B. Individual B is added as a property value for property *R* of individual A. For the sake of brevity, in this chapter, the URI fragment identifier “#” may be used to indicate ontology elements in Zhi# programs instead of using fully qualified names. The object *o* shall be an instance of the C# implementation of the CHIL OWL API (see Chapter 4). The given code is a well-typed C# program. It may, however, fail at runtime if in the TBox of the referenced ontology classes *A* and *B* and property *R* do not exist.

```
1 ICHILOWLAPI o = [...];
2 o.addIndividual("#A", "#A");
3 o.addIndividual("#B", "#B");
4 o.addObjectPropertyValue("#A", "#R", "#B");
```

In Zhi#, the same declarations can be rewritten as shown below. As a result, the Zhi# compiler statically checks if concept descriptions  $A$  and  $B$  and property  $R$  exist in the imported ontology and raises an error if they are undefined.

```
1 import OWL alias = ontology namespace ;  
2 #A a = new #A("#A");  
3 #B b = new #B("#B");  
4 a.#R = b;
```

### 6.1.2.2 Creation of individuals

In C#, the *new* operator can be used to create objects on the heap and to invoke constructors. In Zhi#, the *new* operator can also be used to answer ontological individuals in a knowledge base as follows.

```
1 #Event e = new #Meeting("#BRIEFING");  
2 #Meeting m = new #Event("#BRIEFING"); // Rejected by Zhi# compiler
```

For ontological concepts the Zhi# programming language provides a constructor that takes the URI of the individual. As in conventional C#, the *new* operator cannot be overloaded. In contrast to .NET objects, ontological individuals are not created on the heap but in the addressed ontological knowledge base, and as such they are subject to ontological reasoning. This is in contrast to naïve approaches where wrapper classes for ontological concepts are instantiated as plain .NET objects. In Zhi#, the instances of the single wrapper class are just handles to the individuals in the ontology. Also note that any existing individual in the ontology with the given identifier is reused, which is the standard behavior of ontology management systems for the Web Ontology Language.

As for assignments of .NET object creation expressions to variables or fields, the type of the individual creation expression must be subsumed by the type of the lvalue based on the classified concept hierarchy (see line 2 in the code snippet above). Zhi# supports covariant coercions for ontological individuals and arrays of ontological individuals.



### 6.1.2.3 Disjoint concepts

In OWL DL, classes can be stated to be disjoint from each other using the *owl:disjointWith* constructor. It guarantees that an individual that is a member of one class cannot simultaneously be an instance of a specified other class. For example, *MeetingRoom* and *LargeRoom* can be stated to be disjoint classes as follows.

```
1 <rdf:RDF xmlns="http://chil.server.de/ontology" ...>
2   <owl:Class rdf:ID="LargeRoom">
3     <owl:disjointWith>
4       <owl:Class rdf:ID="MeetingRoom"/>
5     </owl:disjointWith>
6     <rdfs:subClassOf>
7       <owl:Class rdf:ID="Location"/>
8     </rdfs:subClassOf>
9   </owl:Class>
10  <owl:Class rdf:about="#MeetingRoom">
11    <owl:disjointWith rdf:resource="#LargeRoom"/>
12    <rdfs:subClassOf rdf:resource="#Location"/>
13  </owl:Class>
14 </rdf:RDF>
```

From the *owl:disjointWith* statements a reasoner will deduce an inconsistency when an individual is stated to be an instance of both classes. Similarly a reasoner can deduce that if an individual is an instance of *MeetingRoom*, then it is not an instance of *LargeRoom*. These conclusions do not depend on the context (i.e. a particular ABox) of the disjoint class definitions and can thus be drawn on the meta-level. In Zhi#, the disjointness of classes is exploited to statically type-check cast expressions and property value declarations. The following Zhi# program will produce a compile-time error in line 2 because the ontological individual that is referred to by variable *l* will never be in the extension of the ontological concept *MeetingRoom* for consistent ontologies.

```

1 #LargeRoom l = [...];
2 #MeetingRoom m = (#MeetingRoom) l;

```

Property domain and range restrictions along with disjoint concept descriptions are exploited by the Zhi# compiler as follows. The range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. In OWL, multiple range values are interpreted as a conjunction. In the following ontology snippet, which extends the previous definitions of a *MeetingRoom* and a *LargeRoom*, the property *takesPlaceInAuditorium* is stated to have the range of *LargeRoom*. From this a reasoner can deduce that if an event takes place in an auditorium then then this location must be a *LargeRoom*.

```

1 <rdf:RDF xmlns="http://chil.server.de/ontology" ...>
2   <owl:Class rdf:ID="Event"/>
3   <owl:Class rdf:ID="Lecture">
4     <rdfs:subClassOf rdf:resource="#Event"/>
5     <owl:disjointWith>
6       <owl:Class rdf:ID="Meeting"/>
7     </owl:disjointWith>
8   </owl:Class>
9   <owl:Class rdf:about="#Meeting">
10     <owl:disjointWith rdf:resource="#Lecture"/>
11     <rdfs:subClassOf rdf:resource="#Event"/>
12   </owl:Class>
13   <owl:ObjectProperty rdf:ID="takesPlaceInAuditorium">
14     <rdfs:domain rdf:resource="#Lecture"/>
15     <rdfs:range rdf:resource="#LargeRoom"/>
16   </owl:ObjectProperty>
17 </rdf:RDF>

```

Consequently, the assignment in line 2 in the following Zhi# code snippet results in a compile-time error because – in the given ontology – a location can never be a *MeetingRoom* and a *LargeRoom* at the same time.

```
1 #Lecture l = [...];  
2 l.#takesPlaceInAuditorium = new #MeetingRoom ([...]);
```

The domain of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class (multiple domain values are interpreted as a conjunction). For example, the property *takesPlaceInAuditorium* is stated to have the domain *Lecture*. From this a reasoner can deduce that if an event takes place in a auditorium, then the this event must be a *Lecture*. As a consequence, the assignment in line 2 in the following Zhi# code snippet results in a compile-time error because an event can never be a *Meeting* and a *Lecture* at the same time.

```
1 #Meeting m = [...];  
2 m.#takesPlaceInAuditorium = new #LargeRoom ([...]);
```

Note that the described reasoning based on property domain and range restrictions, where the types of subjects and objects of RDF triples are inferred based on the properties' domain and range restrictions, happens only at runtime and is not available at compile time. The Zhi# compiler does not perform any type inference for ontological individuals, which are always considered to be of the explicitly declared type.

OWL DL includes language constructs to describe boolean combinations of class expressions. The *owl:complementOf* construct selects all individuals from the domain of discourse that do not belong to a given class . Therefore, similar conclusions can be drawn on the meta-level for *owl:complementOf* concept descriptions as for the *owl:disjointWith* concept constructor. This has, however, not yet been implemented in the current release of the Zhi# OWL compiler plug-in because *owl:complementOf* traits are more difficult to reconstruct for given concept descriptions.

#### 6.1.2.4 Disjoint XML data types

In *Zhi#*, a “frame-like” view on OWL object properties is provided by the *checked*-operator used in conjunction with assignments to OWL object properties (see Subsection 6.1.5). For assignments to OWL datatype properties in *Zhi#* programs, the “frame-like” composite view is the default behavior. The data type of the property value must be a subtype of the datatype property range restriction. The following assignment in line 2 fails to type-check for a datatype property *hasCapacity* with domain *MeetingRoom* and range *xsd#byte* because in *Zhi#* programs the literal *23.5* is interpreted as a .NET floating point value (i.e. *xsd#double*), which is disjoint with the primitive base type of *xsd#byte*.

```
1 #MeetingRoom r = [...];  
2 r.#hasCapacity = 23.5;
```

The XSD compiler plug-in allows for downcasting objects to compatible XML data types (i.e. XSD types that are derived from the same primitive base type). The assignment in line 3 in the following *Zhi#* code snippet is validated by such a downcast. In general, this may lead to an *InvalidCastException* at runtime, which prevents OWL datatype properties from being assigned invalid property values.

```
1 int i = [...];  
2 #MeetingRoom r = [...];  
3 r.#hasCapacity = (#xsd#byte) i;
```

#### 6.1.2.5 Ontological roles

In *Zhi#* programs, ontological individuals can be related to other individuals and XML data type values using an object-oriented notation. In contrast to authoritative type declarations of class members in statically typed programming languages, domain and range restrictions of OWL object properties are used to infer the types of the subject (i.e. host object) and object (i.e. property value) of the declared RDF triple in the ontology.

Hence, the types of the related individuals do not necessarily need to be subsumed by the domain and range concept descriptions of the used object property *before* the declaration. The only requirement here is that the related individuals are not declared to be in the extensions of concept descriptions that are disjoint with the object property's domain and range restrictions. This case is statically checked by the Zhi# compiler (see above).

Similarly, OWL datatype properties can only be used with elements of the value space of that XSD data type that was used for the datatype property range restriction (cf. Subsection 3.1.1). Consequently, the following assignment statement fails to type-check for a datatype property  $U$  with domain  $A$  and range  $xsd\#byte$  since the literal  $23.5$  is interpreted as a .NET floating point value (i.e.  $xsd\#double$ ) and the value spaces of the two primitive base types  $xsd\#decimal$  and  $xsd\#double$  are disjoint.

```
a.#U = 23.5;
```

Both for OWL object and non-functional OWL datatype properties the property assignment semantics are *additive*. The following assignment statement *adds* the individual B as a value for property  $R$  of individual A; it does not remove existing triples in the ontology model. Note: A and B are the individuals referred to by  $a$  and  $b$ , respectively.

```
a.#R = b;
```

Correspondingly, property access expressions yield arrays of individuals and arrays of XML data type values for OWL object properties and non-functional OWL datatype properties, respectively, since an individual may be related to more than one property value. Accordingly, the type of OWL object property and non-functional OWL datatype property access expressions is always an array type, where the base type is the range restriction of the property. Note that this typing rule is incomplete for OWL object properties since the related individuals may as well be in the extensions of a number of different concept descriptions than the property range restriction. Still, at compile time (i.e. on the meta-level with no knowledge about particular ontology models) no further conclusions are safe.

The type of an assignment to an OWL object property and a non-functional OWL datatype property is always an array type, too. This behavior is slightly different from the typical typing assumptions in programming languages. Because the assignment operator ( $=$ ) cannot be overloaded in .NET, after an assignment of the form  $x = y = z$  all three objects can be considered equal based on the applicable kind of equivalence (i.e. reference and value equality). The same is not always true for assignments to OWL properties considering the array ranks of the types of the involved objects. In the following cascaded assignment expression, the static type of the expression  $b.\#R = c$  is `Array Range(R)` because individual B may be related by role  $R$  to more individuals than only C. As a result, with the following assignment, individual A is related by role  $R$  to *all* individuals that are related to individual B by role  $R$  (A, B, and C are referred to by  $a$ ,  $b$ , and  $c$ ).

```
a.#R = b.#R = c ;
```

In the following ontology fragment, which extends the previous definitions in this subsection, the non-functional properties *hasParticipant* and *beginsAt* are introduced to relate events with their participants and starting times.

```
1 <rdf:RDF xmlns="http://chil.server.de/ontology" ...>
2   <owl:Class rdf:ID="Participant">
3     <rdfs:subClassOf><owl:Class rdf:ID="Person"/></rdfs:subClassOf>
4   </owl:Class>
5   <owl:ObjectProperty rdf:ID="hasParticipant">
6     <rdfs:domain rdf:resource="#Event"/>
7     <rdfs:range rdf:resource="#Participant"/>
8   </owl:ObjectProperty>
9   <owl:DatatypeProperty rdf:ID="beginsAt">
10    <rdfs:domain rdf:resource="#Event"/>
11    <rdfs:range rdf:resource="xsd:dateTime"/>
12  </owl:DatatypeProperty>
13 </rdf:RDF>
```

As shown in the Zhi# code snippet below, arrays of individuals and XML data type values can be assigned to OWL object properties and non-functional OWL datatype properties. As a result, every non-null array element is added as a property value of the specified subject (i.e. individual). As for assignments of single individuals, for OWL object properties the base type of the array type of the source object does not need to be in the extension of the property range restriction before the assignment (i.e. covariant subtyping). In the given example, *Persons* ALICE and BOB are inferred to be *Participants*. As can be seen, the property centric modeling features of the Web Ontology Language make it possible to add “members” of ontological individuals on a per instance basis. Thus, OWL properties facilitate *ad hoc relationships* between objects that may not have been foreseen when a concept was defined.

```

1 #Event e = [...];
2 #Person[] persons = new #Person[] {           ↔
3     new #Person("#ALICE"), new #Person("#BOB")};
4 e.#hasParticipant = persons;
5 #xsd#dateTime[] startTimes = new #xsd#dateTime[] {   ↔
6     new #xsd#dateTime("2008-06-27T13:00:00Z"),       ↔
7     new #xsd#dateTime("2008-07-07T13:00:00Z")};
8 e.#beginsAt = startTimes;

```

### 6.1.2.6 Cardinality restrictions

In OWL, cardinalities can be stated on a property with respect to a particular class. If a maximum cardinality of 1 is stated on a property with respect to a class, then any instance of that class will be related to at most one individual by that property. Since there is no *unique name assumption* in OWL, there may be more than one value also for functional OWL object properties. From this a reasoner will infer that these individuals are equivalent and refer to the same entities in the described world. Accordingly, in Zhi#, for functional OWL object properties the property assignment semantics are *additive*.

In contrast, assignments to functional OWL datatype properties, which relate an OWL individual with an XML data type value, *update* the property value of the specified individual. The value is added if no such property value has been added to the individual before. In the following ontology snippet, the OWL object property *takesPlaceIn* and the OWL datatype property *startsOnlyAt* are declared to be functional.

```

1 <rdf:RDF xmlns="http://chil.server.de/ontology" ...>
2   <owl:FunctionalProperty rdf:about="#takesPlaceIn">
3     <rdf:type rdf:resource="owl:ObjectProperty"/>
4     <rdfs:domain rdf:resource="#Event"/>
5     <rdfs:range rdf:resource="#Room"/>
6   </owl:FunctionalProperty>
7   <owl:FunctionalProperty rdf:ID="startsOnlyAt">
8     <rdf:type rdf:resource="owl:DatatypeProperty"/>
9     <rdfs:domain rdf:resource="#Event"/>
10    <rdfs:range rdf:resource="xsd:dateTime"/>
11  </owl:FunctionalProperty>
12 </rdf:RDF>

```

Tagging an OWL object property as functional does not limit the number of property values (there may as well be no property values at all). Instead, a number of values may be added for the OWL object property *takesPlaceIn*. As a result, based on the assignments in lines 2 and 3 in the following Zhi# code snippet the two individuals ROOM248 and MEETINGROOM can be inferred to be identical (i.e. refer to the same room in the real world). In Zhi#, when functional object properties are accessed no arbitrary choice is made between the identical property values but all related individuals are returned. In line 4, the array *rooms* will contain both ROOM248 and MEETINGROOM. Accordingly, for functional object properties the type of assignment and access expressions is an array type with the property range restriction as the base type (e.g., the type of the property access expression in line 4 is *#Room[]*).



```

1 #Event e = [...];
2 e.#takesPlaceIn = new #Room("#ROOM248");
3 e.#takesPlaceIn = new #Room("#MEETINGROOM");
4 #Room[] rooms = e.#takesPlaceIn;
5 e.#startsOnlyAt = new #xsd#dateTime("2008-06-27T13:00:00Z");
6 e.#startsOnlyAt = new #xsd#dateTime("2008-07-07T13:00:00Z");
7 #xsd#dateTime startTime = e.#startsOnlyAt;

```

In line 6, the only start time of the event under consideration is *updated* (i.e. the event starts only on July 7). The type of property access and property assignment expressions is always the property range restriction (i.e. a non-array type) for functional OWL datatype properties because XML data type values are not subject to ontological reasoning and cannot be inferred to be identical (e.g., the type of the property access expression in line 7 is *xsd#dateTime*).

### 6.1.2.7 Ontological concepts vs. frames

In general, neither domain nor range restrictions of OWL properties are authoritative. This is in contrast to frame languages, object-oriented programming languages, some primitive ontology management systems with limited reasoning capabilities, and some knowledge acquisition systems such as Protégé.

Marvin Minsky introduced frame languages in the field of Artificial Intelligence in the 1970s. In most frame-based knowledge representations inheritance is the central inference mechanism. In a general context, a frame is “something that has to be fulfilled”. In this sense object-oriented programming languages are frame languages.

In statically typed object-oriented programming languages such as C#, properties are declared as class members. The domain of a property corresponds to the type of the containing host object. Only instances of the domain type can have the declared property. The range of a property (i.e. class attribute) is also given by an explicit type declaration.

This type declaration is authoritative, too. All objects that are declared to be values of a property must be instances of the declared type at the time of the assignment.

Ontology management systems that are not able to exploit the differences between object and datatype properties in order to perform efficient reasoning over ontologies only accept object property values that are already in the extension of the range class of the object property *before* the property value statement is made (i.e. the property range restriction cannot be exploited to *infer* the type of the property value). Similarly, domain restrictions are interpreted like frames or class attributes in object-oriented programming languages (i.e. only objects that are in the extension of the property domain restriction can have values for this property).

The ontology editor Protégé partly ignores the effects of ontological reasoning, too. For an *Event* individual, the “frame slot” *takesPlaceInAuditorium* is omitted in the individual editor. For the *takesPlaceInAuditorium* property of a cognitive systems *Lecture* only individuals that are asserted or inferred to be in the extension of the property range restriction (i.e. the concept *LargeRoom*) *before* the assignment are made available in the GUI as shown in Fig. 6.2.

In contrast, in RDF and OWL it is not possible to ask “Which properties can be applied to resources of class *C*?”. Except for disjoint OWL concept descriptions, the RDF answer is “Any property”. Ontology engineers do not need to make premature commitments about all possible relationships that instances of concept *C* may have (in a statically typed object-oriented programming language such as C# this corresponds to using the top level type *System.Object* for the domain and range of a property). Instead, ontological individuals can be constructed in a prototype-style (which is not possible if the type of the property domain and range is *System.Object*). Still, many developers favor a rather frame-like composite view of classes and their associated properties. Indeed, the advantage of using property domain and range descriptions to constrain the set of conforming RDF triples is a more succinct structuring of an ontology or schema.

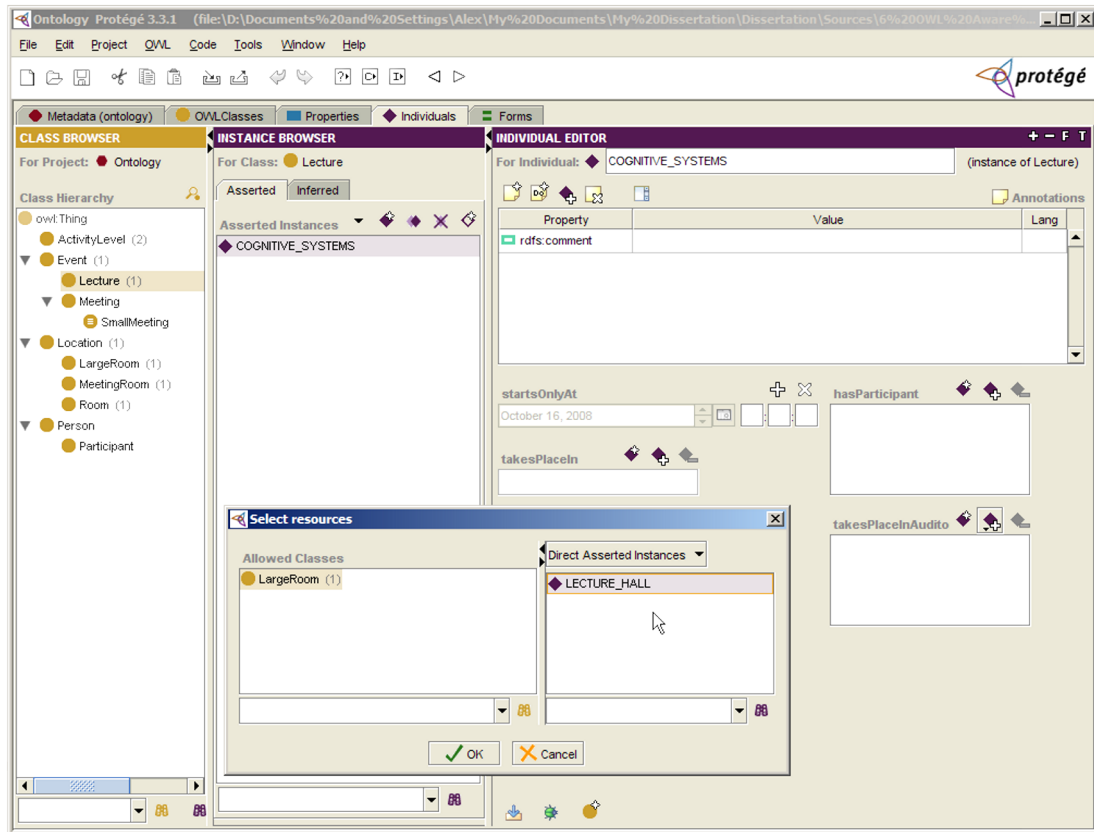


Figure 6.2: Protégé individual editor

In the Jena Semantic Web Framework, this requirement is taken into consideration by the *listDeclaredProperties()* method, which attempts to identify the properties that are intended to apply to instances of this class. The CHIL OWL API provides the *AddObjectPropertyValueChecked()* and *AddDatatypePropertyValueChecked()* methods, which check that 1) the specified subject is in the extension of the domain of the specified property and 2) the specified object is in the extension of the range of the specified property before the RDF triple is added to the ontology. In Zhi#, the *checked*-operator (see Subsection 6.1.5.1) can be used to support the frame-like notion of property domain and range restrictions. Checked property assignments throw an exception if the subject and object of the RDF triple, which is being declared, are not in the extension of the domain and range, respectively, of the used OWL object property.

### 6.1.2.8 Referential equality vs. ontological equality

In the C# programming language, the equality operator (`==`) can be used to test if its two operands are equal. For predefined value types, `==` returns true if the *values* of its two operands are equal. For reference types other than *System.String*, the equality operator returns true if its two operands *refer* to the same object. In the Zhi# programming language, the equality semantics of the `==` and `!=` operator for ontological individuals implements the notion of *ontological equality*.

In Zhi#, the equality operator (`==`) can thus be used to determine if two ontological individuals are identical (i.e. refer to the same entity in the described world). Ontological individuals are identical if they are explicitly declared to be the same (using the OWL *sameAs* feature) or if the reasoner infers their identity based on, for example, the use of several individuals as values of a functional object property as shown below.

```
1 #Event e = [ ... ];
2 #Room r1 = new #Room("#ROOM248 ");
3 #Room r2 = new #Room("#MEETINGROOM ");
4 bool b1 = r1 != r2; // b1 will be false since it is not known
5                       // that r1 and r2 refer to different rooms
6 e.#takesPlaceIn = r1;
7 e.#takesPlaceIn = r2;
8 bool b2 = r1 == r2; // b2 will be true because r1 and r2 refer to the same room
9                       // (i.e. an event only takes place in one room)
```

The inequality operator (`!=`) is implemented such that, it returns true if two individuals are *not* identical. Since in OWL there is no unique name assumption, `!=` returns true for ontological individuals if they are explicitly stated to be different using OWL's *differentFrom* or *AllDifferent* modeling features, or if the individuals each belong to pairwise disjoint class descriptions. Note that the inequality operator is *not* implemented as the logical negation of `==` as individuals can be unknown to be identical or different.

### 6.1.3 Auxiliary properties and methods

The Zhi# compiler framework supports a full-fledged object-oriented notation for external types. In particular, external compiler plug-ins can provide methods, properties, and indexers for static references and instances of external types. The OWL compiler plug-in implements a number of auxiliary properties and methods for ontological concepts and individuals in Zhi# programs.

In Zhi#, RDF triples of the form [*Subject Property Object*] can be added to an ontological knowledge base using an object-oriented property assignment notation. In order to remove property value assertions from the ontology, the OWL compiler plug-in provides the auxiliary methods *Remove* and *Clear* for OWL properties to remove one particular value and all values for an OWL property of the specified individual, respectively. In line 2 in the following code snippet the ontological individual B is removed as a property value for property *R* of individual A (i.e. the RDF statement [A *R* B] is removed from the knowledge base). In line 3 all property values for property *R* of individual A are removed (i.e. all RDF statements of the form [A *R* *x*] are removed).

```
1 #A a = new #A("#A");
2 a.#R.Remove(new #B("#B"));
3 a.#R.Clear();
```

In Zhi#, OWL properties also provide the *Exists* and *Count* properties. The *Exists* property yields true if any values are defined for the given OWL property for the given individual; otherwise false. The *Count* property yields the number of the defined property values for the given individual (i.e. for the following usage the number of RDF statements of the form [A *R* *x*]).

```
int i = a.#R.Count;
```

For static references of OWL concepts the auxiliary properties *Exists*, *Count*, and *Individuals* are defined. The *Exists* property yields true if individuals of the given type exist

in the ontology, otherwise false. *Count* returns the number of individuals in the extension of the specified concept description. *Individuals* yields an array of defined individuals of the given type. The *Individuals* property is generic in respect of the static type reference on which it is invoked. In the following array definition, the type of the property access expression *#Person.Individuals* is **Array Person** (and not **Array Thing**). Accordingly, it can be assigned to variable *persons* of type **Array Person**.

```
#Person [] persons = #Person.Individuals ;
```

Instances of OWL concepts provide the *Types* property, which yields a *System.String* array of the RDF types of the given individual. Note that this property is not meant to manually check the RDF type of an ontological individual as indicated in line 2 in the following code snippet. Preferably, the *is*-operator shall be used to check whether the RDF type of an ontological individual is compatible with a named concept description as shown in line 3. Also note that there is no type inference for ontological individuals based on the *is*-operator. This is different from the effect of the *is*-operator when used with, for example, XSD variables (cf. Subsection 5.1.3.6).

```
1 #Person p = [...];
2 if (Contains(p.Types, "#Participant")) { [...] };
3 if (p is #Participant) { [...] };
```

The *IdenticalIndividuals* property yields an array of individuals that are declared or inferred to be identical to the given individual (i.e. refer to the same entities in the described world). The *IdenticalIndividuals* property is generic in respect of the type of the individual on which it is invoked (i.e. the base type of the returned array is the declared type of the host individual). In the following array definition, the type of the property access expression *p.Individuals* is **Array Person** (and not **Array Thing**). Accordingly, it can be assigned to variable *persons* of type **Array Person**.

```
1 #Person p = [...];
2 #Person [] persons = p.Individuals ;
```

#### 6.1.4 Compilation to C#

Zhi# expressions that comprise elements of an ontology are compiled into conventional C# code. The Zhi# runtime library was extended with OWL specific functionality in order to keep the generated C# code simple (e.g., C# code for declaring an ontological individual is not reproduced for each declaration but exists only once in the Zhi# runtime library). The single proxy class *OWLIndividual* is substituted for ontological concept names in Zhi# programs. The only class member of *OWLIndividual* is a read only *System.String* field *Name*, which references the actual ontological individual in the knowledge base. Because different OWL concept names are replaced by one single proxy class (i.e. the presented solution is a language feature and not yet a feature of the runtime) the same restrictions apply for OWL concepts in Zhi# programs as for generic types in the Java programming language: OWL concepts cannot be used for method and operator overloading.

Ontological individuals can be downcast and cross cast to OWL concepts. Cast expressions are translated into calls of the Zhi# runtime library method *CastOWLIndividual*, which checks dynamically whether the ontological individual referred to by the *OWLIndividual* source object is in the extension of the target type of the cast expression (i.e. the cast is performed in the context of the ontology and not based on types of the programming language). The following cast expression is compiled into the C# code shown below. The *AssertKindOf* method facilitates a dynamic type check on the provided object (see next subsection). For the sake of brevity, in the following code snippets qualifying aliases and namespaces of Zhi# runtime library methods and ontology elements are omitted.

```
1 #B b = [...];
```

```
2 #A a = (#A) b;
```

```
1 OWLIndividual b = [...];
```

```
2 OWLIndividual a = CastOWLIndividual( AssertKindOf(b, "#B"), "#A");
```

The object-oriented notation that allows access of and assignments to ontological properties of ontological individuals is translated into invocations of Zhi# runtime library

methods for getting and setting property values of ontological individuals in the knowledge base. The following assignment of individual B as a value for property *R* of individual A (i.e. the declaration of the RDF statement [A *R* B]) is replaced by an invocation of the *AddOWLObjectPropertyValues* method as shown below. The last parameter of the *AddOWLObjectPropertyValues* method (i.e. the property values parameter) is declared with the C# *params* keyword. Hence, the argument can be both a single *OWLIndividual* object, a number of *OWLIndividual* objects, as well as an *OWLIndividual* array.

```
a.#R = b;
```

```
1 AddOWLObjectPropertyValues(                                     ↪
2     AssertKindOf(a, "#A"), "#R", AssertKindOf(b, "#B"));
```

In contrast to declarations of OWL object properties and non-functional OWL datatype properties, assignments to functional OWL datatype properties update existing property value statements in the knowledge base and are thus substituted by invocations of the *SetOWLDatatypePropertyValue* method. Assignments to OWL object properties and non-functional OWL datatype properties are additive while assignments to functional OWL datatype properties update an existing property value. In order to allow for cascaded assignments to OWL properties of ontological individuals, the *AddOWLObjectPropertyValues*, the *AddOWLDatatypePropertyValue*, and the *SetOWLDatatypePropertyValue* methods return the value(s) that are declared for the specified property of the specified individual as an array of *OWLIndividual* objects, an *RTSimpleType* array, and an *RTSimpleType* object, respectively. Note that this requires a query on the knowledge base, which determines – for some ontology management systems – the latest point in the execution of a *Zhi#* program when a knowledge base can be reported to be inconsistent and the *Zhi#* program aborts with an exception (see next subsection).

Usages of the auxiliary properties and methods of static OWL concept references, ontological roles, and ontological individuals in *Zhi#* programs are translated into multi-argument function calls on the *Zhi#* runtime library as follows.



$$o.R[.p][= v] \xrightarrow{\text{is translated into}} f_{R/p}(o, R[, p][, v])$$

In C#, so called indexers can be declared for classes and structs in order to index a class or struct in the same way as an array. In Zhi#, indexers can return enumerators that read data in collections of .NET objects and ontological individuals. Because ontological individuals are subject to dynamic type checks, the elements of a collection must not be returned directly as with the use of a conventional *IEnumerator* object. Instead, *foreach*-statements that operate on user defined class or struct indexers, which return collections of ontological individuals, are compiled into invocations of the *ForEachOWL* method of the Zhi# runtime library. The following user defined indexer returns an enumerator that reads from an array of ontological individuals.

```

1 public class T : IEnumerable {
2     public IEnumerator GetEnumerator() {
3         ArrayList l = new ArrayList();
4         l.Add(new #C("#c1"));
5         l.Add(new #C("#c2"));
6         return l.GetEnumerator();
7     }}

```

The following *foreach*-statement is transformed as shown below.

```

1 T t = new T();
2 foreach (#C c in t) { [...] }

```

```

1 T t = new T();
2 foreach (OWLIndividual c in ForEachOWL("#C", t)) { [...] }

```

The *ForEachOWL* method implements a coroutine, which subsequently yields the elements (i.e. ontological individuals) of a specified *IEnumerable* object (i.e. object *t* in the above example). Each ontological individual is dynamically type checked to be in the extension of the given concept (i.e. *C* in the given example) before it is returned.

A complete list of OWL related program transformations is given in Appendix B.2.

### 6.1.5 Dynamic checking

In a statically typed programming language such as C# the possible types of an object are known at compile time. Hence, typing errors can be caught early at compile time (i.e. fail-fast) as opposed to runtime, which allows for efficient execution at runtime. Unfortunately, the non-contextual property centric data modeling features of the Web Ontology Language render static type checking only a partial test on Zhi# programs. As a result, the OWL plug-in for the Zhi# compiler framework and the Zhi# runtime library facilitate dynamic checking of ontological knowledge bases.

Ontological individuals can be in the extensions of a number of different concept descriptions, which do not necessarily subsume each other and which cannot completely be predicted at compile time (i.e. on the meta-level). In the same way, explicitly made RDF type assertions may be inconsistent with particular property values or the number of values for a particular property of an individual. More severely, ontological knowledge bases are subject to concurrent modifications via interfaces of different levels of abstraction (e.g., RDF triples, logical concept view).

The Zhi# solution for dynamic checking of ontological knowledge bases is as follows: In Zhi# programs, declarations of ontological individuals do not guarantee that the individual will ever be in the extension of the provided concept description and the knowledge base to be consistent. Instead, before each single usage of an individual 1) the individual is dynamically checked to be in the extension of the declared concept and 2) the knowledge base is checked to be consistent; an exception is thrown if either is not the case.

The following ontology defines a small meeting as a meeting with at most three participants. In line 1 in the Zhi# code snippet shown below the ontological individual M is declared as a *SmallMeeting*. This, however, does not prevent the declaration of more than three values for the property *hasParticipant* of individual M. These declarations can be made by the Zhi# program itself as shown in line 4 or by different knowledge base clients, which may, for example, directly modify the RDF triple store.

```

1 <rdf:RDF xmlns="http://chil.server.de/ontology" ...>
2   <owl:Class rdf:ID="Person"/>
3   <owl:ObjectProperty rdf:about="#hasParticipant">
4     <rdfs:domain rdf:resource="#Meeting"/>
5     <rdfs:range rdf:resource="#Person"/>
6   </owl:ObjectProperty>
7   <owl:Class rdf:ID="SmallMeeting">
8     <owl:equivalentClass>
9       <owl:Class>
10        <owl:intersectionOf rdf:parseType="Collection">
11          <owl:Restriction>
12            <owl:maxCardinality rdf:datatype="xsd:int">3
13          </owl:maxCardinality>
14          <owl:onProperty>
15            <owl:ObjectProperty rdf:ID="hasParticipant"/>
16          </owl:onProperty>
17        </owl:Restriction>
18      <owl:Class rdf:ID="Meeting"/>
19    </owl:intersectionOf>
20  </owl:Class>
21 </owl:equivalentClass>
22 </rdf:RDF>

```

```

1 #SmallMeeting m = new #SmallMeeting("#M");
2 #Person p1 = [...], p2 = [...], p3 = [...],
3   p4 = [...], p5 = [...], p6 = [...];
4 m.#hasParticipant = new #Person []{p1, p2, p3, p4, p5, p6};
5 #Person [] persons = m.#hasParticipant;

```

Note that based on the above example ontology it is admissible to relate an arbitrary number of *Persons* with *SmallMeeting* *M*. Since OWL does not make the unique name

assumption, the related individuals will be assumed to somehow refer to the same three *Persons* in the described world. The ontology will, however, be in an inconsistent state if M is related with more than three *different Persons*. In the ontology, individuals can be declared to be mutually distinct using OWL's *differentFrom* and *AllDifferent* language constructs. In *Zhi#*, the same statements can be made by calling the auxiliary *IsDifferentFrom* method with an individual or an array of individuals on M.

If the *Zhi#* program itself modifies the knowledge base after the declaration of individual M in line 1 inconsistencies between the declared type of M and the number of declared values for its property *hasParticipant* are reported in course of the assignment of the array of *Persons* in line 4. Early error reporting (i.e. fail-fast) is worth having here for two reasons. First, the program does not go on to operate on inconsistent data (e.g., for an inconsistent ontology one cannot assume that a small meeting has at most three participants). Second, the *Zhi#* program itself can contain exception handlers, which undo the modifications that presumably caused the inconsistency (e.g., remove the values for the *hasParticipant* property of individual M).

In the above *Zhi#* program the assignment of the *Persons* array in line 4 appears to be an atomic operation. In fact, the actual property value declarations in the knowledge base are executed sequentially for each contained variable in the array (i.e.  $p1, \dots, p6$ ). Under the assumption that *Persons* P1, ..., P6 have been declared to be different from each other the knowledge base is in an inconsistent state already after the assignment of variable  $p4$ . In such cases the fail-fast design principle takes precedence over the apparent atomicity of the assignment statement. This happens for two reasons. Firstly, using some simple logging mechanisms to, for example, record the individuals that have already been added as property values, it is likely to be easier to restore the knowledge base to a consistent state. Secondly, some ontology management systems such as the Pellet OWL DL Reasoner [Pel06] completely cease to process inconsistent ontologies on a concept view level, which makes it impossible to add further property values after an inconsistent state has been reached.

If the knowledge base is modified (on the RDF triple level) by a third party the inconsistency between the declared type of individual *M* and the number of property values is at the latest detected right before the usage of individual *M* in line 5.

In compiled *Zhi#* programs all references to ontological individuals are substituted by invocations of the *AssertKindOf* method, which is provided by the OWL component of the *Zhi#* runtime library. The *AssertKindOf* method takes as input an *OWLIndividual* object and a number of *strings* that denote OWL concept names. The ontological individual referred to by the specified *OWLIndividual* object is checked to be in the extensions of the given concept descriptions. The input *OWLIndividual* object is returned if this type check succeeds; an exception is thrown if it fails. The type check fails if the individual is not subsumed by the given concept descriptions or the ontology is inconsistent for other reasons. Inconsistent ontologies cause failure since the reasoning results (e.g., the inferred RDF types of the checked individual) may be incorrect or even be unavailable from the ontology management system.

The *Zhi#* program above is compiled into *C#* as follows. Note the invocations of the *AssertKindOf* method for each reference of an ontological individual. Recurring definitions and usages of variables *p2*, ..., *p6*, qualifying aliases and namespaces of *Zhi#* runtime library methods and types, and qualifying namespaces of ontological concepts, roles and individuals are omitted for brevity.

```

1 OWLIndividual m = CreateOWLIndividualByName("#M", "#SmallMeeting");
2 OWLIndividual p1 = AssertKindOf([...], "#Person"), [...];
3 AddOWLObjectPropertyValues(                                     ↔
4     AssertKindOf(m, "#SmallMeeting"),                          ↔
5     "#hasParticipant",                                         ↔
6     new OWLIndividual[] { AssertKindOf(p1, "#Person"), [...]});
7 OWLIndividual[] persons =                                     ↔
8     GetOWLObjectPropertyValues(                                 ↔
9     AssertKindOf(m, "#SmallMeeting"), "#hasParticipant");

```

An important form of dynamic typing is “duck typing” in which only the considered methods and properties of an object are considered rather than its inheritance from a particular class. Zhi#’s dynamic type checking of OWL individuals is different to “duck typing” since not that part of a type’s structure that is accessed is checked but rather the declared type of the object. As a consequence, typing errors at runtime are detected not when a particular object property is accessed but when the object itself is referenced. The following “duck typed” pseudo code program will run properly if the object  $a$  has a property  $R$ ; it will fail otherwise. Due to the lack of type declarations no particular types are required for objects  $a$  and  $b$ . Instead, only the used aspects of  $a$  are considered, which allows for the dynamism in programming languages such as Perl and Python.

```
1 var a = [ . . . ];
2 var b = a.R;
```

In contrast, Zhi#’s dynamic type checking enforces the declared types at runtime while the referenced property may be undefined for the considered individual. Given the ontological concepts  $A$  and  $B$ , and the OWL object property  $R$ , which relates  $A$ s with  $B$ s, the following typing constraints are enforced for the Zhi# statements shown below. First, the Zhi# compiler statically checks that the type of the lvalue in line 2 is an array type with a base type that subsumes the property range restriction of property  $R$ . At runtime, the individual referred to by  $a$  must be an  $A$  (i.e. be in the extension of the concept description named  $A$ ) and the individual referred to by  $b$  must be a  $B$ . On the other hand, Zhi#’s dynamic type checking is negligent in terms of the number of defined values for property  $R$ . In particular, if no values are defined for property  $R$  of the individual referred to by  $a$  an empty array of individuals and data type values is returned for OWL object properties and non-functional OWL datatype properties, respectively. A *null* reference is returned if no value is declared for functional OWL datatype properties.

```
1 #A a = [ . . . ];
2 #B [] b = a.#R;
```

One of the main reasons for null pointer exceptions in object-oriented programming languages are mutually referential data structures, which are not initialized properly in constructors. This issue is apparently evident for ontological data, too, since OWL DL allows for concept constructors that include constraints such as cardinalities of ontological properties or the existence of property values of a particular type. Still, the Web Ontology Language allows two different interpretations of an apparent inconsistency between the RDF type of an individual and its declared property values.

On the one hand, in OWL, individuals can be declared to be of a certain type even if required property values do not exist for the individual. This issue is resolved by OWL's *open world assumption* (OWA) in which no single observer has complete knowledge (and therefore cannot make the *closed world assumption*). Assume, for example, an ontological TBox that includes two atomic concepts  $A$  and  $B$ , role  $R$ , and the complex concept description  $C \equiv \exists R.B$ . Under the closed world assumption it is inadmissible to declare an instance of concept  $C$  with no value of type  $B$  for property  $R$ . Under OWA, the required property value can be assumed to exist but to be unknown for the current observer. From the lack of knowledge of a value for property  $R$  of type  $B$  a reasoner cannot conclude that individual  $c$  is not a  $C$ . The other way around one cannot conclude from the asserted type of an individual that particular property values exist. Consequently, the following isolated definition is a valid Zhi# statement.

```
#C c = new #C("#C");
```

On the other hand, in OWL, the asserted RDF types of an individual can be complemented by inferred types based on the individual's particular properties. In the current example one may therefore preferably create a  $C$  starting from an  $A$ , which is successively initialized, as follows.

```
1 #A a = new #A("#C");
2 a.#R = new #B("#B");
3 #C c = (#C) a;
```

After the declaration and initialization of the ontological individual `C` in line 1 and 2, respectively, the downcast of variable `a` to concept `C` is effectively a dynamic type check that individual `C` is in the extension of concept `C` based on its particular properties. Note that in an environment where the ontological knowledge base is concurrently modified a different knowledge base client may assert that individual `C` is a `C` – without `C` having any property values. Again, in contrast to frames, because of the OWA it is inadmissible to presume from the RDF type of an individual the existence of particular property values. It is, however, allowed to presume the non-existence of property values in a consistent knowledge base for concept descriptions that comprise a maximum cardinality restriction (e.g., a small meeting that is defined as a meeting with at most three participants can only have at most three different participants in a consistent knowledge base).

### 6.1.5.1 Checked property assignments

In `C#`, the *checked*-keyword can be used to control the overflow-checking context for integral-type arithmetic operations and conversions. It can be used as an operator or a statement. In a checked context, if an expression produces a value that is outside of the range of the destination type, the result may be a compile-time error or an exception at runtime depending on whether the expression is constant or non-constant.

In a similar manner, the *checked*-keyword can be used in `Zhi#` to support the frame-like notion of OWL object properties. The following example demonstrates the *checked*-operator on an OWL object property assignment expression.

```

1 #Event e = [...];
2 #Location lectureHall = [...];
3 e.#takesPlaceInAuditorium = checked(lectureHall);

```

The OWL object property *takesPlaceInAuditorium* (see Subsection 6.1.2.3) relates *Lectures* with *LargeRooms* (i.e. the subject of the RDF statement must be a *Lecture* and the object of the statement must be a *LargeRoom*). Usually, an OWL DL reasoner will



use these domain and range restrictions to infer the RDF types of the related individuals. If, however, concepts and roles are interpreted as frames and frame slots, respectively, the used subjects and objects must be of the given types already before the declaration of the RDF triples that relate them by the *takesPlaceInAuditorium* role. Exactly this frame-like behavior is enforced by the *checked*-operator in the code snippet above. At runtime, an exception is thrown since the individual referred to by variable *e* is not in the extension of the domain restriction of property *takesPlaceInAuditorium* (i.e. *Lecture*). Similarly, an error is raised if the individual referred to by variable *lectureHall* is not in the extension of the range restriction of property *takesPlaceInAuditorium* (i.e. *LargeRoom*). Note that the *checked*-operator does not influence the static type checks that apply, for example, to disjoint concept definitions.

By default, assignments of XML data type values to OWL datatype properties are executed in a *checked* context, where incompatible property values are detected at compile time.

## 6.2 Example Scenario Implementation

Based on the TBox and ABox of the example scenario described in Subsection 1.3.7 an OWL DL knowledge base can be queried and modified in *Zhi#* as shown in the following code snippet. In line 1 and 2, primitive built-in XML data types from the XSD namespace and elements from the CHIL ontology, respectively, are made public in the *Zhi#* program using the keyword *import*. The referenced OWL concepts, roles, and XML data types are statically checked to exist in the imported namespaces. The string literal in line 8 is checked at compile time, too, to denote a valid *xsd#dateTime* value. The *getFirstStartingTime* method in line 22 takes as input an *Event* and returns the first *xsd#dateTime* the given *Event* occurs. The *foreach*-statement in line 14 benefits from ontological reasoning in two ways. Firstly, *Meetings* such as the one referred to by variable *m* are also classified as *Events*. Secondly, *ROOM248* *hosts* the *PROJECTMEETING*

because the ontological role *hosts* was declared as the inverse of role *takesPlaceIn*. RDF type checks do not have to be devised manually but are facilitated by the *is*-operator as shown in line 19. See Section 7.2 for how OWL DL concept constructors would have to be manually implemented in conventional C#.

```

1 import XML xsd = http://www.w3.org/2001/XMLSchema;
2 import OWL cho = http://chil.server.de/ontology;
3 namespace MyCHILApplication {
4     class MyClass {
5         public void f() {
6             #cho#Meeting m =                                     ↔
7                 new #cho#Meeting("http://chil.server.de/ontology#PROJECTMEETING");
8             #xsd#dateTime dt = "2008-07-07T13:00:00";
9             m.#cho#scheduledAt = dt;
10            #xsd#dateTime firstTime = getFirstStartingTime(m);
11            Console.WriteLine("First meeting at " + (string) firstTime + "!");
12            #cho#Room r =                                       ↔
13                new #cho#Room("http://chil.server.de/ontology#ROOM248");
14            foreach (#cho#Event e in r.#cho#hosts) {
15                Console.WriteLine("Room " + r + " hosts event " + e + "!");
16            }
17            #cho#Person alice =                                   ↔
18                new #cho#Person("http://chil.server.de/ontology#ALICE");
19            if (alice is #cho#Moderator) {
20                Console.WriteLine("Alice is a moderator!");
21            }
22            public static #xsd#dateTime getFirstStartingTime(#cho#Event e) {
23                [...]
24            }
25        }
    }
}

```

### 6.3 OWL and XSD

Recently, several approaches have emerged to refine on the integration of external type systems such as XML Schema Definition and the Web Ontology Language. Pan and Horrocks presented an extension of OWL DL, called OWL-Eu [PH06], which allows for class descriptions based on (customized) data types. Especially, customized data types can be used in datatype exists restrictions ( $\exists T.D$ ) and datatype value restrictions ( $\forall T.D$ ). It can be imagined to use the Zhi# compiler framework with an OWL-Eu API since Zhi#'s XSD compiler plug-in already provides type checking and type inference capabilities that are needed to cope well with OWL-Eu's data type expressions. Zhi#'s OWL compiler plug-in would have to be extended in order to track local range restrictions that can be inferred for properties of individuals that are of a particular type. Given complex concept descriptions the OWL type checking component could augment the type information of ontological roles accordingly.

### 6.4 Integration of the CHIL OWL API

The OWL plug-in for the Zhi# compiler framework provides a number of properties and methods that can be accessed on ontological individuals, roles, and static concept references (see Subsection 6.1.3). These auxiliary properties and methods complement inherent language features to create concept instances and assign property values with functionality to, for example, delete statements from a knowledge base. In addition, conventional ontology management systems such as the CHIL Knowledge Base Server may be utilized in Zhi# programs. In the following code snippet an ontological individual A is created in the managed knowledge base. Thereafter, the CHIL OWL API<sup>1</sup> is used to connect to the CHIL Knowledge Base Server, create an individual B, and query the RDF representation of the ontological knowledge base. Note that the *IOWLAPI* object

---

<sup>1</sup>The C# interface to the CHIL Knowledge Base Server is defined in the *Zhimantic.OWL.API* namespace.

must be configured identically to the OWL plug-in of the Zhi# runtime library in order to connect to the same instance of the CHIL Knowledge Base Server.

```
1 using System.IO;
2 using Zhimantic.OWL.API;
3 import OWL ont = http://www.zhimantic.com/eval;
4 namespace ProgramNamespace {
5     class Program {
6         static void Main() {
7             // Using language inherent support for OWL
8             #ont#A a = new #ont#A("#A");
9             // Using the CHIL OWL API
10            IOWLAPI api = new OWLAPI("localhost", 2342);
11            api.AddIndividual("http://www.zhimantic.com/eval#B",           ↔
12                            "http://www.zhimantic.com/eval#B");
13            using (TextWriter tw = new StreamWriter(@"KB.rdf")) {
14                tw.WriteLine(api.GetRDFXML(true));
15            }
16        }
17    }
18 }
```

One may argue that the concurrent use of Zhi#'s language features and API-based access to the managed knowledge base undermines the static type checking of Zhi# programs. This is, however, only true for those portions of the Zhi# programs that are actually API-based (e.g., a typo in line 12 may cause an *UndeclaredConceptDescription* at runtime). The type checking of Zhi#-specific code to manipulate ontological knowledge bases *always* has to take into account concurrent knowledge base access, where it does not matter if the API-based modifications originate from the same program or third parties. In any case, it would be desirable to have most parts of an OWL API being provided as auxiliary fields and methods of ontological individuals, roles, and concept references in Zhi# programs as this would allow for type checking on the ontology level. On the other hand, it does of course not appear reasonable to use an auxiliary method to query

the RDF serialization of a knowledge based since this kind of functionality can hardly be related to one particular ontology element.

Appendix H presents a mapping of the CHIL OWL API to auxiliary properties and methods of ontology elements in Zhi# programs (only the most indispensable methods were implemented in the current version of the OWL plug-in for the Zhi# compiler framework). For each API method, eliminated exceptions, which are stipulated by the given method preconditions, are given that may occur at runtime with API-based access but could be statically checked for auxiliary properties and methods. In total, the functionality of 50 CHIL OWL API methods may reasonably be provided by the Zhi# programming language, which would facilitate static checking of 91 defined method preconditions.

## 6.5 Related Work

To the best of the author's knowledge there have been no substantial approaches to devise programming language inherent support for the Web Ontology Language. Still, the problem to provide a framework for compile-time support and dynamic type checking for OWL DL ontologies stems from practice.

In the CHIL research project [Inf04], which aims to introduce computers into a loop of humans interacting with humans, rather than condemning a human to operate in a loop of computers, a semantic middleware has been developed that fusions information provided by so called *perceptual components* in meaningful ways. Each perceptual component (e.g., image and speech recognizers, body trackers, etc.) contributes to the common domain of discourse. The Web Ontology Language OWL DL was decidedly used to replace previous domain models that had been based on particular programming languages. A major disadvantage of using an OWL API compared to previously used Java-based domain models had been the lack of type checking for ontological individuals. This lack of compile-time support has lead to the development of code generation tools such as the Ontology Bean Generator [Pro07] for the Java Agent Development Framework [Tel07], which generates

proxy classes in order to represent elements of an ontology. Similarly, Kalyanpur et al. [KPB04] devised an automatic mapping of particular elements of an OWL ontology to Java code. Although carefully engineered the main shortcomings of this implementation are the blown up Java class hierarchy and the lack of a concurrently accessible ontological knowledge base at runtime (i.e. the “knowledge base” is only available in one particular Java virtual machine in the form of instances of automatically generated Java classes). This separation of the ontology definition from the reasoning engine results in a lack of available ABox reasoning (e.g., type inference based on nominals).

The two latter problems were circumvented by the RDFReactor approach [V06] where a Java API for processing RDF data is automatically generated from an RDF schema. While this may still result in a significant number of automatically generated Java classes, the generated API operates on one single external RDF model, which may be handled by existing ontology management systems. Still, since RDFReactor operates on RDF triples, obsolete generated code may result in meaningless or invalid modifications of the triple store. Also, RDFReactor only provides a frame-like view of OWL ontologies.

In stark contrast to these systems, the Zhi# programming language syntactically integrates OWL concepts and properties with the C# programming language using conventional object-oriented notation. Also, Zhi# provides static type checking for atomic XML data types, which may be the range of OWL datatype properties, while many ontology management systems – not to mention the above approaches – simply discard range restrictions of OWL datatype properties. A combination of static typing and dynamic checking is used for ontological concept descriptions. In contrast to static type checking that is based on generated proxy classes, Zhi#'s OWL compiler plug-in considers disjoint concept descriptions and copes well with multiple inheritance.

In the above systems, which introduce proxy classes of OWL concepts, instances of these proxy classes are simply instantiated in the context of the runtime environment of the programming language. In Zhi#, ontological individuals are actually created in the

ontological knowledge base and are thus subject to ontological ABox reasoning. Moreover, the number of introduced proxy classes is constant and does not depend on the number of imported XML data types and OWL concept descriptions.

Koide and Takeda [KT06] implemented an OWL reasoner for the  $\mathcal{FL}_0$  Description Logic on top of the Common Lisp Object System [DG87] by means of the Meta-Object Protocol [KRB91]. Their implementation of the used structural subsumption algorithm [BCM03] is described, however, to yield only incomplete results.

The *Zhi#* compiler framework does not impose particular type checking strategies on its compiler plug-ins. *C#* is a statically typed programming language. *Zhi#*'s compiler plug-in for XML Schema Definition implements static type checking for constrained atomic types, too. The non-contextual property-centric modeling features of OWL, however, render static type checking only a partial test on *Zhi#* program text that includes ontological concepts and properties.

The representation and the type checking of ontological individuals in *Zhi#* is similar to the type *Dynamic*, which was introduced by Abadi et al. [ACP89, ACP91]. Values of type *Dynamic* are pairs of a value  $v$  and a type tag  $T$ , where  $v$  has the type denoted by  $T$ . The result of evaluating the expression *dynamic*  $e:T$  is a pair of a value  $v$  and a type tag  $T$ , where  $v$  is the result of evaluating  $e$ . The expression *dynamic*  $e:T$  has type *Dynamic* if  $e$  has type  $T$ . *Zhi#*'s dynamic type checking of ontological individuals corresponds to the *typecase* construct as proposed by Abadi et al. in order to inspect the type tag of a given *Dynamic*. An object (i.e. ontological individual)  $x$  can be checked to be a *SmallMeeting* using the *typecase* construct as follows.

$\lambda x : \textit{Dynamic}.$

*typecase*  $x$  of

$(sm : \textit{SmallMeeting})$  *use*( $sm$ )

*else throw type checking exception*

*end*

In Zhi# source programs, the use of OWL concept names corresponds to explicit *dynamic* constructs. In compiled Zhi# code, invocations of the *AssertKindOf* method of the Zhi# runtime correspond to explicit *typecase* constructs.

Abadi et al. obtained a soundness theorem for a  $\lambda$ -calculus based language that includes the proposed *dynamic* and *typecase* constructs. In contrast to the OWL type checking in Zhi# their *dynamic* expressions can even cause the creation of new tags at runtime (e.g., a function may take a dynamic value and return a *Dynamic* whose value part is a pair, both of whose components are equal to the value part of the original dynamic value).

Thatte described a “quasi-static” type system [Tha90], where explicit *dynamic* and *typecase* constructs are replaced by implicit coercions and runtime checks. A static type checking phase is followed by a *plausibility* checking phase, where dynamic type errors are statically detected. Though there is no extra plausibility checking phase in Zhi#, a typing error of this kind that is also detected in Zhi# is the use of an ontological individual as a property value, where the range description of the property and the type of the individual are disjoint (i.e. the dynamic type error can be statically detected). A runtime check in Thatte’s work is unplausible if the known type of the expression being checked and the type required by the check do not have a common subtype (which does not apply to OWL concept descriptions). As in Thatte’s work, Zhi#’s dynamic typing for OWL detects errors as early as possible to make it easy to find the programming error that led to the type error. Abadi et al. and Thatte’s dynamic types were only embedded with a simple  $\lambda$ -calculus. The same is true for recent *gradual typing* proposals [ST06]. Tobin-Hochstadt and Felleisen developed the notion of *occurrence typing* and implemented a Typed Scheme [TF08]. Occurrence typing assigns distinct subtypes of a parameter to distinct occurrences, depending on the control flow of the program. Such distinctions are not made by Zhi#’s OWL compiler plug-in since it is hard to imagine that appropriate subtypes can be computed considering complex OWL concept descriptions.



## 6.6 Summary

The author devised an OWL DL plug-in for the Zhi# compiler framework to provide a combination of static typing and dynamic checking for ontological concept descriptions. Used concept descriptions and role names are checked at compile time to exist in the referenced ontology. Static type checks include disjoint concepts, and cardinalities and disjoint property domain and range restrictions of ontological roles. Property domain and range restrictions can be used for ontological reasoning or the *checked*-operator can be used to enforce frame-like semantics of OWL concepts and roles. The declared RDF types of ontological individuals in Zhi# programs are dynamically checked at runtime.

Auxiliary properties were implemented for ontological individuals, roles, and static concept references in Zhi# programs in order to make it particularly easy to, for example, get all individuals in the extension of a specified concept description or to get all RDF types of a specified individual.

Ontological reasoning was embedded with C# programming language features such as the *is*-operator, which can be used for dynamic RDF type checks of ontological individuals. OWL concept descriptions can be used for formal parameters of methods, user defined operators, and indexers.

The compiler plug-in for the Web Ontology Language can be used cooperatively with the plug-in for XML Schema Definition (see Chapter 5) in order to support OWL datatype properties. Zhi# code that uses elements of an ontology is compiled into conventional C#. In the current implementation, the OWL component of the Zhi# runtime library utilizes the CHIL OWL API (see Chapter 4) in order to manage the ontological knowledge base. Eventually, the OWL DL plug-in reduces the dependency on particular OWL APIs. Assuming an identical behavior to the required fragment of the CHIL OWL API both the used ontology management system and the used OWL API can be substituted in the Zhi# runtime library without recompilation of Zhi# programs.



## CHAPTER 7

### Validation and Evaluation

In common usage, *validation* is the process of checking a thesis, a plan, or a solution approach with regard to a particular problem. As such, validation confirms that the needs of a user of a product are met. Similarly, *evaluation* is the systematic determination of a solution's merits, worth, and significance based on a set of criteria. In this chapter, the suitability and usefulness of the Zhi# approach will be shown on a technical level and on an application level.

The next section provides basic software metrics of the Zhi# compiler framework, its XSD and OWL compiler plug-ins, the Zhi# Eclipse-based frontend, and the CHIL Knowledge Base Server in order to summarize the technical foundation of the presented solution approach and to indicate its adequacy, dependability, and extensibility.

The following evaluation section will show the usefulness of the Zhi# approach for the so-far considered XSD and OWL application domains on a microscopic level. For particular XSD constraining facets, OWL concept constructors, and OWL role restrictions Zhi# code will be presented that is inherently less error-prone and more intuitive than conventional C# code implementing the same modeling features.

Finally, the prospective applicability of the Zhi# compiler framework to additional technologies and its usefulness for the development of full-fledged knowledge-based applications will be shown. In particular, it will be demonstrated how Object Constraint Language (OCL) expressions could be translated into Zhi# code in order to facilitate static checking of OCL invariants. It will also briefly be indicated how the Zhi# approach could be further evaluated by means of case studies.

## 7.1 Technical Validation

The complete Zhi# tool suite including the compiler framework, the compiler plug-ins for XML Schema Definition and the Web Ontology Language, the Eclipse-based frontend, and the  $\lambda_C$ -calculus and ontology management back ends totals 142,039 lines of code (LOC). As shown in Fig. 7.1 the Zhi# compiler amounts to 93,683 LOC, the CHIL Knowledge Base Server to 25,496 LOC, the implementation of the  $\lambda_C$ -calculus to 17,295 LOC, and the Eclipse-based frontend to 9,019 LOC.

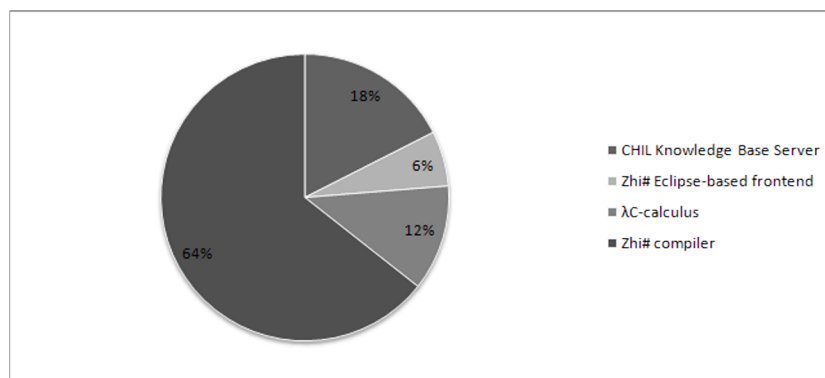


Figure 7.1: Code base size

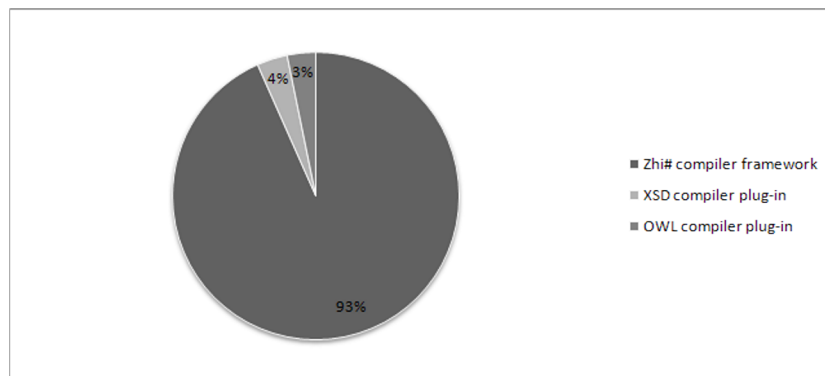


Figure 7.2: Zhi# compiler code base size

The Zhi# compiler comprises the extensible framework plus two compiler plug-ins for XML Schema Definition and Web Ontology Language specific language extensions. As shown in Fig. 7.2 the framework amounts to 87,435 LOC. The XSD and OWL plug-ins

total 3,227 and 3,021 LOC, respectively. It is conceivable that the code size of compiler plug-ins would be larger and the code be more dispersed if the compiler framework would not expose two well defined extension points (see Section 2.3).

Fig. 7.3 shows the number of C#/Java classes and a code complexity value for each component of the Zhi# tool suite. The given code complexity was computed using Camwood Software's SourceMonitor tool [Cam09]. LOC numbers were confirmed using JLS-Soft's SourceCode Counter [JLS09]. The complexity metric is counted approximately as defined by Steve McConnell in his seminal book *Code Complete* [McC04]. The complexity metric measures the number of execution paths through a method where each method has an initial complexity value of one. This value increases by one for each control flow statement (e.g., *if*, *for*, *?:* operator) and for each *&&* and *||* operator in these statements. *Switch*-statements add complexity for each exit from a case (due to a *break*, *goto*, *return*, *throw*, etc.). One count is added for the default case even if it is not present. Each *catch*-statement (but not the *try* and *finally*-statement) in a *try*-block adds one count to the complexity as well.

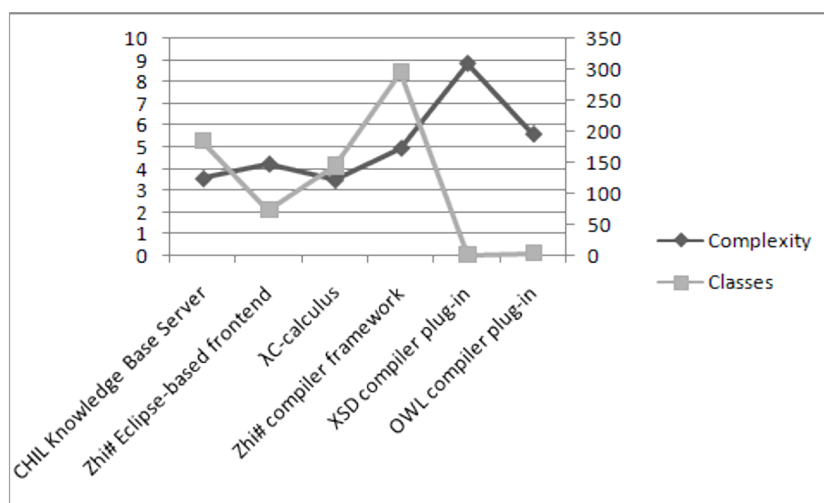


Figure 7.3: Code base complexity

The small number of plug-in classes and their higher complexity indicates that the developed compiler framework fosters the implementation of strongly cohesive plug-ins

whose functionality is then automatically available everywhere in the complete Zhi# programming language. Note that the higher complexity of the compiler plug-ins may as well be attributed to the 19 different primitive XML data types and the ubiquitous distinction between OWL object and datatype properties, which leads to a higher number of *if* and *switch-case*-statements in the XSD and OWL plug-in program codes.

The usual validation technique for each compiler software to compile itself has not been possible for the Zhi# compiler since its Zhi# input language is a proper superset of C# 1.0 while the compiler is written in C# 2008, which adds – among other language features – heavily used generics, partial classes and methods, and Language Integrated Query (LINQ). Instead, the author devised 60 categorized test cases amounting to 12,212 lines of Zhi# test code that was inductively constructed based on the Zhi# language grammar. External XSD and OWL type definitions were used wherever applicable. Additionally, 8,997 lines of typing information are used to regression test the results of the semantic analysis phase of the Zhi# compiler.

Using the current implementation of the Zhi# compiler framework (and affiliated components such as the XSD and OWL plug-ins) the compilation of the 12,212 lines of Zhi# test code to plain C# takes about 40 minutes on a Pentium M computer with 1.8 GHz and 2 GB of RAM. Fig. 7.4 depicts how the total time spreads over different compilation phases. Note that each phase is repeatedly executed for each of the 60 test cases. The “Type systems setup phase” includes the initialization of the type table, loading referenced .NET assemblies (e.g., mscorlib.dll), initializing external type systems, and loading external type definitions (e.g., .xsd and .owl-files). What makes the type table and type systems setups computationally expensive is the fact that Zhi# type names are represented by an LL(3) language<sup>1</sup>. Hence, even simple type table lookups may require several passes of the type names parser. Admittedly, in the current implementation only some optimizations were applied to the type table implementation (e.g., applicable

---

<sup>1</sup>Recall that Zhi# type names may comprise constraint notations, which may be made up of regular expressions. Also remember that in Zhi# external type names can be ubiquitous in .NET type definitions.

operators of a type are lazily added to its type table entry upon operator identification). Most computation time is spent on the semantic analysis phase, which is based on the results of external subsumption mechanisms and deduction engines such as the  $\lambda_C$ -calculus for XML data types and an ontological reasoning engine for OWL concept descriptions. Accordingly, the performance of the semantic analysis phase is highly contingent on the used external type systems (i.e. compiler plug-ins).

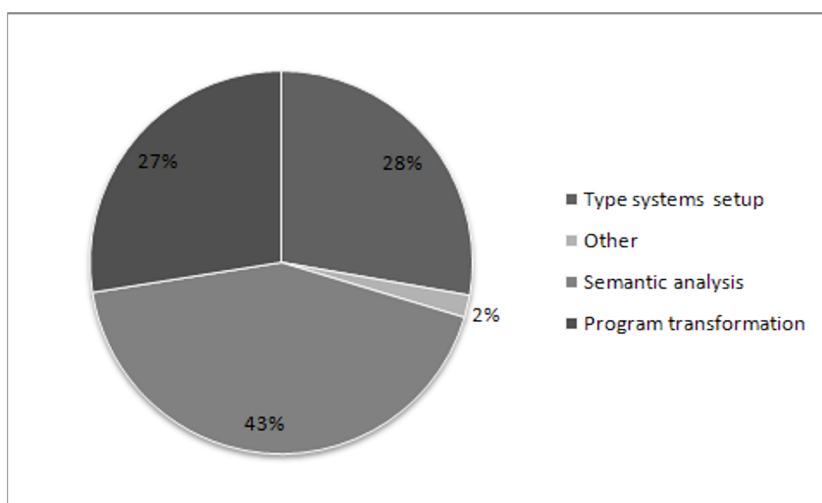


Figure 7.4: Compilation time

The same goes for the program transformation phase since program transformations (i.e. implementations of the *IExternalCompiler* framework extension point) may as well use typing information. In fact, neither the program transformation of references of XSD nor of OWL type definitions is purely syntactical. Necessary distinctions based on typing information is reflected in a comparatively higher code complexity of the XSD and OWL compiler plug-ins compared to the compiler framework.

Performance of the *Zhi#* compiler framework may certainly be improved by a more optimized type table implementation. A (mechanical) rewriting of the *Zhi#* type names grammar to LL(1) may yield only minor performance gains because each type name's parse requires a full-fledged AST construction instead of a simple application of a (pre-compiled) regular expression as it would suffice for conventional *C#* type names.

## 7.2 Microscopic Evaluation

The purpose of the Zhi# compiler framework is to facilitate the use of external type definitions in otherwise conventional C# programs. Thesis 2 (see Section 1.1) claimed that the proposed framework-based type system integration is preferable to purely C#-based approaches to accommodate XSD and OWL processing by means of proxy classes and libraries. Hence, the evaluation of the presented approach will start with microscopic evaluations on the source code level.

The difficulty of the OO paradigm to accommodate XML processing by means of plain object-oriented types has been widely acknowledged. The next subsection alludes to manual mapping options from restricted XSD types to C# type definitions that were proposed by Lämmel and Meijer [LM07]. The `xsd.exe` tool of the .NET platform will be exercised to automatically generate C# classes from an XSD schema file. The usage of both manually as well as automatically devised C# proxy classes will be compared to Zhi#'s native XML data types.

The following subsection scrutinizes to what extent the *System.XML* set of .NET APIs supports restricted value spaces of atomic XML data types. This discussion will for the most part be universally valid for all available off-the-shelf XML APIs since *System.XML* supports the widely implemented W3C DOM standards [The98], which define platform and language-neutral interfaces to access and update the content of XML documents.

Next, the C#-based realization of OWL concept and role constructors will be compared with Zhi#'s programming language inherent support for ontological concept and role descriptions, which particularly facilitates equality and runtime type checks, the declaration of ad hoc relationships, and method prototyping based on ontological concepts.

Eventually, client code for the Protégé knowledge base framework and the Jena Semantic Web Framework will be compared with equivalent Zhi# programs to accomplish a set of seven common knowledge base programming tasks such as making an ontology available in a computer program and checking individual inclusion.



### 7.2.1 XML data types

The difficulties of devising XML-to-OO mappings, which take into account the plethora of XML and XSD idiosyncracies while at the same time providing a general and convenient programming model, where mapped object types reasonably compare to native object types, have been widely acknowledged and extensively studied [MS03, Tho03, LM07, CCD08]. Lämmel and Meijer [LM07] propose four mapping options for atomic XML data types when type derivation by restriction is involved. Ponder the definition of the restricted type *age*, which may be used to declare elements in complex XML type definitions, in the upper part of the following schema.

```
1 <xsd:schema xmlns:xsd="..." >
2   <xsd:simpleType name="age">
3     <xsd:restriction base="xsd:unsignedInt">
4       <xsd:maxExclusive value="110"/>
5     </xsd:restriction >
6   </xsd:simpleType>
7   <xsd:element name="person">
8     <xsd:complexType>
9       <xsd:sequence>
10        <xsd:element name="hasAge" type="age"/>
11      </xsd:sequence >
12    </xsd:complexType>
13  </xsd:element >
14 </xsd:schema>
```

The first mapping option assumes that XML complex types are directly represented by OO class types and maps all simple XML types (including the derived ones) to primitive types of the used OO programming language. Accordingly, the complex XSD content model *person* in the lower part of the schema above would be mapped to the C# class *Person* as shown below.

```

1 class Person {
2     public uint HasAge;
3 }

```

Clearly, there is no guarantee at all that *Person* objects will have valid age properties. As a first enhancement, the second mapping option transports restrictions as dynamic type checks into the OO class definition as follows. While data type restrictions are now enforced at runtime by schema derived setter implementations there is obviously still no schema-aware static type checking.

```

1 using System.Diagnostics;
2 internal class Person {
3     private uint _HasAge;
4     public uint HasAge {
5         get { return _HasAge; }
6         set { Trace.Assert(value < 110);
7             _HasAge = value; }
8     }
9 }

```

A second enhancement as shown below is proposed in order to enable static type checking to guarantee that values of restricted types do meet the defined restrictions. Each derived XSD type would be represented by a .NET *struct* type, which provides implicit conversion operators to and from the best-matching .NET intrinsic data type. Hence, repeated dynamic checks of the value of the restricted type can be omitted. Still, in contrast to XML data types in Zhi#, assignments to the .NET *struct* objects can only be checked at runtime. The use of value types minimizes storage overhead for wrapping the actual data values. However, because there is no subtyping for *struct* types in C#, there would be no proper substitution for derived simple types (e.g., XSD's  *xsi:type* feature can be used to change the type elements in XML instance documents, where the new type must be validly derived from the overridden type).

```

1 using System.Diagnostics;
2 internal struct uint110 {
3     private uint value;
4     public static implicit operator uint110(uint value) {
5         Trace.Assert(value < 100);
6         return new uint110 {value = value};
7     }
8     public static implicit operator uint(uint110 value) {
9         return value.value;
10    }
11 }

```

Eventually, Lämmel and Meijer suggest the use of wrapper classes as indicated below (wrapper classes for other simple types are designated likewise). In a similar way to Zhi#'s XSD plug-in there are predefined classes for XSD's built-in simple types. However, the approach shown below is quite costly because of required *boxing* and *unboxing*<sup>2</sup>.

```

1 abstract class XsdAnySimpleType {
2     protected object value;
3 }
4 class XsdUnsignedInt : XsdAnySimpleType {
5     public static implicit operator XsdUnsignedInt(uint value) {
6         return new XsdUnsignedInt {value = value};
7     }
8     public static implicit operator uint (XsdUnsignedInt value) {
9         return (uint) value.value;
10    }
11 }

```

---

<sup>2</sup>Boxing is the process of converting a value type to the type *object* or to any interface type implemented by the given value type. When the CLR boxes a value type, it wraps the value inside an *object* and stores it on the managed heap. Unboxing extracts the value type from the *object*, which will eventually be garbage collected.

There are no such value type/reference type conversions (and no related performance issues in both speed of execution and code size) in the implementation of the  $\lambda_C$ -type system for XSD, which is used to process XML values in Zhi# (see Subsection 5.1.4).

Lämmel and Meijer note that “OO programmers may prefer object models that leverage the familiar primitive types of their OO language”. The latter mapping option is found to impair programming convenience even if implicit casts are defined such that wrapper classes can be used interchangeably with associated primitive types. Still, “the resulting hybrid may be difficult to comprehend by the programmer”. Lämmel and Meijer conclude that “there is no fully satisfactory simple-type mapping”.

In contrast, the framework-based Zhi# approach provides for the realization of object models that contain restricted XML data types whose usage is statically checked and conforms to the familiar usage of .NET primitive types. For the considered XML data types *age* and *person* a Zhi#-based object model can be straightforwardly devised as follows.

```
class Person { public #age HasAge; }
```

Given the substantial research efforts that have been devoted to work out practical XML-to-OO mapping options one may assume that available off-the-shelf code generation tools will regard recent insights. The XML Schema Definition (`xsd.exe`) tool, which ships with the .NET framework, can be used to automatically generate Common Language Runtime classes from XML schema definitions. Unfortunately, `xsd.exe` implements the sloppiest mapping option discussed here where derived XML data types are simply represented by plain .NET intrinsic data types. If the XML complex type *person* was augmented to include a child element *birthday* of type *gMonthDay* the `xsd.exe` tool would generate the following C# class definition (generated comments and class attributes are omitted for brevity). The XML child element *hasAge* of type *age* is simply represented by a *uint* type property of the .NET wrapper class (the `xsd.exe` tool can also be configured to generate publicly visible fields instead). User defined restrictions of derived XML data

types are discarded, which results in a lack of static typing and dynamic checking and a lack of support for round-tripping since the .NET *hasAge* property does not report its value to be of the restricted XML data type *age*.

```
1 public partial class person {
2     private uint hasAgeField;
3     private string birthdayField;
4     public uint hasAge {
5         get { return this.hasAgeField; }
6         set { this.hasAgeField = value; }
7     }
8     [System.Xml.Serialization.XmlElementAttribute(DataType="gMonthDay")]
9     public string birthday {
10        get { return this.birthdayField; }
11        set { this.birthdayField = value; }
12    }
13 }
```

Characteristics of XML elements can be declared by using the .NET *XmlElement* attribute. In line 8, the *birthday* type property is designated the XML data type *gMonthDay*. However, these annotations are only made for built-in XML data types (note there is no such metadata for the *hasAge* field, which would have to be annotated with the user defined type name *age*). Also, only a plain (i.e. unchecked) .NET *string* property is designated to represent the *gMonthDay birthday* element. Hence, dynamic checking of serialized XML documents is facilitated but made completely optional and left to the programmer. In contrast, both static and dynamic schema-aware checking is inherently provided for Zhi#-based object models. Also, there is no need for attribute-based programming to annotate .NET type properties with XSD type information. In Zhi#, XML data type values provide an auxiliary *Type* property to query the represented XML data type, which may be used to control the serialization of object models.

### 7.2.2 XML APIs vs. XML data types

Any Unicode character stream that transports syntactically valid text data according to the W3C syntax rules of XML can be considered an XML *instance document*. While this is the most general definition of valid XML data, it is in no way conceivable to compel programmers to process XML instance documents on the character stream level. Instead, programmatic access and manipulation of XML data should be facilitated by an application programming interface that abstracts from the concrete syntax of XML documents. In fact, the need for a higher level representation of XML data arose from technical differences between the ways early versions of the Internet Explorer (IE) [Mic95] and Netscape Navigator [Net94] Web browsers supported Dynamic HTML (DHTML). IE and Navigator named particular parts of DHTML documents differently, which impeded cross-browser scripting. The need to facilitate cross-browser scripting triggered the development of the *Document Object Model* (DOM) [The98]. The W3C DOM is a language neutral object-oriented data model to represent XML documents. Implementations of the W3C DOM exist for all widely used programming languages. The .NET framework base class library contains the *System.XML* set of APIs to support several W3C standards for processing XML – among others the W3C-recommended schema language XML Schema Definition. The *XmlSchemaSimpleType* class provides an in-memory representation of restricted XML data types. The following C# program demonstrates how the user-defined *age* data type from the previous subsection could be made available in C# programs by using *XmlSchemaSimpleType* objects.

A *TypePool* hides an *XmlSchema* object, which can be initialized with type definitions from XML schema files using the *LoadSchema()* method (invalid schema files are handled by the *VCHandler* validation callback handler). *XmlSchemaSimpleType* wrapper objects for particular XML data types, which are identified by their fully qualified type name, are returned by the *GetSimpleType()* method. The static *XmlSchemaSimpleTypeExtension* class defines a *Validate()* extension method for *XmlSchemaSimpleType* objects.

```

1 using System;
2 using System.Xml;
3 using System.Xml.Schema;
4 namespace XMLDemo {
5     public class TypePool {
6         private XmlSchema schema;
7         private static void VCHandler(object sender, ValidationEventArgs e) {
8             // Handle schema errors...
9         }
10        public void LoadSchema(string FilePath) {
11            var xtr = new XmlTextReader(FilePath);
12            schema = XmlSchema.Read(xtr, ValidationCallbackHandler);
13            var schemaSet = new XmlSchemaSet();
14            schemaSet.ValidationEventHandler += VCHandler;
15            schemaSet.Add(schema);
16            schemaSet.Compile();
17        }
18        public XmlSchemaSimpleType GetSimpleType(string N, string NS) {
19            foreach (XmlSchemaObject v in schema.Items) {
20                if (v is XmlSchemaSimpleType &&
21                    ((XmlSchemaSimpleType) v).Qualified.Name == N &&
22                    ((XmlSchemaSimpleType) v).Qualified.Namespace == NS) {
23                    return (XmlSchemaSimpleType) v;
24                }
25            }
26            throw new ApplicationException("Type not available");
27        }
28        public static class XmlSchemaSimpleTypeExtension {
29            public static void Validate(this XmlSchemaSimpleType T, object V) {
30                if (V != null) {
31                    T.Datatype.ParseValue(V.ToString(), null, null);
32                } else {
33                    throw new ArgumentNullException("Data type value must not be 'null'");
34                }
35            }
36        }
37    }
38 }

```

Note that *Validate()* takes as input arbitrary *objects*. The string representation of non-null input arguments is attempted to be parsed as a valid value for the XML data type that is represented by the *XmlSchemaSimpleType* host object. On the client side, the given code may be used as follows.

```
1 var pool = new TypePool ();
2 pool.LoadSchema(@"Person.xsd");
3 var age = pool.GetSimpleType("age", "http://www.zhimantic.com/eval");
4 age.Validate(200);
```

In line 2, a *TypePool* object is initialized with the type definitions of a given schema file. Hence, the *XmlSchemaSimpleType* object *age* can be set up as an in-memory representation of the XML data type *age* defined in the given namespace. The *Validate()* invocation in line 4 will throw an *XmlSchemaException* at runtime. Again, there is no means to implement static type checks for XML data types using the *System.XML* APIs. The same goes for LINQ to XML [Mic07a], which depends on the same schema validation classes that are defined in *System.Xml.Schema*.

Zhi#'s static type checking of XML data types can be assumed to be particularly beneficial when used in conjunction with XML literals, which are – up to now – only supported in the VB.NET programming language [Mic02]. If such XML declarations allowed for namespace-related “xmlns” attributes, occurrences of Zhi#'s XML data types within XML literals could be statically checked as indicated in the following pseudo VB.NET code (i.e. VB.NET + “xmlns” attributes + Zhi#-like XML data types).

```
1 Sub Main()
2   Dim age as #age;
3   Dim d = <?xml version="1.0" xmlns="http://www.zhimantic.com/eval"?>
4       <Person>
5           <hasAge>age</hasAge>      ' Validity of age data type in this
6       </Person>                    ' position could be statically checked!
7 End Sub
```



### 7.2.3 OWL concept constructors

The introductory Section 1.3 described the applicability of the Web Ontology Language to represent knowledge. In particular, the rich set of OWL DL concept constructors allows for intuitive modeling of complex concept descriptions. OWL DL’s TBox and ABox reasoning can be used to automatically classify a class hierarchy and to infer the types (i.e. the class memberships) of ontological individuals. Section 6.5 indicated the superiority of the Zhi# approach over related work to improve the programmability of OWL DL knowledge bases. In this section, the integration of OWL DL concept and role descriptions with the C# programming language will be compared to a naïve but straightforward mocking up of OWL concept and role descriptions by means of the modeling features of conventional statically typed programming languages such as ECMA standard C# 3.0 and Java 6.

In C# and Java, entities in the described world can be directly represented by programming language class types. While both C# and Java classes are limited to single inheritance, interfaces can be used to allow for multiple inheritance. The OWL DL concept description  $C \equiv A \sqcap B$  can, in a first step, be represented by C# interfaces as follows (later it will be shown that both definitions behave differently under certain circumstances).

```
1 interface IThing {}
2 interface IA : IThing {}
3 interface IB : IThing {}
4 interface IC : IA, IB {}
```

Mapping real world entities to programming language types is tempting but falls short of support for ontological ABox reasoning where the types of instances can change at runtime. Dynamic type changes can be implemented by separating programming language types from conceptual types. The following C# class definition *Individual* can be used to represent ontological individuals of different RDF types, which are listed in the *Types* property. The default constructor of the derived convenience class *A* initializes the *Types* set with the type name “A”. In program text, only references of class *Individual* would

be used, though. Type changes at runtime due to ontological reasoning can now be implemented by modifying the *Types* property.

```
1 public class Individual {
2     public HashSet<string> Types = new HashSet<string> {"Thing"};
3     public Individual(IEnumerable<string> Types) {
4         this.Types.UnionWith(Types);
5     }
6 }
7 public class A : Individual {
8     public A() : base(new [] {"A"}) {}
9 }
```

Each of the two approaches has its advantages and drawbacks in practice. In the following discussion of OWL DL's modeling features it will eventually become apparent that both approaches are inapt to implement even basic OWL concept and role constructors in their entirety in C# or Java. Zhi#'s superior programmability of ontological knowledge bases will be described for OWL DL's concept and role constructors along the following dimensions, where both the ease of modeling as well as programmability are considered.

- Ease of modeling “ontological” .NET objects (i.e. how could OWL concept constructors and role restrictions be mimicked with plain C#?).
- Ease of equality and runtime type checks (i.e. implementation and use of the `==` and *is*-operator in respect of ontological ABox reasoning and individual inclusion).
- Ease of declaration of (ad hoc) relationships between ontological individuals.

The following concept and role constructors are partly introduced by very inexpressive Description Logics such as  $\mathcal{AL}$ . Consequently, mimicking even basic Description Logic modeling features in C# and Java may be non-trivial. For the following concept and role constructors, symbols of the Description Logic naming scheme such as  $\mathcal{AL}$  and  $\mathcal{C}$  indicate the least expressive Description Logic that comprises the constructor under discussion.

**The top level concept  $\top$  ( $\mathcal{AL}$ ).** The Web Ontology Language provides the top level concept constructor  $\top$  (*Thing*). The universal concept  $\top$  subsumes all concept descriptions in an ontology (i.e. every class has  $\top$  as a superclass). As such, it is similar to the top level C# and Java programming language classes *System.Object* and *java.lang.Object*, respectively (in Java, the built-in primitive data types such as *int* and *boolean* are not inherited from *java.lang.Object*). In OWL knowledge bases, the interpretation of the top level concept  $\top$  includes *all* ontological individuals that are declared in that knowledge base (i.e.  $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$ ). Similarly, all instances of C# and Java classes are of type *System.Object* and *java.lang.Object*, respectively. It is also very easy both in C# as well as in Java to query the most specific (i.e. most-derived) type of a particular object via the ubiquitous *GetType()* and *getClass()* methods, respectively. It is, however, not possible in Java to query *all* types of a particular object. The C# *Type* class provides the *GetInterfaces()* method, which yields all implemented or inherited interfaces by the current *Type* of an object; it remains silent about the class type of the object and its inherited base classes. Both C# and Java lack support for the most common query on OWL knowledge bases, which is to answer all individuals that are in the extension of a given concept description. In C# and Java, this would, for example, require to override the constructors of the classes under consideration or to augment the class loader or garbage collector with logging functionality. Overriding the (default) constructors of C# and Java classes and holding references to the instances of interest may hamper the garbage collection of these objects. In C#, *weak references* may be used by the bookkeeping object. Still, the garbage collection would be affected. Modifying the class loader or garbage collector with logging functionality may be transparent to programs but would require a modified runtime environment being deployed with the actual program.

In OWL, a role hierarchy can be devised independently from a concept hierarchy. Also, it is possible to relate two individuals that are not in the extensions of a property's domain and range restriction. Given a role description  $R$ , where  $\exists 1R \sqsubseteq A$  and  $\top \sqsubseteq \forall R.B$  (i.e.  $R$  relates elements from the domain  $A$  with elements from the range  $B$ ), the ontological

individuals  $Thing(x)$  and  $Thing(y)$  can be related in Zhi# by  $R$  as follows. As a result, individuals  $X$  and  $Y$  are inferred to be an  $A$  and a  $B$ , respectively.

```

1 #Thing x = [...];
2 #Thing y = [...];
3 x.R = y;           // x and y are inferred to be an A and a B, respectively

```

In C# and Java there is no support for such kinds of ad hoc relationships. Instead, programmers are forced to make premature commitments about inter-entity relationships [Rum87]. Two C# *objects* cannot be related by an a priori unanticipated role  $R$  as shown below. Support for adding and overriding members on a per instance basis in object-oriented programming languages is – independently from the Web Ontology Language – demanded by Erik Meijer and Peter Drayton [MD04].

```

1 Object x = [...];
2 Object y = [...];
3 x.R = y;           // Rejected by C# compiler

```

In order to allow for ad hoc relationships in statically typed programming languages, one could use hash tables that hold property values as key/value pairs.

```

1 class Individual {
2     [...]
3     Dictionary<string/*Property name*/,           ↔
4         HashSet<Individual>/*Property values*/> PropertyValues;
5 }

```

This approach does, however, not allow for normal member access to declare property values. More severely, the link between two objects would be navigable only into one direction while in OWL (and in Zhi#) it is possible to use inverse roles to navigate in both directions (using normal member access). Being restricted to C# and Java code, one may eventually end up implementing a full-fledged relational model as shown below in order to provide for navigable relationships for declared subject-property-object triples.

For the following discussions, the *Individual* class will be used for simplicity while it will only be indicated how a relational model could help out whenever navigable links are necessary. Eventually, a full-fledged relational model to represent ontological information will most likely be least convenient in programming.

```
table A{string AID} table B{string BID} table R{string AID; string BID}
```

**Universal negation**  $\neg C$  ( $\mathcal{C}$ ). Except for the basic Description Logic  $\mathcal{AL}$  (= attributive language), where negation can only be applied to atomic concepts, all Description Logics provide a concept constructor for universal negation ( $\neg C$ ). This *complement of* constructor selects all individuals from the domain of discourse that do not belong to a given class. Implementing this constructor in C# and Java is even more difficult than providing support for the top level concept. Not only knowledge about the instances of one particular class but about *all* instances of *all* classes is required to query the extension of a complement concept description. On the other hand, a runtime type check of one particular object can be easily performed in C# and Java, too, as follows for a concept description  $\neg C$ . Shown below is a runtime type check of an ontological individual represented by an *Individual* object. Note that in order for the purely syntactical latter type check to work the names of *all* types of the individual must be contained in the *Types* collection of the *Individual* object and not only the most-derived ones.

```
1 IThing x = [...];
2 if (!(x is IC)) { [...] }
```

```
1 Individual x = [...];
2 if (!x.Types.Contains("C")) { [...] }
```

In Zhi#, complement classes can be used natively to declare formal method parameters. For an OWL concept description  $NotC \equiv \neg C$  the following Zhi# method accepts individuals that are *not* in the extension of concept  $C$ .

```
void f(#NotC p) { [...] }
```

In C#, the *complement of* constructor is unknown. For given interfaces  $IA$ ,  $IB$ , and  $IC$  one could introduce an interface  $INotC$  that subsumes  $IA$  and  $IB$ . Accordingly, the resulting C# code will look similar to the Zhi# code for the ontological concept  $NotC$ .

```
void f(INotC p) { [...] }
```

The C# solution will, however, not work in the presence of an interface  $ID$  that extends both  $IA$  or  $IB$  and  $IC$ . The method argument might then be both an  $INotC$  and an  $IC$  object and will erroneously be accepted. If later (i.e. after the definition of interface  $INotC$ ) an additional interface  $IE$  is introduced, instances of  $IE$  will erroneously not be accepted since  $IE$  is not subsumed by interface  $INotC$ .

If *Individual* objects were used to represent ontological individuals the now dynamic type check of argument  $p$  would have to be devised manually within the method body as follows.

```
1 void f(Individual p) {
2     if (!p.Types.Contains("C")) {
3         [...] // Runtime type error
4     } else {
5         [...] // Method implementation
6     }
7 }
```

**Conjunction**  $A \sqcap B$  ( $\mathcal{AL}$ ). The *conjunction* constructor  $C \equiv A \sqcap B$  states that the concept description  $C$  is *exactly* the intersection of the concepts  $A$  and  $B$ . This means that if something is an  $A$  and a  $B$ , then it is an instance of  $C$ . This definition seemingly corresponds with the derivation of the C# interface  $IC$  from interfaces  $IA$  and  $IB$ . On closer consideration one will, however, notice that a runtime type check based on C# programming language types will yield different results. Assume an interface  $ID$ , which extends both  $IA$  and  $IB$ . An instance of  $ID$  is both an  $IA$  and an  $IB$ . Still, the nominal C# type system will not draw the conclusion that an object that is an  $IA$  and an  $IB$

automatically is an  $IC$ . Accordingly, it is not possible to facilitate a runtime type check in C# by simply taking the type  $IC$  as the intersection of  $IA$  and  $IB$ . Instead, one has to devise an explicit dynamic check for each conjoined concept description as shown below. This type check must always be provided extra to a given object since instances of  $ID$  do not know by themselves that they represent entities of class  $C \equiv A \sqcap B$ .

```

1  if ((x is IA) && (x is IB)) {
2      [...] // x is a C according to the concept description C ≡ A ∩ B
3  }

```

In the class *Individual* the *Types* property could be augmented as follows to take into account the ontological reasoning that every object that is an  $A$  and a  $B$  is also a  $C$ .

```

1  public class Individual {
2      private readonly HashSet<string> _Types = new HashSet<string> {"Thing"};
3      public HashSet<string> Types {
4          get {
5              var types = new HashSet<string>(_Types);
6              if (_Types.Contains("A") && _Types.Contains("B")) {
7                  types.UnionWith(new[] {"C"});
8              }
9              return types;
10     } }
11     public Individual(IEnumerable<string> Types) {
12         _Types.UnionWith(Types);
13     }
14 }

```

In the Zhi# programming language, the OWL concept description  $C \equiv A \sqcap B$  can be used natively along with the *is*-operator and as the type of a formal method parameter as shown below. Static type checks based on the TBox of the current ontology are automatically facilitated by the Zhi# compiler. At runtime, dynamic checking based on the TBox and the ABox of the current ontology is accomplished by the Zhi# runtime library and the used OWL reasoner.

```

1 #Thing x = [...];
2 if (x is #C) { [...] }

void f(#C p) { [...] }

```

Note that the described usages of the complex concept description  $C \equiv A \sqcap B$  in Zhi# programs are aware of modifications of the TBox of the ontology after the compilation of the Zhi# program. Runtime type checks will still function properly and method  $f()$  in the above code snippet will be executed exactly for those ontological individuals that are in the intersection of the extensions of concept descriptions  $A$  and  $B$  (no matter if an individual was explicitly declared as a  $C$  or as a  $D$ , where  $D \equiv A \sqcap B$ ).

**Disjunction**  $A \sqcup B$  ( $\mathcal{U}$ ). The *union* constructor is available in all Description Logics that contain negation and conjunction. A union of ontological concepts corresponds to extending a common base interface in  $C\#$ . Thus, a concept description  $C \equiv A \sqcup B$  can be represented by  $C\#$  interfaces  $IA$  and  $IB$  extending interface  $IC$ . This representation is appropriate for declaring method signatures with a formal parameter of type  $IC$  where both  $IAs$  and  $IBs$  are valid input objects. Also, a runtime type check using  $C\#$ 's *is*-operator will yield that  $IA$  and  $IB$  objects are compatible with interface  $IC$ . Still,  $C\#$ 's modeling features are not sufficient to relate two  $Cs$  by a role  $R$  with domain  $A$  and range  $B$  because interface  $IC$  will not contain a slot  $R$ . Also, reasoning based on property domain and range restrictions is not supported in  $C\#$ .

```

1 IC x = [...];
2 IC y = [...];
3 x.R = y;           // Error in C# because there is no slot R in IC

```

If individuals of RDF types  $A$ ,  $B$ , and  $C$  were represented by *Individual* objects, the *Types* property would have to be modified such that  $C$  is added to the set of RDF types if an individual is an  $A$  or a  $B$  (cf. the modified *Types* property for the conjunction concept constructor).



```

1  [...]
2  public HashSet<string> Types {
3      get {
4          var types = new HashSet<string>(_Types);
5          if (_Types.Contains("A") || _Types.Contains("B")) {
6              types.UnionWith(new[] {"C"});
7          }
8          return types;
9      }
10 }
11 [...]

```

**Enumeration**  $\{\mathbf{o}_1, \dots, \mathbf{o}_n\}$  ( $\mathcal{O}$ ). The OWL DL *one of* constructor can be used to define a class by explicitly enumerating the individuals that make up the class. The members of the class are exactly the set of enumerated individuals; no more, no less. Both in C# and Java the *enum* keyword can be used to declare enumerations, too. Unfortunately, C# and Java enumerations simply consist of a set of named constants and their integral values. There is no built-in standard way to trace back from a value to the name of the constant without already knowing the enumeration type. Also, the same integral values may be used for different constant names in different enumerations. In contrast, runtime type checks in Zhi# programs work properly for ontological individuals that are members of enumerated classes. Let variable  $o$  refer to the ontological individual  $\mathbf{o}$  that was used to make up the definition of the enumerated concept  $E \equiv \{\mathbf{o}, \dots\}$ .

```

1 #Thing o = [...];
2 if (o is #E) { [...] } // Individual o is compatible with concept description E

```

In the same way, individual  $\mathbf{O}$  is a valid input object for the method  $f()$  declared below (*only* the enumerated members of concept description  $E$  are valid input objects).

```

void f(#E p) { [...] }

```

By using the auxiliary property *Individuals* that is defined for static concept references in Zhi# programs it is possible to yield the enumerated individuals of a concept description. For an enumerated concept  $WineColor \equiv \{Red, Rose, White\}$  the output of the program shown below is

Red

Rose

White.

```
1 foreach (#Thing x in #E.Individuals) {
2     Console.WriteLine(x);
3 }
```

**Exists restriction  $\exists R.C$  ( $\mathcal{E}$ ).** The OWL DL *someValuesFrom* restriction is stated on a property with respect to a class. For a given class a property can have a restriction that at least one value for that property is of a certain type. Individuals where *some* (i.e. at least one) values of the given property are of that certain type will be inferred to be in the extension of the given class. Recall that because of OWL's open world assumption an OWL reasoner will not infer from the absence of such a property value that an individual is not of a certain type. Also, a reasoner cannot deduce that *all* related property values are of a certain type.

Accordingly, the concept description  $C \equiv \exists R.D$  could be implemented by an *Individual* object in C# as shown below. Publicly visible class members to modify the *PropertyValues* property are omitted for brevity (recall from the discussion of the top level concept ( $\top$ ) that the implementation of bidirectionally navigable subject-property-object triples is itself a burden in a conventional object-oriented programming language).

The given C# code is only to demonstrate the semantics of the *someValuesFrom* restriction and is by no means sufficient to cope with, for example, mutually dependent class definitions. For concept descriptions  $C \equiv \exists R.D$  and  $D \equiv \exists R.E$  the inference results would otherwise vary depending on the evaluation order of the *Types* property.

```

1 public class Individual {
2     private readonly HashSet<string> _Types = new HashSet<string> {"Thing"};
3     private Dictionary<string, HashSet<Individual>> PropertyValues = ↔
4         new Dictionary<string, HashSet<Individual>>();
5     public HashSet<string> Types {
6         get {
7             var types = new HashSet<string>(_Types);
8             if (PropertyValues.ContainsKey("R")) {
9                 foreach (var v in PropertyValues["R"]) {
10                    if (v.Types.Contains("D")) {
11                        types.UnionWith(new[] {"C"});
12                    }
13                }
14            }
15            return types;
16        }
17    }
18 }

```

Also, the *someValuesFrom* restriction can as well be used along with OWL datatype properties. For example, a class *Teenager* may be defined as  $\exists hasAge.teenAge$ , where *hasAge* shall be a functional OWL datatype property and the XML data type *teenAge* be defined as an integer between 13 and 19. This would require implementing full-fledged type inference for data types, too.

In contrast, the results of ontological reasoning and XML data type subsumption are immediately available for ontological individuals and data type values in Zhi#. Ponder the following Zhi# code, which uses the considered class and data type definitions. Note that the order of the assignment statements in line 2 and 3 is irrelevant for the ontological reasoning that is triggered in line 4. Also, as shown in line 8 and 9, the availability of XML data types in the Zhi# programming language makes it possible to obey the *someValuesFrom* restriction for OWL datatype properties, too.

```

1 #Thing c = [...], d = [...];
2 c.#R = d;
3 d.#R = new #E([...]);
4 if (c is #C) {
5     // Individual c is a C
6 }
7 #Thing p = [...];
8 p.#hasAge = 15;
9 if (p is #Teenager) {
10     // Teenage rampage will happen here
11 }

```

**Value restriction  $\forall R.C$  ( $\mathcal{AL}$ ).** The OWL DL *allValuesFrom* restriction is stated on a property with respect to a class. It associates a local range restriction with the given property for the given class. If an instance of the class is related by the property to a second individual, then the second individual will be inferred to be an instance of the local range restriction class. Note that due to the monotonic nature of the Web Ontology Language, an OWL reasoner cannot infer a subject to be of a particular type only because all related objects are in the extension of the local range restriction. Moreover, an individual may as well have no property values at all.

Implementing the *allValuesFrom* restriction in conventional object-oriented programming languages would be particularly tedious and problematic because concept descriptions of the form  $C \equiv \forall R.D$  base the type inference of objects on their use as property values (of other objects)! This directly renders field-based representations of object properties insufficient. Bidirectionally navigable links between objects are required to consider for type inference all objects that have the object under consideration as a property value. Such an implementation would most likely lead to devising a full-fledged relational model (cf. the discussion of the top level concept). It can be understood that such a model is

indispensable for all Description Logic applications since the *allValuesFrom* restriction is already introduced in the very basic description language  $\mathcal{AL}$ , which has been introduced by Schmidt-Schauss and Smolka [SS91] as a minimal language that is of practical use.

In the Zhi# programming language, type inference of individuals that are used as objects of locally range restricted properties is delegated to the OWL DL reasoner. As a consequence, there is no need to implement an auxiliary relational model on the application level. In the following Zhi# code snippet, individual  $D$  will automatically be inferred to be a  $D$  based on the concept description  $C \equiv \forall R.D$ .

```

1 #C c = [...];
2 #Thing d = [...];
3 c.#R = d; // This makes D a D

```

**Nominals**  $\exists R.o$  ( $\mathcal{O}$ ). Sometimes, it is useful to allow *individual names* (also called *nominals*) not only in the ABox but also in the description language. The Web Ontology Language provides the *hasValue* local range restriction to express the notion that for a given class a property is required to have a certain individual as a value. That is, classes are specified based on the existence of particular property values. An individual is a member of such a class whenever at least *one* of its property values is equal to the *hasValue* resource. This leads directly to the issue of object equality in object-oriented languages (cf. the equality discussion in Subsection 6.1.2).

Ponder the OWL concept description  $C \equiv R.o$ . An ontological individual  $C$  is a  $C$  if it is related to individual  $o$  by property  $R$  (i.e.  $\langle C, o \rangle \in R^{\mathcal{I}}$ ). Note that in OWL, which does not make the unique name assumption, the equality of individuals is based on the represented entities in the described world. Individuals are identical if they refer to the same entity in the described world. The OWL feature *sameAs* can be used to state that two individuals do refer to the same entity (similarly, the *differentFrom* and *AllDifferent* features can be used to declare individuals mutually different).

Object-oriented programming languages do neither support OWL's entity-based equality definition, nor is there a *sameAs* predicate that could be used to declare identity. The ubiquitous *CompareTo<T>()* method of the .NET BCL interface *IComparable<T>* can in fact be implemented arbitrarily (there is a common understanding, though, of the meaning of its return value). The BCL interface *ICloneable* stipulates the implementation of its *Clone()* method, which can most likely be related with OWL's *sameAs* feature. However, the usefulness of the *ICloneable* interface is currently under debate within the .NET community since the official specification does not explicitly say that objects implementing this interface must return a *deep copy* of the object. Thus, it is technically possible that objects implementing *ICloneable* actually return only a *shallow copy*, which clearly generates a good deal of confusion. Consequently, it is particularly tedious and error-prone to implement nominals-based concept descriptions in object-oriented programming languages. The C# code shown below is an attempt to mimic OWL nominals.

A dedicated class *HasValueRestriction* is introduced to represent a property value restriction (the provided *Type* shall be the equivalent concept name). Effective value restrictions are considered for type inference in the implementation of the *Types* property of class *Individual*. Note that the common *Contains()* method of the *ICollection<T>* interface uses reference equality for object comparison, which is not applicable here because several *Individual* objects may be used to represent the same entity. Value-based equality (based on, for example, an extra *Name* field in the *Individual* class) would be inadequate, too, since OWL does not make the unique name assumption. As a result, extra housekeeping code (e.g., a relational model) is required to identify conceptually identical *Individual* objects. This assumed housekeeping code is used in line 16 by the *ContainsIndividual()* method, which could be made available for .NET *ICollection<T>* objects as a C# extension method<sup>3</sup>.

---

<sup>3</sup>Extension methods were introduced in C# 3.0 and make it possible to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they can be called as if they were instance methods on the extended type.

```

1 public struct HasValueRestriction {
2     public string Property;
3     public Individual Individual;
4     public string Type;
5 }
6 public class Individual {
7     private readonly HashSet<string> _Types = new HashSet<string> {"Thing"};
8     private readonly HashSet<HasValueRestriction> HasValueRestrictions;
9     private Dictionary<string, HashSet<Individual>> PropertyValues = ↔
10     new Dictionary<string, HashSet<Individual>>();
11     public HashSet<string> Types {
12         get {
13             var types = new HashSet<string>(_Types);
14             foreach (var r in HasValueRestrictions) {
15                 if (PropertyValues.ContainsKey(r.Property)) {
16                     if (PropertyValues[r.Property].ContainsIndividual(r.Individual)) {
17                         types.UnionWith(new[] {r.Type});
18                     }
19                 }
20             }
21             return types;
22         }
23     }
24 }

```

The described difficulties do not apply to ontological individuals in Zhi# programs. First, no user defined housekeeping code is required for object equality. For ontological individuals the equality (==) and inequality (!=) operators are overridden to consider conceptual identity based on the ontological model. Second, no user defined deduction rules are required to implement type inference. Property value restrictions are automatically used for reasoning by Zhi#'s OWL runtime library (i.e. the ontology management system). The Zhi# program shown below is backed by an ontology where persons can be consid-

ered as friends of Bill Gates (i.e.  $\text{FriendOfBillGats} \equiv \text{Person} \sqcap \exists \text{hasFriend.BILLGATES}$ ). Also, the individuals `BILLGATES` and `WILLIAMHENRYGATES` refer to the same entity Bill Gates in the real world. The following program will correctly output the information that Paul Allan is a friend of Bill Gates (see Subsection 6.1.3 for an explanation of the *Individuals* property of static concept references in Zhi# programs).

```

1 // WILLIAMHENRYGATES and BILLGATES are identical
2 #Person p = new #Person("#PAULALLAN");
3 p.#hasFriend = new #Person("#WILLIAMHENRYGATES");
4 foreach (#Person person in #FriendOfBillGates.Individuals) {
5     Console.WriteLine(person + " is a friend of Bill Gates!");
6 }

```

**(Un)qualified number restrictions**  $[\geq | = | \leq]R[C]$  ( $\mathcal{N}/\mathcal{Q}$ ). The basic description language  $\mathcal{AL}$  contains the *limited existential quantification* ( $\exists R.\top$ ) concept constructor. As one might guess there are several ways in which the expressive power of the notion of number restrictions can be enhanced.

The *SHIF*(**D**) Description Logic, which constitutes the formal foundation of OWL Lite, introduces *unqualified number restrictions* ( $\mathcal{F}$ ) limited to statements concerning cardinalities of value 0 or 1. In the *SHOIN*(**D**) Description Logic, arbitrary non-negative integer cardinalities can be used with *unqualified number restrictions* ( $\mathcal{N}$ ). Number restrictions can be qualified ( $\mathcal{Q}$ ) in *SHIQ*(**D**), which had initially been intended to be the formal foundation of OWL DL. In fact, OWL DL has been developed to implement *SHOIN*(**D**) with subtle limitations imposed on the usage of role axioms (e.g., inverse roles cannot be declared to be transitive). Still, in OWL DL, arbitrary non-negative integer cardinalities can be stated on properties with respect to a certain class using OWL's *minCardinality* and *maxCardinality* features. As a convenience, the *cardinality* feature can be used when a property on a class has identical values for its *minCardinality* and *maxCardinality* facets.



It is important to understand that number restrictions (despite their name) do *not* restrict the number of OWL object property values. Instead, number restrictions are used for ontological reasoning such that, individuals are considered identical if their number exceeds the maximum cardinality of a property on a certain class (i.e. number restrictions operate on the conceptual entities and not on the surrogate individuals). In contrast, in object-oriented programming languages instances of a class cannot be inferred to be identical. If there are fewer OWL property values than required by a *minCardinality* restriction these values are assumed to exist but are considered unknown (i.e. open world assumption). Again, there is no such assumption in object-oriented programming languages. The described reasoning for *maxCardinality* restrictions does, of course, not happen for OWL datatype property values because data type values are not subject to ontological reasoning.

In view of these observations the semantics of OWL number restrictions must not be confused with multiplicities of associations in object-oriented designs such as UML class diagrams. In UML, multiplicities define definitive bounds for the number of linked objects<sup>4</sup>. These conceptual differences between the semantics and the interpretation of number restrictions in OWL and object-oriented programming languages make it again particularly difficult to implement OWL's cardinality features in conventional C#.

Ponder a concept description  $C \equiv \leq 1R.\top$  (i.e. role  $R$  is *functional* or *unique* on class  $C$ ). All individuals that are related to a  $C$  by role  $R$  will be inferred to be identical (i.e.  $\{\langle O, I_1 \rangle, \dots, \langle O, I_n \rangle\} \in R^{\mathcal{I}} \Rightarrow I_1 = \dots = I_n$ ). The other way around it will be a consistency error if a  $C$  is related to several different individuals. Recall that due to the monotonic nature of OWL an individual  $O$  will not be inferred to be a  $C$  only because it (currently) has at most one value for property  $R$  since later on further values may be added, which would require to revoke a previous conclusion. The following C# code is an attempt to implement these behaviors.

---

<sup>4</sup>A note on multiplicities in UML class diagrams: Akehurst et al. [AHM07] observed that seven out of ten investigated UML design tools support modeling multiplicities while only one actually facilitates bounds checks.

```

1 public struct MaxCardinalityRestriction {
2     public string Property;
3     public uint MaxCardinality;
4     public string Type;
5 }
6 public class Individual {
7     readonly HashSet<string> _Types = new HashSet<string> {"Thing"};
8     readonly HashSet<MaxCardinalityRestriction> MaxCardinalityRestrictions;
9     public Dictionary<string, HashSet<Individual>> PropertyValues = ↵
10     new Dictionary<string, HashSet<Individual>>();
11     public HashSet<string> Types {
12         get {
13             // No type inference for the current individual
14             // based on maximum cardinality restrictions!
15             return _Types;
16         }
17     }
18     public void AddObjectPropertyValue(string p, Individual i) {
19         if (PropertyValues.ContainsKey(p)) {
20             PropertyValues[p].Add(i);
21         } else {
22             PropertyValues.Add(p, new HashSet<Individual> {i});
23         }
24         foreach (var r in MaxCardinalityRestrictions) {
25             if (Types.Contains(r.Type) && r.Property == p &&
26                 PropertyValues.ContainsKey(p) &&
27                 (from v in PropertyValues[p]
28                  where v.GetHashCode() == i.GetHashCode()
29                  select v).Count() > r.MaxCardinality) {
30                 throw new InconsistentOntologyException();
31             }
32         }
33     }
34 }

```

How does this implementation cope with the given scenario of adding values for property  $R$  of an ontological individual of type  $C$ , where  $C \equiv \leq 1R.\top$ ?

First, note that maximum cardinality restrictions do not contribute to the type inference for the current *Individual* object (i.e. the *Types* property is not modified in the presence of aggregated *MaxCardinalityRestriction* objects).

Second, the implementation of the *AddObjectPropertyValue()* method to modify the *PropertyValues* field includes a consistency check in order to obey each aggregated *MaxCardinalityRestriction*. The implemented check simply throws for inconsistent objects but does not roll back the operation that lead to the incongruity with the TBox (i.e. the current *Individual* object will remain in an inconsistent state). This behavior is identical to current ontology management systems, where inconsistencies of the ontology are merely reported to the user. Still, the given code is incomplete for two reasons.

First, it does not take into account the possibility of transitive and inverse properties and role hierarchies (see below).

Second, the given code assumes that the *Equals()* method for *Individual* objects was overridden or an extra *IEqualityComparer<Individual>* is provided in order to account for the required ontological equality between *Individual* objects (operations on .NET BCL *HashSet* objects can be configured in order to not use the standard comparer for equality). For semantic reasons, overriding the *Equals()* method also requires overriding the ubiquitous *GetHashCode()* function. Accordingly, equal *Individual* objects will be stored in the same slots in the *HashSet* values of the *PropertyValues* field.

Do understand that for the given scenario a user defined comparer for *Individual* objects would have to take into account 1) the RDF type of the current *Individual* object (i.e. the subject of the RDF statement), 2) the imposed maximum cardinality restriction on property  $R$  on class  $C$ , and 3) the fact that the *Individual*  $i$  that has to be compared for equality with existing property values is also being added as a value for property  $R$  of the current *Individual*. In particular, the hash code that is computed for *Individual*

objects would have to vary based on their (tentative) usage as property values! If such a hash function would have been implemented for *Individual* objects the *MaxCardinalityRestriction* checks could be facilitated as shown in lines 24 to 32. A *MaxCardinalityRestriction* is enforced if the current *Individual* represents the same RDF type as given by the *Type* property of the restriction (note that the given code is incomplete with respect to subsumption!). If there are values for the property to which the *MaxCardinalityRestriction* applies then the LINQ expression<sup>5</sup> in line 27 is used to query and count all *Individual* objects with the same hash value. An exception is thrown if there are more different property values than allowed by the current *MaxCardinalityRestriction*.

If one wanted to allow for *qualified* number restrictions, the same complexity as for the implementation of the hash function would apply plus the handling of the typing information of the property values (recall that the implementation of the *Types* property of an *Individual* object has to take into account the usage of this object as a property value of different objects).

On a related note it should be clear that each OWL concept constructor discussed above is not an island unto itself. In fact, the basic Description Logic  $\mathcal{AL}$  is widely considered to be the minimal combination of a set of concept constructors that is of practical use. For the sake of conciseness, the presented C# code snippets do little in the way of combination. For settings that include several OWL concept constructors plus OWL concept axioms (e.g., *equivalentClass*, *disjointWith*, *subClassOf*) structural subsumption algorithms, which simply compare the syntactic structure of concept descriptions, are known to be incomplete (i.e. negation and disjunction cannot be handled). Instead, tableau-based algorithms have turned out to be very successful [SS91]. Eventually, the *Individual* objects as introduced above would be subject to such full-fledged reasoning algorithms. In contrast, ontological reasoning is inherently available for ontological concepts and individuals in Zhi#.

---

<sup>5</sup>A note on LINQ syntax: The expression form “from  $x$  in  $l$  select  $y$ ” denotes the computation of a new list  $l'$  from the given list  $l$  such that, each element  $x$  of  $l$  is mapped to an element  $y$  of  $l'$ . One can add *where* clauses so as to filter the list  $l$ .

#### 7.2.4 OWL role constructors and restrictions

In the Web Ontology Language, properties can be used to state relationships between individuals (i.e. object properties) or from individuals to data type values (i.e. datatype properties). The distinction between object and datatype properties is made because 1) not every Description Logic includes a concrete domain comprising data types and 2) data types are not subject to ontological reasoning.

The OWL vocabulary includes a number of identifiers to provide information concerning properties and their values. The *rdfs:domain* feature can be used to describe the type of individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual is inferred to belong to the class<sup>6</sup>. Similarly, the *rdfs:range* feature describes the type of individuals and data values that the property may have as its value. Range restrictions of OWL object properties are used for ontological reasoning, too. If an object property relates an individual to another individual, and the object property has a class as its range, then the other individual can be inferred to belong to the range class.

In contrast to OWL, object-oriented programming languages do not allow for self-sufficient property definitions that are not part of class types. Accordingly, the declaration of ad hoc relationships is not supported either. Finally, statically typed object-oriented programming languages do not facilitate type inference based on property domain and range restrictions. The programming effort that is required to mimic those fundamental OWL modeling features in object-oriented programming languages was already indicated in the previous subsection as part of the discussion of the top level concept. This subsection will carry on scrutinizing how additional OWL property features could be implemented in plain C#. As in the previous subsection, the conventional solutions will be compared to Zhi#'s inherent support for OWL properties and ontological reasoning.

---

<sup>6</sup>A note on OWL domain and range descriptions: In OWL, a sequence of elements without an explicit operator represents an implicit conjunction. Accordingly, the domain and range of an OWL property is the conjunction of the given concept descriptions.

**Equivalent properties**  $R_i \sqsubseteq R_j, 1 \leq i, j \leq n$  ( $\mathcal{H}$ ). In OWL, equivalent (i.e. synonymous) properties can be declared using the *equivalentProperty* feature. Equivalent properties relate one individual to the same set of other individuals. From this a reasoner can deduce that if an individual is related to another individual by a property, then the individual is also related to the other individual by all declared equivalent properties and vice versa. Accordingly, a reasoner will also deduce that equivalent properties subsume each other. The following C# code implements the observable behavior of equivalent property declarations for the well-tried *Individual* type introduced in the previous subsection (unused members are omitted for brevity, only object properties are considered). Read access to the *Individual* object's property values is implemented as a C# indexer<sup>7</sup>. The value for a property of an *Individual* object is not only the set of explicitly declared values but the union of all values for all equivalent properties, which the auxiliary *GetEquivalentProperties()* function yields.

```

1 public class Individual {
2     private static HashSet<string> GetEquivalentProperties(string p) {
3         [...] // Note: equivalence relation is reflexive!
4     }
5     private readonly Dictionary<string, HashSet<Individual>> PropertyValues =
6         new Dictionary<string, HashSet<Individual>>();
7     public HashSet<Individual> this[string p] {
8         get {
9             var values = new HashSet<Individual>();
10            foreach (var ep in GetEquivalentProperties(p)) {
11                values.UnionWith(PropertyValues[ep]);
12            }
13            return values;
14        }
15    }
16 }

```

<sup>7</sup>A note on C#: Indexers allow you to index a class or a struct in the same way as an array. The *get* and *set* accessor bodies of an indexer are similar to a method body.

Recall that the implementation of the union operation on the used hash sets would require the implementation of ontological equality for *Individual* objects if one wants to consider only one single representative for a set of equivalent individuals.

Also note how the deductions facilitated by *equivalentProperty* declarations are used to make implicit knowledge (in a knowledge base) explicit. Ponder the effect of ontological reasoning in this place on the implementation of OWL concept descriptions in the previous subsection. For example, the implementation of the *someValuesFrom* concept constructor would have to be modified to take into account equivalent properties.

As of today, in computer science, the term “intelligent” mainly refers to the ability to automatically infer implicit consequences from explicitly represented knowledge. This described simple form of artificial intelligence is readily available in Zhi# as shown below. Subject *s* will have object *o* as a value for the equivalent property *ep* despite the fact that this statement has never been explicitly added to the knowledge base.

```

1 #Thing s = [...], o = [...];
2 s.#p = o;
3 #Thing[] values = s.#ep; // Individual o is also a value for property ep

```

**Subproperties**  $R_1 \sqsubseteq R_2$  ( $\mathcal{H}$ ). Equivalent properties subsume each other. Property hierarchies can be created by making nonsymmetric *rdfs:subPropertyOf* statements that a property is a subproperty of one or more other properties. A reasoner can deduce that if an individual is related to another by a subproperty, then it is also related to the other by the respective super-property. The C# code for an isolated implementation of OWL subproperties is almost identical to the previous C# code snippet for equivalent properties. The only difference is that the *GetEquivalentProperties()* method must be replaced by a function that yields the subproperties for a given property. Again, the also isolated implementations of OWL concept constructors described in the previous subsection would have to be extended for settings that include property hierarchies. This would analogously be the case for all remaining role constructors described below. Recall that OWL DL

comprises *all* concept constructors mentioned above and *all* role constructors discussed in this subsection, which would eventually require the (re-)implementation of full-fledged reasoning algorithms.

In Zhi#, no additional user defined code is required to use role hierarchies programmatically. In the following code snippet, object *o* is a value for properties *sp* and *p* of subject *s*, where *sp* is a subproperty of *p*.

```

1 #Thing s = [...], o = [...];
2 s.#sp = o;
3 #Thing[] values = s.#p; // Individual o is also a value for property p

```

**Symmetric properties**  $R \sqsubseteq R^-$  ( $\mathcal{HI}$ ). OWL object properties can be declared symmetric. If a property *R* is symmetric, then if  $\langle x, y \rangle \in R^{\mathcal{I}}$ , then  $\langle y, x \rangle \in R^{\mathcal{I}}$  (i.e. a reasoner will deduce that role *R* is subsumed by its inverse).

Implementing inverse roles requires bidirectionally navigable links between related objects. For the *Individual* type shown below the *AddPropertyValue()* method is implemented such that the given individual is added as a value for the given property of the current subject *and* the current subject is itself added as a value for the given property of the given individual. The read only indexer yields for a given property the union of explicitly added values and the set of individuals that are related to the current individual by symmetric properties. The auxiliary function *IsSymmetric()* determines if a property is symmetric. In the given implementation two separate property value dictionaries are used in order to facilitate the removal of inferred values of symmetric properties. If a statement of the form  $R(s, o)$ , where *R* shall be a symmetric property, is removed, then a previously inferred triple  $R(o, s)$  has to be removed, too. Still, possible explicitly added statements  $R(o, s)$  must be preserved (i.e. an inferred statement  $R(s, o)$  would still be present, too). In the same vein, ontology management systems typically use two stores for explicitly added and inferred statements. Here, the code for removing property values is omitted for the sake of brevity.



```

1 public class Individual {
2     private static bool IsSymmetric(string p) {
3         [...] // Determines if property p is symmetric
4     }
5     private readonly Dictionary<string, HashSet<Individual>> PropertyValues =
6         new Dictionary<string, HashSet<Individual>>();
7     private readonly Dictionary<string, HashSet<Individual>>
8         SymmetricPropertyValues = new Dictionary<string, HashSet<Individual>>();
9     private void AddSymmetricPropertyValue(string p, Individual i) {
10        if (SymmetricPropertyValues.ContainsKey(p)) {
11            SymmetricPropertyValues[p].Add(i);
12        }
13        else {
14            SymmetricPropertyValues.Add(p, new HashSet<Individual> {i});
15        }
16    }
17    public void AddPropertyValue(string p, Individual i) {
18        if (PropertyValues.ContainsKey(p)) {
19            PropertyValues[p].Add(i);
20        }
21        else {
22            PropertyValues.Add(p, new HashSet<Individual> {i});
23        }
24        if (IsSymmetric(p)) {
25            i.AddSymmetricPropertyValue(p, this);
26        }
27    }
28    public HashSet<Individual> this[string p] {
29        get {
30            var values = new HashSet<Individual>();
31            values.UnionWith(PropertyValues[p]);
32            values.UnionWith(SymmetricPropertyValues[p]);
33            return values;
34        } } }

```

In Zhi# programs, symmetry traits of ontological roles are automatically obeyed as shown below. For a symmetric property  $p$  the *values* array will be initialized in line 3 with value  $s$  for property  $p$  of individual  $o$ . The inferred triple  $p(o, s)$  is automatically removed as soon as  $o$  is removed as a value for property  $p$  of individual  $s$  in line 4.

```

1 #Thing s = [...], o = [...];
2 s.#p = o;
3 #Thing[] values = o.#p; // Individual s is a value for property p
4 s.#p.Remove(o);
5 values = o.#p; // Individual s is no longer a value for property p

```

**Inverse properties**  $R \sqsubseteq R_0^-$  ( $\mathcal{HL}$ ). If an OWL object property is declared symmetric it is subsumed by its inverse. In OWL, it is also possible to explicitly declare a property as the inverse of another. If property  $R$  is stated to be the inverse of property  $R_0$ , then if individual  $x$  is related to  $y$  by  $R_0$ , then  $y$  is related to  $x$  by role  $R$ . Accordingly, a reasoner will infer that role  $R$  is subsumed by the inverse of  $R_0$ .

In the code snippet shown below the *AddPropertyValue()* method of the *Individual* type is implemented such that, the given individual is added as a value for the given property of the current subject *and* the current subject is itself added as a value of the given individual for the inverse properties of the given property (i.e. bidirectionally navigable links). The read only indexer yields for a given property the union of explicitly added values and the set of individuals that are related to the current individual by inverse properties. The auxiliary function *GetInverseProperties()* yields for a given property the inverses. In the given implementation two separate property value dictionaries are used to facilitate the removal of inferred values of inverse properties. If a statement of the form  $R_0(s, o)$ , where  $R$  is the inverse of  $R_0$ , is removed, then a previously inferred triple  $R(o, s)$  has to be removed, too. Still, possible explicitly added statements  $R(o, s)$  will have to be preserved (i.e. an inferred statement  $R_0(s, o)$  would still be present, too). Here, the code for removing property values is omitted for the sake of brevity.

```

1 public class Individual {
2     private static HashSet<string> GetInverseProperties(string p) {
3         [...] // Returns the inverses of p
4     }
5     private readonly Dictionary<string, HashSet<Individual>> PropertyValues =
6         new Dictionary<string, HashSet<Individual>>();
7     private readonly Dictionary<string, HashSet<Individual>>
8         InversePropertyValues = new Dictionary<string, HashSet<Individual>>();
9     private void AddInversePropertyValue(string p, Individual i) {
10        if (InversePropertyValues.ContainsKey(p)) {
11            InversePropertyValues[p].Add(i);
12        }
13        else {
14            InversePropertyValues.Add(p, new HashSet<Individual> {i});
15        }
16    }
17    public void AddPropertyValue(string p, Individual i) {
18        if (PropertyValues.ContainsKey(p)) {
19            PropertyValues[p].Add(i);
20        }
21        else {
22            PropertyValues.Add(p, new HashSet<Individual> {i});
23        }
24        foreach (string ip in GetInverseProperties(p)) {
25            i.AddInversePropertyValue(ip, this);
26        }
27    }
28    public HashSet<Individual> this[string p] {
29        get {
30            var values = new HashSet<Individual>();
31            values.UnionWith(PropertyValues[p]);
32            values.UnionWith(InversePropertyValues[p]);
33            return values;
34        } } }

```

In Zhi# programs, values of inverse roles are automatically considered as shown below. For an inverse *ip* of property *p* the *values* array will be initialized in line 3 with value *s* for property *ip* of individual *o*. The inferred triple  $p(o, s)$  is automatically removed as soon as *o* is removed as a value for property *p* of individual *s* in line 4.

```

1 #Thing s = [...], o = [...];
2 s.#p = o;
3 #Thing[] values = o.#ip; // Individual s is a value for property ip
4 s.#p.Remove(o);
5 values = o.#p;           // Individual s is no longer a value for property ip

```

**Transitive properties  $\text{Trans}(R)$  ( $\mathcal{S}$ ).** OWL object properties may be stated to be transitive. If a property *R* is transitive, then if  $\langle x, y \rangle \in R^{\mathcal{I}}$  and  $\langle y, z \rangle \in R^{\mathcal{I}}$ , then  $\langle x, z \rangle \in R^{\mathcal{I}}$  (both OWL Lite and OWL DL impose the side condition that transitive properties and their super-properties cannot have a *maxCardinality* 1 restriction, which would make OWL Lite and OWL DL undecidable languages).

The C# code snippet shown below implements transitive properties for the *Individual* type. The auxiliary function *IsTransitive()* determines if a property is transitive. The read only indexer is implemented such that, the result set is recursively populated for all transitive properties.

The attempted solution to implement transitive properties in plain C# may appear sufficient at a first glance. However, as for all mimicked concept and role constructors the suggested code would have to be provided (i.e. duplicated) for all objects under consideration assuming that one does not want to devise a rooted class hierarchy with a top level *Individual* type. Also, ponder how the suggested *Individual* observables would have to be used by client code. Method invocations and indexer accesses would be required in C# instead of intuitive member access as provided for ontological individuals in Zhi#. Moreover, ontological roles are addressed by their names as *System.String* objects, which is not type-safe on the ontology level.

```

1 public class Individual {
2     private static bool IsTransitive(string p) {
3         [...] // Determines if property p is transitive
4     }
5     private readonly Dictionary<string, HashSet<Individual>> PropertyValues =
6         new Dictionary<string, HashSet<Individual>>();
7     private void GetPropertyValues(string p, HashSet<Individual> values) {
8         values.UnionWith(this[p]);
9         foreach (var i in values) {
10            i.GetPropertyValues(p, values);
11        }
12    }
13    public HashSet<Individual> this[string p] {
14        get {
15            var values = new HashSet<Individual>();
16            values.UnionWith(PropertyValues[p]);
17            if (IsTransitive(p)) {
18                foreach (var i in values) {
19                    i.GetPropertyValues(p, values);
20                }
21            }
22            return values;
23    } } }

```

In Zhi#, transitivity is a native feature of ontological roles as shown below. In contrast to isolated implementations of transitive properties in plain C#, genuine ontological roles in Zhi# can accumulate a number of traits (e.g., transitivity and symmetry). These traits can be encoded in original OWL syntax and imported into Zhi# programs.

```

1 #Thing x = [...], y = [...], z = [...];
2 x.#p = y;
3 y.#p = z;
4 #Thing[] values = x.#p; // Individual z is a value for property p, too

```

**Functional properties**  $\top \sqsubseteq \leq 1R (\mathcal{F})$ . The Web Ontology Language supports the notion of *functional* (i.e. *unique*) properties. If a property is functional, then it has no more than one value for each individual. Recall that ontological equality is based on the represented entities. Accordingly, there can be a number of objects being related to a subject by a functional property. As a consequence, all related objects will be inferred to be identical. The described kind of functionality cannot be easily implemented as a local extension of .NET objects such as the *Individual* type. In particular, central housekeeping code would be required to merge sets of individuals that are each inferred to be identical based on relations with different subjects (e.g., two individuals  $s_1$  and  $s_2$  may be related by a functional property  $p$  to objects  $\{o_1, o_2\}$  and  $\{o_2, o_3\}$ , respectively). A global *disjoint-set* data structure and a *union-find* implementation would be necessary to manage partitions of identical individuals. As a result, implementations of the *Equals()* method or the  $==$  operator of such “ontological” .NET objects would be required to refer to the central housekeeping code. In addition to the acquired code complexity, the garbage collection of such objects would most likely be compromised due to perpetual references in the disjoint-set data structure.

Functional properties along with their effects on ontological reasoning and object identity are provided natively in Zhi#. In the following code snippet the comparison expression  $x==z$  will evaluate to true since individuals X, Y, and Z will be inferred to be identical based on the assignment statements in lines 3 to 6 for a functional property  $p$ .

```

1 #Thing s = [...], t = [...];
2 #Thing x = [...], y = [...], z = [...];
3 s.#p = x;
4 s.#p = y;
5 t.#p = y;
6 t.#p = z;
7 bool b = x == z; // Individuals X, Y, and Z are identical

```

**Inverse functional properties**  $\top \sqsubseteq \leq 1R^- (\mathcal{IF})$ . OWL object properties can be stated to be *inverse functional* (i.e. *unambiguous*). If a property is inverse functional then the inverse of the property is functional. Thus, the inverse of the property has at most one value for each individual. Accordingly, a reasoner will deduce that if two individuals have the same value for an inverse functional property, then those two individuals refer to the same entity in the described world.

As for functional properties, the implementation of unambiguous properties for .NET objects could barely be facilitated as a local extension of the *Individual* type. Also, the implementation of object equality as a result of the use of objects with inverse functional properties would certainly require housekeeping code similar to the one necessary for inverse properties. In contrast, unambiguous properties can easily be defined using the Web Ontology Language as in the following OWL snippet. The inverse functional property definition can then be imported into Zhi# programs as shown below.

The same goes for all OWL DL concept and role constructors: Instead of mimicking ontological behavior with .NET objects, OWL description languages can be (visually) modeled using tailored knowledge acquisition tools and right away be leveraged in Zhi# programs.

```

1 <rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#" [...] >
2   <owl:ObjectProperty rdf:ID="p">
3     <rdf:type rdf:resource="http://www.w3.org/2002/07/           ↔
4                               owl#InverseFunctionalProperty"/>
5   </owl:ObjectProperty>
6 </rdf:RDF>

```

```

1 #Thing s = [...], t = [...], o = [...];
2 s.#p = o;
3 t.#p = o;
4 bool b = s == t; // Individuals s and T are identical

```

**Domain and range restrictions**  $\geq 1R \sqsubseteq C, \top \sqsubseteq \forall R.C$ . Previous paragraphs already alluded to the fact that in the Web Ontology Language individual inclusion is inferred from the use of individuals as subjects and objects of ontological roles. Such dynamic type changes necessitate the introduction of user defined proxy types (e.g., *Individual*) for ontological individuals. The following code snippet is an attempt to mimic property domain and range restriction based type inference for *Individual* objects.

```

1 public class Individual {
2     static HashSet<string> GetDomainOfProperty(string p) {
3         [...] // Gets the domain classes of the given property
4     }
5     static HashSet<string> GetRangeOfProperty(string p) {
6         [...] // Gets the range classes of the given property
7     }
8     readonly Dictionary<string, HashSet<Individual>> PropertyValues = ↪
9         new Dictionary<string, HashSet<Individual>>();
10    readonly HashSet<string> ReferringProperties = new HashSet<string>();
11    readonly HashSet<string> _Types = new HashSet<string> {"Thing"};
12    public HashSet<string> Types {
13        get {
14            var types = new HashSet<string>(_Types);
15            foreach (var p in PropertyValues.Keys) {
16                types.UnionWith(GetDomainOfProperty(p));
17            }
18            foreach (var p in ReferringProperties) {
19                types.UnionWith(GetRangeOfProperty(p));
20            }
21            return types;
22        }
23    }
24    public bool Is(string T) {
25        return Types.Contains(T);
26    } }

```



The set of type names that is yielded by the *Types* property is complemented to include the domain and range classes of those properties that relate the given individual as a subject and as an object, respectively. Note that for the sake of conciseness subsumption is not considered. In order to be complete in the presence of subsumption, the *GetDomainOfProperty()* and *GetRangeOfProperty()* functions would have to yield *all* named classes that are subsumed by the domain and range restriction of the input property. Also, the given code does only provide for object properties.

Runtime type checks for *Individual* objects could be facilitated in C# programs as follows. The use of the *Is()*-function in line 3 requires that the infrastructure described above is available for object *s* in order to provide ontological type inference. Still, the use of string literals to denote property and class names is error-prone and all but type-safe.

```

1 Individual s = [ ... ] , o = [ ... ];
2 s.AddPropertyValue("p", o);           // if the domain of role p includes concept A...
3 if (s.Is("A")) { [ ... ] }           // ...then individual s will be an A

```

Note that all described implementations of “ontological” behaviors for .NET objects are isolated and do only consider selected OWL modeling features. For example, the *Individual* object above does not take into account transitive roles or role hierarchies. Also, none of the presented solution attempts implements error handling for ontological inconsistencies (e.g., the *Types* property yields both a class as well as its negation).

In Zhi# programs, the required ontology management infrastructure is readily available for imported concept and role descriptions. There is no need to (re-)implement ontological behavior for .NET objects. Moreover, Zhi# facilitates type-checking and autocompletion for ontological concepts and roles. The next subsection will compare Zhi# programs with conventional Java code that takes advantage of available off-the-shelf ontology management systems. It will be shown that the presented Java code will still be more error-prone than corresponding Zhi# programs due to the different conceptual bases of the Web Ontology Language and object-oriented programming languages.

### 7.2.5 OWL APIs vs. OWL types

In the previous two subsections solution attempts were presented to implement ontological behavior for .NET objects and to use those conventional .NET objects in place of ontological knowledge bases. The author trusts that the presented C# implementations dispel the view that it was easy to combine the effects of reasoning algorithms for expressive Description Logics with programming language type definitions. Also, it should have become clear that for fundamental programming tasks such as equality tests and runtime type checks the programmability of the exemplary user-defined *Individual* proxy type is inferior compared to Zhi#'s native support for ontological concepts and roles.

In this subsection, the programmatic use of the Protégé knowledge base framework [Sta06] and the Jena Semantic Web Framework [HP 04] will be compared to Zhi# programs. Protégé and Jena will be used for comparison because both frameworks are widely used in practice and are tailored to the Web Ontology Language. Unfortunately, both APIs are only available in Java. This is why Java client code will be presented for the conventional solutions.

The author will leave it to the reader to assess hybrid approaches that propose methodological means of integrating OWL models, which are managed by frameworks such as Protégé and Jena, with computer programs (see Puleston et al. [PPC08] for an OWL-Java combination). The author knows from experience that integration shortcomings of hybrid approaches can barely be compensated by methodologies, which usually put the burden to behave compliantly to the ontology on the programmer.

The Protégé knowledge base framework comprises the Protégé OWL API, which is an open-source Java library for the Web Ontology Language and RDF(S). It provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to perform reasoning based on Description Logic inference engines (in contrast, the Protégé ontology editor is more general in that it supports a variety of different formalisms such as frames).

The Jena Ontology API particularly supports the Web Ontology Language, too. In contrast to Protégé, the Jena Semantic Web framework includes a number of own reasoner components for OWL Lite and OWL DL. However, for the following tasks Protégé, Jena, and Zhi#'s OWL plug-in were configured to use the Pellet OWL DL reasoner [Pel06], which, at the time of this writing, is probably the most widely used software providing standard reasoning services for OWL.

The ontology that will be used for the remainder of this subsection specifies the description language shown in Table 7.1. Concepts named  $A$ ,  $B$  and  $C$ , and the top level concept  $\top$  will be used. Concepts  $B$  and  $C$  are declared to be disjoint. The functional object role  $R$  relates subjects of type  $A$  to objects of type  $B$ . The datatype role  $U$  relates individuals to values of type  $xsd\#positiveInteger$ .

Table 7.1: TBox used for comparison

<b>Concepts</b>	$A, B \sqsubseteq \neg C$
<b>Object role</b>	$\geq 1R \sqsubseteq A, \top \sqsubseteq \forall R.B, \top \sqsubseteq \leq 1R$
<b>Datatype role</b>	$\top \sqsubseteq \forall U.xsd\#positiveInteger$

The Zhi# approach will be compared to Protégé and Jena based on the following programming tasks, which are all imminent for ontology-based applications.

**Task 1** Make an ontology available in a computer program.

**Task 2** Create named individual instances  $A$ ,  $B$ ,  $C$ , and  $O$  of concepts  $A$ ,  $B$ ,  $C$ , and  $\top$ .

**Task 3** Add individual  $O$  as a value for property  $R$  of individual  $A$ .

**Task 4a)** List the RDF types of individual  $O$ .

**Task 4b)** Check whether individual  $O$  is included by concept description  $B$ .

**Task 4c)** List all individuals that are in the extension of concept description  $B$ .

**Task 5** Add individual C as a value for property *R* of individual A, which causes an inconsistent ABox since concept descriptions *B* (range of role *R*) and *C* (which includes C) are disjoint.

**Task 6** Add individual B as a value for property *R* of individual A and test if individuals O and B are equal (i.e. is there an inferred *sameAs*(O, B) statement in the ontology?).

**Task 7** Add integer literals 23, -23, and string literal “NaN” as values for property *U* of individual O, where -23 and “NaN” are invalid values for the given TBox.

In the following paragraphs it will be described how each of the above tasks can be accomplished using Protégé, Jena, and the Zhi# programming language. The presented code snippets will indicate the improved programmability of ontologies using Zhi# and the shortcomings of the conventional solutions in practice.

**Task 1.** Ontological description languages and assertional knowledge reside in ontologies that can exist in different forms such as .owl-files and RDF triples in a relational database. Protégé and Jena facilitate programmatic access to ontologies by providing an in-memory ontology model. The following code snippets show how these ontology models are initialized in Java programs. The referenced *Evaluation.owl* file contains the TBox specified in Table 7.1. The defined base URI is *http://www.zhimantic.com/eval*.

Both the Protégé as well as the Jena solution require extra set-up of the external Pellet OWL reasoner. As will be seen later, Protégé can use Pellet only through its DIG interface, which results in a limited number of available tell and ask operations. In Zhi# programs, ontology models do not need to be set up manually. Instead, required .owl-files are passed into the Zhi# compiler as arguments exactly like referenced .NET assemblies. The same goes for XML data type definitions contained in .xsd-files. The Zhi# *import* statement abstracts from the storage location of the ontology (just like C# *using* statements abstract from the referenced .NET assemblies). In addition, in Zhi# programs, imports of external namespaces are checked such that, a compile-time error occurs if no type definitions exist

in the given namespace for the given type system. In contrast, the set-ups of the Protégé and Jena ontology models do not give an indication about available external namespaces nor do they constrain the use of the ontology models to concepts and roles that are defined within a particular namespace, which is less documentary.

Listing 7.1: Importing the ontology (PROTÉGÉ)

```

1 OWLModel m = ProtegeOWL.createJenaOWLModelFromInputStream(      ↪
2     new FileInputStream("Evaluation.owl"));
3 ProtegeOWLReasoner reasoner = ReasonerManager.getInstance().getReasoner(m);
4 reasoner.setURL("http://localhost:8081");

```

Listing 7.2: Importing the ontology (JENA)

```

1 OntModel m =                                                    ↪
2     ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC);
3 m.read(new FileInputStream("Evaluation.owl"), "");

```

Listing 7.3: Importing the ontology (ZHI#)

```
import OWL ont = http://www.zhimantic.com/eval;
```

**Task 2.** In the following listings, individual instances  $A$ ,  $B$ ,  $C$ , and  $O$  of the named concept descriptions  $A$ ,  $B$ ,  $C$ , and the top level concept  $\top$  are created. The Protégé OWL API features a namespace manager that allows for the use of unqualified concept and individual names. In the Jena-based program concept names must be fully qualified.

Both with Protégé as well as Jena the creation of individuals in a knowledge base is facilitated as an operation on a surrogate ontology model, where concept and individual names are provided as untyped string arguments. Consequently, there is no way to statically check that 1) the given strings denote valid concept and individual names according to OWL's naming scheme and 2) concept descriptions with the given names exist in the TBox of the referenced ontology. In fact, automatic debugging techniques such as *delta debugging* [Zel05] regularly find exactly those kinds of errors where external resources that

are not subject to built-in static type checking are incorrectly referenced by string literals (e.g., SQL query strings). Moreover, programmers may accidentally reference elements of an ontology that are defined in inappropriate namespaces for the given application.

In contrast, concept names in *Zhi#* programs are strongly typed. A compile-time error occurs if no such concept descriptions exist in the imported TBoxes. Namespace prefixes that are bound to imported namespaces are used to abbreviate concept names in *Zhi#* program code. Individual names can be given in a fully qualified form (lines 1 and 2) or their containing namespaces are inferred from the names of the used concept descriptions (lines 3 and 4). Also, the *Zhi#* editor of the Eclipse-based frontend auto-completes the namespaces of individual names within *new*-statements. Constant string expressions such as string literals are checked to comply with OWL's URI-based naming scheme for individual names.

Listing 7.4: Creating individuals (PROTÉGÉ)

```

1 OWLIndividual o = m.getOWLThingClass().createOWLIndividual("o");
2 OWLIndividual a = m.getOWLNamedClass("A").createOWLIndividual("A");
3 [...] // analogous for individuals B and C

```

Listing 7.5: Creating individuals (JENA)

```

1 Individual o = m.getOntClass("http://www.w3.org/2002/07/owl#Thing").    ↪
2   createIndividual("http://www.zhimantic.com/eval#o");
3 Individual a = m.getOntClass("http://www.zhimantic.com/eval#A").      ↪
4   createIndividual("http://www.zhimantic.com/eval#A");
5 [...] // analogous for individuals B and C

```

Listing 7.6: Creating individuals (ZHI#)

```

1 #owl#Thing o = new #owl#Thing("http://www.w3.org/2002/07/owl#o");
2 #ont#A a = new #ont#A("http://www.zhimantic.com/eval#A");
3 #ont#B b = new #ont#B("B"); // Fully qualified individual names are
4 #ont#C c = new #ont#C("c"); // inferred from the used concept names

```

A note on the use of Protégé and Jena’s ontology models: Despite the fact that Java’s interface polymorphism can be used to refer with *OWLModel* and *OntModel* variables to differently configured knowledge base access points, the client code will always be bound to the Protégé and Jena framework, respectively. In contrast, even for compiled Zhi# programs the entire ontology management infrastructure may be replaced because in Zhi# program code no commitments are made on the use of a particular ontology management system. Zhi#’s OWL runtime library can be implemented and substituted to use arbitrary knowledge base frameworks that provide support for OWL DL.

On a related note, the configuration information for Zhi#’s OWL runtime library (e.g., the access point to a remote ontology management system) is provided in a Zhi# application’s *App.config* file. Hence, Zhi# program code only contains logical instructions on the ontology level but no information about 1) the .owl-files that contain the ontology (see above), 2) the ontology management API, and 3) the configuration of the ontology management system.

**Task 3.** The following code snippets show how in the managed knowledge base individual *O* can be added as a value for property *R* of individual *A*. Note that Protégé and Jena’s ontology models do, of course, not provide dedicated methods for adding values for property *R* (e.g., *addR()*). Instead, a reference to the role object is passed in as a second argument to the general *addPropertyValue()* and *addProperty()* methods (this subsection will wrap up with a brief allusion to Protégé’s OWL-to-Java code generator plug-in that generates Java classes and named getter and setter-methods for ontological concepts and roles, respectively). The Zhi# programming language facilitates the declaration of RDF triples in a knowledge base via usual member access (see Subsection 6.1.2 for the semantics of such assignment statements). Again, role names are statically checked to be defined in the referenced TBoxes and autocompletion for roles names is provided by the Eclipse-based Zhi# editor. Task 5 will show how the Zhi# compiler is also able to detect inconsistent triple declarations because of disjoint concept descriptions.

Listing 7.7: Adding object property values (PROTÉGÉ)

```
a.addPropertyValue(m.getOWLObjectProperty("R"), o);
```

Listing 7.8: Adding object property values (JENA)

```
a.addProperty(m.getObjectProperty("http://www.zhimantic.com/eval#R"), o);
```

Listing 7.9: Adding object property values (ZHI#)

```
a.#ont#R = o;
```

**Task 4a).** One of the most common tasks in ontology-based applications is to query the concept descriptions that include a given individual. In object-oriented programming languages, this corresponds to dynamic type checks. The code snippets shown below list the RDF types of individual *o* that was just used in Task 3 as a value for property *R*. Individual *o* would therefore, because of the range restriction of role *R*, be inferred to be not only a *Thing* but also a *B*.

Listing 7.10: Listing RDF types (PROTÉGÉ)

```
1 for (Object T : reasoner.getIndividualTypes(o, null)) {
2   System.out.println(((OWLClass) T).getURI());
3 }
```

Listing 7.11: Listing RDF types (JENA)

```
1 for (Iterator it = o.listRDFTypes(false); it.hasNext();) {
2   System.out.println(((Resource) it.next()).getURI());
3 }
```

Listing 7.12: Listing RDF types (ZHI#)

```
1 foreach (string T in o.Types) {
2   Console.WriteLine(T);
3 }
```



The Protégé-based code requires the explicit use of a reference to a *ProtegeOWL-Reasoner* object, which should preferably be transparent to users of the ontology. As in Jena-based code, the yielded type objects must be manually downcasted to *OWLClass* and *Resource* objects, respectively. It is conceivable, however, that future versions of Protégé and Jena will make use of generic container and iterator classes, which can be parameterized appropriately. It can be also be considered to be a flaw in the API design that the *listRDFTypes()* method of Jena's *Individual* objects returns a (raw) *Iterator* instead of an iterable container object that could better be used with a *for-each*-loop. The *Types* property that is provided for individual objects in Zhi# programs returns an enumerable array of strings, which can conveniently be used with a *for-each*-loop and whose elements do not need to be downcasted.

**Task 4b).** In many applications, the basic reasoning service to compute the inferred types of ontological individuals, which is available for OWL knowledge bases, will be used to query the knowledge base whether an individual is in the extension of one particular concept description. This corresponds to the use of the *instanceof* and *is*-operator in Java and C#, respectively. In the following code snippets, individual *O* is dynamically checked to be subsumed by concept description *B*. Unfortunately, there is no support neither in Protégé nor in Jena to determine whether an ontological individual is in the extension of one particular concept description. Instead, such a test must be handcrafted similarly as shown below. Again, the use of the *getIndividualTypes()* and *listRDFTypes()* methods suffers from a lack of type safety due to the required use of downcasts. More severely, the concept names under consideration are denoted by string literals, which are untyped on the ontology level. Thus, it is not only tedious for programmers to cope with raw iterator and container objects but also error-prone to base a type check on purely syntactical string comparisons. Note that the *getIndividualTypes()* and *listRDFTypes()* methods return all concepts that include the individual and not only the most specific one. Hence, it is sufficient to consider only the single concept name one is interested in.

Listing 7.13: Individual inclusion (PROTÉGÉ)

```

1 boolean bIsB = false;
2 for (Object cls : reasoner.getIndividualTypes(o, null)) {
3     bIsB = "http://www.zhimantic.com/eval#B".equals(           ↪
4         ((OWLClass) cls).getURI()) ? true : bIsB;
5 }
6 System.out.println(o.getURI() + " is" + (bIsB ? "" : " not") + " a 'B'!");

```

Listing 7.14: Individual inclusion (JENA)

```

1 boolean bIsB = false;
2 for (Iterator it = o.listRDFTypes(false); it.hasNext();) {
3     bIsB = "http://www.zhimantic.com/eval#B".equals(           ↪
4         ((Resource) it.next()).getURI()) ? true : bIsB;
5 }
6 System.out.println(o + " is" + (bIsB ? "" : " not") + " a 'B'!");

```

Listing 7.15: Individual inclusion (ZHI#)

```

1 Console.WriteLine(o + " is" + (o is #ont#B ? "" : " not") + " a 'B'!");

```

In Zhi# programs, it is particularly easy to use the *is*-operator to determine whether an individual is in the extension of a particular concept description. The given Zhi# code is completely statically type-checked both on the programming language and the ontology level. Moreover, the Zhi# compiler will detect if an individual will never be included by a concept description that is disjoint with or complementary to its asserted type.

**Task 4c).** Finally, one may be interested in all individuals that are in the extension of a given concept description. The two Java code snippets shown below depend on typo-prone string literals to denote the name of concept description *B*. Again, the Protégé-based program uses an explicit reference to the Pellet OWL reasoner and requires downcasting the objects that are yielded by the iterator expression. The Jena API requires the explicit use of a raw iterator object including unsafe downcasts.

Listing 7.16: Concept extension (PROTÉGÉ)

```

1 for (Object i : reasoner.getIndividualsBelongingToClass(      ↪
2                                     m.getOWLNamedClass("B"), null)) {
3     System.out.println(((OWLIndividual) i).getURI());
4 }

```

Listing 7.17: Concept extension (JENA)

```

1 for (Iterator it =                                          ↪
2     m.getOntClass("http://www.zhimantic.com/eval#B").listInstances(); ↪
3     it.hasNext();) {
4     System.out.println(((Individual) it.next()).getURI());
5 }

```

Listing 7.18: Concept extension (ZHI#)

```

1 foreach(#ont#B v in #ont#B.Individuals) {
2     Console.WriteLine(v);
3 }

```

Both the Protégé as well as the Jena-based solution do neither provide type safety on the programming language level nor on the ontology level. In contrast, the Zhi# program is entirely statically type-checked. The read-only *Individuals* property is provided for static concept references and yields the extension of the given concept. The *Individuals* property is generic in regards to the used concept description. In the given program, the enumerated objects are individual instances of concept *B*. Accordingly, variable *v* in the *for-each*-statement can be declared to be a *B* (instead of a bleak *java.lang.Object* as in the above Java programs). The Zhi# compiler would raise an error if the name of a too specific or a disjoint or complementary concept was used to declare variable *v*. As for the previous Zhi# code snippets, referenced concept names are statically checked to be defined within the imported namespaces and referenced ontologies. An implicit conversion operator is defined for individuals. It yields the fully qualified individual name as a *System.String* object, which is a valid argument for the *Console.WriteLine()* function.

**Task 5.** In OWL, property domain and range restrictions are used to infer the concept inclusions for ontological individuals that are related by ontological roles. In general, static type inference based on property domain and range restrictions will always be incomplete since it cannot be fully known at compile time which ontological roles will be used during program execution to relate the individual under consideration. It is, however, possible to statically identify ABoxes for which no models exists (i.e. that will *always* be inconsistent) for a given (immutable) TBox. One such indication is the use of an individual with an ontological role, where the individual is declared to be in the extension of a concept description that is disjoint with the role’s domain or range restriction (*every* ABox  $\mathcal{A} \cup \{C(a)\} \cup \{\neg C(a)\}$  is inconsistent). In order to treat the described inconsistency at compile time it is required to use in program text objects whose types reflect the RDF types of the represented ontological individuals. Typed role objects are necessary to represent ontological roles. Also, the programming language compiler must be aware of OWL’s consistency rules for ontological ABoxes. Unfortunately, the Protégé and Jena OWL APIs do only provide raw proxy types *OWLIndividual* and *Individual*, respectively, and only blank string objects are used to represent ontological role names as shown in the code snippets below. In contrast, in Zhi# programs, the declared RDF types of ontological individuals and the domain and range restrictions of used ontological roles are known at compile time. Also, the OWL compiler plug-in enforces ontological consistency rules on RDF triple declarations as shown in the following Zhi# code snippet.

Listing 7.19: Adding invalid object property values (PROTÉGÉ)

```
a.addPropertyValue(m.getOWLObjectProperty("R"), c);
```

Listing 7.20: Adding invalid object property values (JENA)

```
a.addProperty(m.getObjectProperty("http://www.zhimantic.com/eval#R"), c);
```

Listing 7.21: Adding invalid object property values (ZHI#)

```
a.#ont#R = c; // Zhi# compile-time error: "Disjoint concept descriptions!"
```

**Task 6.** In OWL, ontological individuals can be explicitly stated to be identical and they can be inferred to be the same. In both cases, an OWL knowledge base will contain a *sameAs* statement. Accordingly, equality tests of ontological individuals cannot be founded on the (reference) equality of Protégé and Jena's *OWLIndividual* and *Individual* objects (see Subsection 6.1.2). Unfortunately, the DIG interface that is implemented by Protégé OWL reasoner objects does not provide a function to test two individuals for equality. Instead, handcrafted code would be necessary to check the presence of *sameAs* statements in a knowledge base. The Jena *Individual* interface contains the *isSameAs()* method to test whether the ontological individual represented by the argument is identical to the individual represented by the host object. Strangely, *isSameAs()* always returns true for *null* arguments, which one may consider counterintuitive. More severely, the symmetry of the *sameAs* relation is broken because a *NullPointerException* is thrown if the host object of the *isSameAs* method invocation is *null*. In Zhi#, the equality (`==`) and inequality (`!=`) operators implement OWL's equality semantics for ontological individuals including the symmetry of the *sameAs* relation. In particular, `==` returns true if both operands are *null*; false if only one operand is *null*.

Listing 7.22: Testing individuals for equality (PROTÉGÉ)

```
1 a.addProperty(m.getOWLObjectProperty("R"), b);
2 // Testing two individuals for equality is not supported by the Protégé OWL API
```

Listing 7.23: Testing individuals for equality (JENA)

```
1 a.addProperty(m.getObjectProperty("http://www.zhimantic.com/eval#R"), b);
2 System.out.println("Individuals 'o' and 'b' are" +           ↪
3                     (o.isSameAs(b) ? "" : " not") + " identical!");
```

Listing 7.24: Testing individuals for equality (ZHI#)

```
1 a.#ont#R = b;
2 Console.WriteLine("Individuals 'o' and 'b' are" +           ↪
3                     (o == b ? "" : " not") + " identical!");
```

**Task 7.** In recent years, there have been ongoing plans to include a concrete domain of XML data types with the specification of prospective versions of the Web Ontology Language. However, currently available ontology management systems such as Protégé and Jena provide only very limited support for coercing range restrictions of OWL datatype properties. In the TBox given at the beginning of this subsection, the range of the OWL datatype property  $U$  is limited to positive integers. Because XML data types are not subject to ontological reasoning in the current version of the Web Ontology Language one might assume that only data values that can be considered positive integers are allowed as property values for  $U$  (i.e. an exception is raised at runtime at the latest).

Unfortunately, the Protégé OWL API is completely negligent in respect of datatype property range restrictions. The following program will execute without any error. Not only are values -23 and “NaN” accepted as positive integer values (which they are certainly not). Even an explicitly typed *RDFSLiteral* object, which in the given program is configured to take XSD *date* values, does not implement any dynamic checking to ensure that the provided literal value is in the lexical and value space of the given XML data type. As a result, the knowledge base will contain obviously invalid statements, which may subsequently lead to application failure.

Listing 7.25: Adding datatype property values (PROTÉGÉ)

```

1 o.addPropertyValue(m.getOWLDatatypeProperty("U"), 23);
2 o.addPropertyValue(m.getOWLDatatypeProperty("U"), -23);
3 o.addPropertyValue(m.getOWLDatatypeProperty("U"), "NaN");
4
5 RDFSDatatype xsdDate = m.getRDFSDatatypeByName("xsd:date");
6 OWLDatatypeProperty dateProperty =                               ↩
7     m.createOWLDatatypeProperty("dateProperty", xsdDate);
8 RDFSLiteral dateLiteral = m.createRDFSLiteral("23", xsdDate);
9 // literal value 23 is typed as an xsd:date!
10 o.setPropertyValue(dateProperty, dateLiteral);
11 RDFSLiteral myDate = (RDFSLiteral) o.getPropertyValue(dateProperty);

```

The Jena framework does not provide any static typing either nor are invalid datatype property values rejected upon declaration. However, in contrast to Protégé, subsequent queries fail if a knowledge base contains invalid statements. Hence, the acceptance of invalid datatype property values can hardly be considered a feature but rather a short-coming of off-the-shelf ontology management systems.

Listing 7.26: Adding datatype property values (JENA)

```

1 o.addProperty(m.getDatatypeProperty("http://www.zhimantic.com/eval#U"), ↔
2   m.createTypedLiteral("23", ↔
3     "http://www.w3.org/2001/XMLSchema#positiveInteger"));
4 // Values "-23" and "NaN" could be added as positive integers, too...
5 // ...but the following query would fail if invalid statements were added
6 for (Iterator it = o.listPropertyValues( ↔
7   m.getDatatypeProperty("http://www.zhimantic.com/eval#U")); ↔
8   it.hasNext();) {
9   Literal v = (Literal) it.next();
10  System.out.println(v.getValue() + " of type " + v.getDatatypeURI());
11 }

```

The Zhi# programming language boasts extensive compile-time support for XML data types. Assignments to XML data types are statically checked by the Zhi# compiler and so are assignments to OWL datatype properties. In Zhi# programs, OWL datatype properties can only be assigned values that are provable valid. The commented assignments in line 5 and 6 in the following Zhi# program would cause compile-time errors.

Listing 7.27: Adding datatype property values (ZHI#)

```

1 #xsd#positiveInteger xpi = 23;
2 #xsd#integer xi = -23;
3 #xsd#string xs = "NaN";
4 o.#ont#U = xpi;
5 //o.#ont#U = xi; // Compile-time error in Zhi#!
6 //o.#ont#U = xs; // Compile-time error in Zhi#!

```

**Protégé-OWL Java code.** The Protégé knowledge-base framework ships with an OWL-to-Java code generator, which produces a Java type hierarchy for a given OWL ontology. For each named concept description in the ontology an interface with the same name extending *OWLIndividual* is created. These interfaces are each implemented by class types inheriting from *DefaultRDFIndividual*. The purpose of the generated application specific Java code is to provide convenience classes and methods for easy use of the Protégé OWL API. The following Java interface definition is generated for the named concept description *A* of the description language given in Table 7.1.

```

1 import edu.stanford.smi.protegex.owl.model.*;
2 public interface A extends OWLIndividual {
3     B getR();
4     void setR(B newR);
5     boolean hasR();
6     RDFProperty getRProperty();
7 }

```

Note that the application specific interfaces do not fully abstract from the Protégé OWL API since type definitions such as *RDFProperty* still occur in generated method signatures. Also note that the generated proxy types do only provide for frame-like semantics of OWL property domain and range restrictions (cf. Subsection 6.1.2). The argument and return type of the *setR()* and *getR()* method, respectively, restricts property values to instances of the proxy type *B* (i.e. property values must be of the given type already *before* the property value declaration and cannot subsequently be inferred to be of that type). Furthermore, only the Java proxy type *A* defines methods to set and get values for property *R* (i.e. only *As* can have values for property *R*). The use of Java interface types in the generated code allows for multiple inheritance, which in turn facilitates the translation of the conjunction concept constructor. In fact, for a concept description  $I \equiv D \sqcap E$  a Java interface will be created as follows. This corresponds to the suggested use of Java types at the beginning of Subsection 7.2.3.

```
public interface I extends D, E { }
```



Unfortunately, the generated Java proxy code falls short of supporting the disjunction concept constructor. For a concept description  $F \equiv D \sqcup E$  the following unrelated Java interfaces will be generated. In particular, the Java code does not express the fact that based on the ontological concept descriptions  $I \equiv D \sqcap E$  and  $F \equiv D \sqcup E$  every  $D$  and  $E$  is also an  $F$  and consequently every  $I$  is an  $F$ , too.

```

1 public interface D extends OWLIndividual { }
2 public interface E extends OWLIndividual { }
3 public interface F extends OWLIndividual { }

```

Each generated Java type hierarchy is accompanied by a factory class as shown below. The singleton factory object can be used to create and answer instances of the represented OWL concepts in Protégé's *OWLModel*. The use of this interface is advantageous since the use of concept names can now be statically type checked using Java's built-in type system. Moreover, the factory class is the central book-keeping instance that can be used to query all instances of a given concept (cf. Subsection 7.2.3). No such auxiliary code is required to answer the extension of a static concept reference in Zhi# programs. Note that both the *getAllInstances()* methods in the generated Java factory class as well as the *Individuals* property of static concept references in Zhi# programs are generic with respect to the considered concept.

```

1 import edu.stanford.smi.protege.owl.model.*;
2 import java.util.*;
3 public class MyFactory {
4     private OWLModel owlModel;
5     public MyFactory(OWLModel owlModel) { [...] }
6     public A createA(String name) { [...] }
7     public A getA(String name) { [...] }
8     public RDFSNamedClass getAClass() { [...] }
9     public Collection<A> getAllInstances() { [...] }
10    public Collection<A> getAllInstances(boolean transitive) { [...] }
11    [...]
12 }

```

### 7.3 Macroscopic Evaluation

The Zhi# compiler framework was designed for easy extensibility and to foster the development of further compiler plug-ins to provide for additional external type systems. The following subsection will describe how existing techniques such as aspect-oriented programming and similar means to implement OCL invariants could be replaced by native OCL data types in Zhi# programs.

Furthermore, the complete Zhi# solution including the compiler framework and the XSD and OWL compiler plug-ins could be empirically evaluated by means of case studies and experiments that examine the impact of the use of the Zhi# tool suite on the development of knowledge-based and Semantic Web applications. In particular, hypotheses 2.1 and 2.2 (see Section 1.1) could be tested.

The directional hypothesis 2.1, stating that the use of native XML data types in Zhi# programs reduces the number of XSD-related runtime validation errors compared to the use of schema-agnostic types in different programming languages, may be tested by a controlled experiment with the number of possible runtime errors in written program code being the dependent variable. Independent variables – in a setting where XML data types are to be considered and used in a computer program – may be the number and complexity of XML data type definitions, the intended frequency of their use in the computer program, and the familiarity of developers with XML Schema Definition. Certainly, training on the proper use of XSD-related Zhi# programming language features will be required in order to avoid, for example, excessive usage of cast expressions, which would counteract static type checking and type inference.

In order to test hypothesis 2.2, several different paradigmatic cases concerning the use of ontological description languages might be considered. First, there are applications that almost solely depend on TBox reasoning such as classifying concept descriptions in a hierarchy. A well-known example of such kinds of applications is University of Manchester's exemplary PizzaFinder application [Hor04]. The Zhi# programming language will

support such scenarios mainly by facilitating the syntactical import of concept and role names into program code while there is almost no use of ontological individuals.

Particularly problematic may be use cases where the TBox is not assumed immutable as in, for example, the OWL2XMI software [OWL09], which generates XMI files from OWL ontologies. Even dynamic checking in Zhi# programs may be rendered useless if concept descriptions are entirely contingent.

The biggest benefit may be provided for the development of knowledge-based applications that are grounded on an immutable TBox with a continuously changing ABox. This is exactly the case for, for example, the prototypical computer services that were showcased at the Second CHIL Technology Transfer Day [Inf04]. The Zhi# solution is likely to be particularly useful for terminologies that include a concrete domain, which is the case for CHIL demo applications that make heavy use of OWL datatype properties to relate ontological individuals with XML data type values. In fact, the then insufficient integration of ontological concept descriptions and XML data types with widely used programming languages triggered the development of the Zhi# solution in course of the CHIL research project.

Hypothesis 2.2 may also be tested in scenarios where OWL concept constructors and role restrictions need to be (re-)implemented in different languages such as C#. Based on the microscopic evaluations in Subsections 7.2.3 and 7.2.4 it is conceivable that – assuming adequate training on the use of ontologies in Zhi# programs – the combined use of OWL and the Zhi# programming language to denote descriptions languages and write computer programs, respectively, is superior to the mere use of ontology-agnostic programming languages. The same is likely to hold for the use of OWL concepts and roles in Zhi# programs compared to the explicit use of OWL APIs as outlined in Subsection 7.2.5. After all, the Zhi# approach coheres with a fundamental and well-proven design principle of programming languages; the Zhi# approach only *adds* levels of abstraction while programmers may still use conventional means such as OWL APIs wherever appropriate.

### 7.3.1 OCL invariants in Zhi#

The Object Constraint Language (OCL) [Obj06] has been included with the specification of the Unified Modeling Language (UML) [Obj09a, Obj09b] since UML version 1.1. OCL provides means for textually specifying constraints, which apply to model elements, that cannot otherwise be expressed by UML's diagrammatic notations. On the other hand, OCL constraints always refer to a UML diagram. For example, class definitions of a UML class diagram complement the set of built-in OCL types. In fact, OCL is most frequently used to adorn UML class diagrams. OCL constraints can be attached to every UML model element. The OCL constraint set is subdivided into six broad categories, where each category of constraints is expressed using a similar syntax.

*Invariants* state that a condition must always be met by, for example, all instances of a class. *Pre-* and *postconditions* are restrictions that must be true before an operation is executed and after an operation has ended, respectively. Note that the OCL specification remains silent about what should happen if the precondition did not hold of the state before the execution of an operation (i.e. the caller is obliged to ensure that given preconditions hold). *Initial* and *derived values* specify the initial values of, for example, attributes and derivation rules for, for example, attributes, respectively. *Definitions* provide for the declaration of supplemental attributes and operations that are not part of the underlying UML model. *Body definitions* specify the bodies of query operations (i.e. operations that are assumed to terminate on every input). *Guards* are constraints that must be true before a state transition happens.

A very frequent constraint/model element combination is the use of OCL invariants to tag UML class attributes. Invariants are described using an expression that evaluates to true if the invariant is met. OCL constraints are always defined in a particular context that specifies the model element for which the OCL expression is defined. Invariants must be defined in a *classifier context*, where the augmented model element usually is a class or interface type. Invariants in a classifier context are denoted in the following form.

**context** [c:] *Type*

**inv** [e:] *expression*

OCL keywords are set in boldface. *Type* is the name of the contextual type of the given OCL *expression*, which has to be of type *Boolean*. OCL expressions are evaluated for instances of the contextual type. The optional parameters *c* and *e* can be used to define a variable of the given type and the given OCL expression, respectively. Alternatively to type variable *c*, the keyword *self* can be used to refer to the *contextual instance* (i.e. the instance for which the given expression is evaluated).

OCL is a typed language and allows for the use of five categories of types. *Model types* and *enumeration types* stem from user-defined type definitions in the underlying UML diagram. *Basic types* are OCL's built-in atomic types *Integer*, *Real*, *String*, and *Boolean*. The only generalization relation stipulated for basic types is *Integer* <: *Real*. Model and basic types can be aggregated in OCL *collection types* (e.g., *Set(T)*). Note that there are no collections of collection types. The *special types* category comprises, among others, the supertype of all non-collection types *OclAny*.

All types are simply denoted by their names. The OCL specification defines a number of arithmetic and logical operations both on basic as well as collection types. For example, numerical types can be added and compared for equality (see [Obj06] for the operations and well-formedness rules of OCL types). OCL expressions can be built-up from typed model elements and operations on them. OCL constraints apply OCL expressions in particular contexts for different purposes. Again, as indicated by the syntax description above, OCL language statements comprise 1) a context, which defines the scope to which a statement pertains, 2) a property that represents some characteristics of the context (e.g., a class attribute), 3) a well-formed OCL expression, and 4) possibly additional keywords such as *if*, *then*, *else* to denote conditional expressions.

In this work, only the frequently occurring case of unconditional constraints on class and instance attributes will be considered (i.e. invariant definitions in a classifier context).

### 7.3.1.1 Exemplary OCL invariants

Ponder the following UML class *Person* with attributes *name*, *age*, and *eMail*. The value space of the OCL *Integer* type comprises the set of integer numbers; the OCL *String* type represents character chains of arbitrary length (i.e. both value spaces are unbounded). In practice, programming techniques such as *lazy evaluation* and *thunks* are required to construct infinite data structures. For the given *Person* class it is, however, a good idea anyhow to further constrain the set of admissible attribute values. Hence, the following UML model may be implemented in Java as shown below.

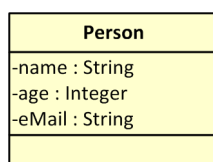


Figure 7.5: Context class for attribute invariants

```
1 class Person {
2     String name;
3     int age;
4     String eMail;
5 }
```

The Java *String* API supports up to  $2^{31} - 1$  (i.e. *Integer.MAX\_VALUE*) characters. The Java *Integer* type implements signed 32-bit integer numbers. While the use of both Java data types does not impose any implementation issues anymore, it makes sense to even further restrict possible *name*, *age*, and *eMail* values of a *Person*. OCL constraints come to the rescue. The following invariant definitions restrict the age of a person to values greater than or equal to zero and less than 110; also, persons can have only names that are not longer than 40 characters.

**context** *Person*

**inv** *age*  $\geq 0$  && *age*  $< 110$

**context** *Person*

**inv** *name.size()*  $\leq 40$

At this point, it is important to understand that UML and OCL-based models only constitute a specification of the system that one wants to build. Especially, there are no prescriptions *how* to *implement* model elements and OCL expressions. In fact, there are a number of approaches how, for example, given OCL invariants can be enforced.

Stirewalt and Rugaber [SR05] employ C++ metaprogramming features. Specified OCL invariants are dynamically enforced by component wrappers, which are implemented as nested C++ template class instantiations. An alternative approach is provided by the Open C++ project [Chi98]. The Open C++ meta-object protocol might be used to reprogram the compiler to guard modifications of class attributes. The GenVoca [BST94] tool, which is at a fundamental level similar to aspect-oriented programming, provides for the definition of higher level code constructs that can be used to mix program code with high-level design features such as invariants. Eventually, at the time of this writing, one of the most complete OCL compilers is the Dresden OCL Toolkit [LO04], where OCL constraints are translated into AspectJ-based constraint code.

### **7.3.1.2 Implementation of OCL invariants with the Dresden OCL Toolkit and AspectJ**

AspectJ [Asp08] is an aspect-oriented extension of the Java programming language. It uses Java-like syntax (i.e. every valid Java program is a valid AspectJ program) to define special classes called *aspects*. In aspect-oriented programming (AOP) [KLM97] aspects describe additional features or behaviors that are orthogonal to a number of core-level concerns of a program. Aspects support the encapsulation of concerns into separate, independent entities. Thus, AOP attempts to facilitate the composition of systems along a number of dimensions (in OOP, there is only one single such dimension). Additional behavior (called *advice*) that one wants to add to an existing program may, for example, be the checking of the values of instance variables in order to implement specified OCL invariants. Such checking may occur at certain points in the execution of a program.

In AOP, a point in the control flow where the main program and an aspect meet is called *join point* (i.e. a join point exists there where Hoare logic places an assertion). A *pointcut* is an expression that determines whether a given join point matches. While an updated runtime environment could be conceived that understands additional AOP features, most AOP implementations such as AspectJ produce combination programs that are indistinguishable from non-AOP programs through a process called *weaving*. Early AspectJ implementations used source code-level weaving while nowadays class files can be combined, which frees developers from providing source code for all AOP-affected entities.

The modification of relevant program code that is described in pointcuts is accomplished automatically by AspectJ. It would, however, still be particularly tedious to manually devise join points and advices in order to implement given OCL constraints. Again, tool support is available in form of, for example, the Dresden OCL Toolkit [LO04]. The Dresden OCL Toolkit is a modular toolset comprising libraries and stand alone tools that can be used to parse and type-check OCL constraints and to facilitate the instrumentation of Java programs for runtime verification of input OCL constraints. In particular, the Dresden OCL Toolkit can be used to automatically translate OCL constraints into AspectJ pointcuts and advices.

In order to do so, MOF models (i.e. UML class definitions) can be imported from XMI files<sup>8</sup>. The Dresden OCL2 Toolkit for Eclipse features special tree views for browsing the class definitions of a UML diagram. UML class diagrams can then be complemented by constraint definitions from separate OCL files. The combined set of UML classes and OCL constraints is taken to generate AspectJ code that implements the specified OCL constraints for the given UML-derived Java classes. The Dresden OCL2 Toolkit will reports errors if constraints are not applicable in the given context (e.g., because of unavailable type names). Eventually, a combined Java program can be compiled from user-defined Java source code and automatically generated AspectJ aspects.

---

<sup>8</sup>Meta Object Facility (MOF) is a metadata architecture standardized by the Object Management Group (OMG). XML Metadata Interchange (XMI) is a supporting standard of MOF for exchanging metadata information in an XML-based format.



The Dresden OCL Toolkit translates the OCL invariant introduced above, which restricts *age* values of *Person* objects, into the following AspectJ code (generated comments are omitted and fully qualified type names are abbreviated for the sake of conciseness). The invariant pertaining to *name* values of *Persons* would be translated likewise.

```

1 @Generated public privileged aspect InvAspect1 {
2   protected pointcut allPersonConstructors(Person p):
3     execution(Person.new(..) && this(p));
4   protected pointcut ageSetter(Person p) :
5     set(* Person.age) && this(p);
6   protected pointcut allSetters(Person p) :
7     ageSetter(p);
8   after(Person p) : allPersonConstructors(p) allSetters(p) {
9     if (!(p.age >= new Integer(0)) && (p.age < new Integer(110)))) {
10      throw new RuntimeException("Error: Constraint was violated.");
11    } } }

```

The *Generated* annotation marks Java source code that has been generated. The *privileged* AspectJ aspect *InvAspect1* is granted access to private class members. At the execution join points defined by the first pointcut a program is executing constructor calls of type *Person*. In the second pointcut, the *set* designator is used to describe all join points based on assignments to the *age* attribute in class *Person* in the primary code. The third pointcut collects the *age* attribute setters. The defined *after* advice executes after the execution of the given join points has completed (either with or without throwing an exception). The (configurable) advice implementation facilitates a runtime check that set values of the *age* attribute are within the borders defined by the initial OCL invariant.

Note that the second pointcut in the above listing covers *all* assignments to the *age* attribute (and not only the execution of dedicated setter methods). The defined advice is, however, only effective for instrumented class files that were available at compile time. Assignments made in additional client code are not (i.e. cannot) be checked.

### 7.3.1.3 Implementation of OCL-like invariants in Zhi#

The Dresden OCL Toolkit and AspectJ-based implementation of OCL constraints accomplishes a completely automatic and type-checked translation of given OCL constraints into a combined Java program. The obvious shortcomings of the presented approach are, however, that 1) four kinds of different input data are necessary (i.e. XMI, OCL, AspectJ, and Java files), 2) a considerable tool suite and explicit user interaction are required, 3) only runtime type checks are facilitated, and 4) certain pointcuts are woven into client code, which must be available at compile time. Also, constraints can only be formulated using OCL's limited type system. There is no means to state, for example, that a *Person* object can only have values for its *eMail* attribute that match a given regular expression. The value space-based subtyping rules of the  $\lambda_C$ -calculus type system will certainly not suffice to implement the complete AspectJ join point model. They may, however, be a valid alternative for implementing OCL-like invariants.

Assume there was an OCL plug-in for the Zhi# compiler framework (similarly to the XSD plug-in) that derives type definitions from OCL invariants on C#'s intrinsic data types. For example, a type  $int\{>= 0\}\{< 110\}$  (cf. Subsection 5.1.1) could be made available to declare the *age* attribute of type *Person*. Hence, the initial OCL invariant specifications could be substituted by context-free type definitions; the weaving of primary code and constraint definitions would be replaced by Zhi#'s source-to-source compilation (i.e. several AspectJ dimensions would be coalesced into the core program code). In contrast to the Dresden OCL Toolkit, programmers would no longer be obliged to provide MOF models and separate OCL constraint definitions, which entail the use of a considerable tool suite. Also, in contrast to aspect-oriented approaches to implement OCL-like invariants, Zhi#'s one single dimension of program code can be fully statically type-checked. The initial *Person* class and OCL invariant definitions could be rewritten as shown below to make use of XSD-like constrained data types. The OCL type system evidence in the *import* directive indicates the use of OCL derived type definitions.

```

1 import OCL inv = namespace name ;
2 class Person {
3     #inv#Name name;    // Name ≡ String{?< 40}
4     #inv#Age age;     // Age ≡ int{>= 0}{< 110}
5     #inv#EMail eMail; // EMail ≡ String{?? 'regular expression for eMail addresses'}
6 }

```

Static type checks in a Zhi#program would occur at all places in a program that match AspectJ pointcut definitions (i.e. the weaving of primary code and OCL-derived assertions is replaced by compile-time program analysis). Accordingly, constraint violations at runtime may be completely avoided. Also, the Zhi# approach would remove an irritating shortcoming of AspectJ when pointcuts are directly defined for assignments to class attributes. AspectJ advices that pertain to assignments to class attributes are interwoven with the client code and not with the restricted object. Consequently, OCL invariants can only be enforced for client code that is readily available at compile time while nothing can be done about conventional Java client code that is contributed later on. In Zhi#, runtime type checks are facilitated by the invariant object itself. Hence, Zhi# components can be used by arbitrary .NET assemblies, which lack Zhi#'s static type checking of constrained types, with invariant declarations still being effective.

Implementing an OCL plug-in for the Zhi# compiler framework appears feasible due to the similarity of OCL invariant definitions to the XSD type system. OCL's four basic types correspond to XSD's built-in *xs#integer*, *xs#decimal*, *xs#boolean*, and *xs#string* types. The OCL subtyping relationship *Integer* <: *Real* is matched in the existing XSD compiler plug-in by, for example, implemented covariant coercions of *xs#integer* data types to floating point numbers. Furthermore, an OCL plug-in for the Zhi# compiler framework may offer supplemental operators on basic OCL types. For example, plain OCL lacks means to restrict *eMail* values of the exemplary *Person* class to strings that match a given regular expression.



## CHAPTER 8

### Conclusion and Outlook

#### 8.1 Conclusion

The author embedded the constraint-based subtyping of XML Schema Definition (XSD) and ontological reasoning of the Web Ontology Language (OWL DL) with the C# programming language. In the resulting programming language Zhi#, XML data types and ontological concept descriptions are first class citizens and can be used along with common features of an object-oriented programming language such as user-defined operators and method overriding. The Zhi# language definition is a proper superset of ECMA standard C# 1.0 with minor syntactical modifications to allow for the import of external namespaces and external type references. Evaluation results indicate the usefulness to have in a program the same convenient and compact notations used to describe data in an ontology.

The Zhi# compiler framework facilitates the integration of external typing and subtyping mechanisms such as, for example, type inference, subsumption and type derivation with the C# programming language. In particular, the compiler framework supports the notion of constrained atomic data types and type inference based on control and data flow analysis. The compiler framework provides the core functionality to type check and transform conventional C# programs. Compiler plug-ins provide support for external type systems. External type definitions can be used in Zhi# programs in all places where .NET types are admissible except for type declarations. The Zhi# compiler facilitates the cooperative usage of type definitions from different type systems. For example, XML data

types can be used along with OWL datatype properties. Types of different type systems can cooperatively be used in one single statement. The compiler framework allows for the accumulation of type inference results from several different compiler plug-ins. Cooperation between external type systems and between external type systems and the .NET type system is achieved by delegating type checking and program transformation tasks to the external compiler plug-ins that are responsible for the particular external parts of an expression. The compiler framework selects the compiler plug-ins based on the classification of types into type systems. The architectural model of the Zhi# compiler framework was designed for easy extensibility. Most of the program analysis tasks that require knowledge about the possibly complex code structure of C# programs were factored out of the framework extension points. Compiler plug-ins must implement extension points for (sub-)typing and program transformation. Both extension points are for the most part context free (i.e. the analysis and transformation of external expressions does not depend on the position of those expressions in the program text). Thus, it is possible to implement plug-ins for the Zhi# compiler framework without exhaustive knowledge about the C# programming language itself. At the same time, no a priori knowledge about the supported external type systems of prospective compiler plug-ins is required in the compiler framework. Zhi# programs are compiled into conventional C# code, which is correct by generation. All external type system functionality is used in a “pay as you go manner”. There is no performance or code size overhead for Zhi# programs that do not use external type definitions. In contrast to naïve approaches that are based on wrapper classes or additional code generation, the program overhead of compiled Zhi# programs is constant and does not grow with, for example, the number of imported external types. Up to now, two compiler plug-ins were implemented for XML Schema Definition and the Web Ontology Language. The presented approach is not limited to the C# programming language but can equally be applied to any statically typed object-oriented programming language (e.g., Java). External type systems are not limited to XML Schema Definition’s value space-based subtyping or ontological reasoning with the Web Ontology Language.

In contrast to radical Curry-style language specifications, the Zhi# programming language is given in the Church-style where typing is prior to semantics. Accordingly, the use of external types requires the availability of corresponding compiler plug-ins. Hence, in contrast to, for example, Gilad Bracca's interpretation of pluggable type systems, Zhi# provides for widely used programming techniques such as static type-based overloading and class-based encapsulation. With completely optional type systems class-based encapsulation would require prohibitively expensive dynamic checks (this is why in programming languages such as Self and Smalltalk, object-based encapsulation is used instead).

The Zhi# compiler framework is complementary to two broad categories of approaches to facilitate the *augmentation* of programming language type systems and to add *syntactic* extensibility to programming languages. Frameworks for augmenting programming language type systems require a complete understanding of the language grammar (i.e. the evaluation rules) while only limited knowledge of the Zhi# grammar is needed to develop type system plug-ins for the Zhi# compiler framework. Most systems that facilitate syntactic programming language extensions only support syntactic safety, leaving out full-fledged type checking of devised language extensions. In contrast, compiler plug-ins for the Zhi# compiler framework can be implemented to provide for type system-specific static typing, dynamic checking, and control and dataflow-based type inference.

XML Schema Definition – the W3C Recommendation for XML data types – introduces atomic data types that can only have atomic values, which are not allowed to be further fractionalized. A primitive atomic data type is a three-tuple consisting of a set of distinct values called its value space, a set of lexical representations called its lexical space, and a set of fundamental facets. The value space is the set of values for a given data type. Each value in a value space is denoted by at least one literal of its lexical space. Fundamental facets semantically characterize abstract properties of the value space such as, for example, order, cardinality, and boundedness. Constraining facets are optional properties that can be applied to a data type to restrict its value space (i.e. value space-based subtyping).

The author devised an extension of the simply typed lambda calculus with subtyping ( $\lambda_{<}$ ) for constrained atomic data types. The  $\lambda_C$ -calculus defines three subsumption rules for constraints based on their application on atomic data types. A constraint is a sub-constraint of another constraint if the value space that results from the application of the former constraint on a data type is a subset of the value space that results from the application of the latter constraint on the same data type. Constraints can be more restrictive based on the widths and depths of their definitions. In the type system of the  $\lambda_C$ -calculus data types are inductively defined by their value spaces. Starting from a set of built-in primitive types with axiomatically defined value spaces, atomic types are derived through the application of value space constraints. Constraint applications on atomic types can be reduced to set operations on the types' value spaces. An atomic type is a subtype of another type if the value space of the former is a subset of the value space of the latter. The type safety of the developed  $\lambda_C$ -type system was proved based on the soundness proof of the simply typed lambda calculus with subtyping ( $\lambda_{<}$ ). The  $\lambda_C$ -calculus allows for the inference of transient constraints that may hold for the instances of constrained types within only a limited scope of a program. Type inference rules were defined that capture the two intuitions of adding constraints to the type of a variable for a limited scope and eventually removing them upon modifications of the variable.

The form of type construction in the  $\lambda_C$ -calculus can be used to mimic type construction and derivation as in XML Schema Definition. The  $\lambda_C$ -type system was fully implemented in Java and C# for the XML Schema Definition type system. These implementations can be used to load type definitions from XSD files and classify these types in a hierarchy. Binary expression types that involve constrained types can be inferred using an extensive interval arithmetic-based constraint arithmetic. The object-oriented architectural model of the  $\lambda_C$ -implementation makes it possible to join an arbitrary number of particular type system implementations in one single type pool. It is conceivable to implement further type systems that are similar to XML Schema Definition such as the Structured Query Language (SQL) or Object Constraint Language (OCL) type systems.



The XSD implementation of the  $\lambda_C$ -type system stipulates formal subtyping rules, which are left out in tools such as CQUAL, which can be used to extend standard programming language types with flow-sensitive type qualifiers. Explicit subtype relations are unsupported, too, in the set of “semantic type qualifier” approaches, which can, similarly to constrained data types in the  $\lambda_C$ -type system, be used to express constraints of interest. Also, type qualifiers are usually defined for particular code patterns, while no knowledge of the evaluation rules is necessary for defining constrained data types in the  $\lambda_C$ -type system.

The implementation of the  $\lambda_C$ -type system lays out the foundation of the Zhi# compiler plug-in for XML Schema Definition. The XSD compiler plug-in facilitates loading of XML schema definitions, static typing of constrained atomic data types in Zhi# programs, and the transformation of XSD type references in Zhi# programs into C# code. At compile time, the use of XML data types is checked according to the constraint-based typing rules of the  $\lambda_C$ -calculus. The Zhi# compiler is able to detect exactly which constraints are violated by an assignment instead of merely reporting generic type incompatibilities. XML data types can be used in any context of Zhi# programs that is valid for built-in .NET value types. In particular, XML Schema Definition type system rules were integrated with programming language features such as method overriding, user defined operators, and runtime type checks (i.e. *XSD aware compilation*). The XSD compiler plug-in takes into account both nominal and structural aspects of the XML Schema Definition type system. Implicit conversions are provided between primitive .NET value types and XML data types for which a value space-based subtype relation exists. The XSD compiler plug-in enforces invariant subtyping for XSD array types, which is a deviation from the covariant subtyping of .NET array types, and allows covariant subtyping for non-array types. The Zhi# compiler plug-in for XML Schema Definition infers types of variables based on control and data flow analysis. The type inference rule for *if*-statements in the  $\lambda_C$ -calculus was complemented with type inference rules for *for* and *while*-statements.

Types of literals are inferred based on the literal expressions' particular values. Types of binary arithmetic expressions are inferred based on the constraint arithmetic of the  $\lambda_C$ -type system. The generated C# output code contains dynamic type checks in order to allow for schema modifications after the compilation of a Zhi# program and a safe usage of Zhi# assemblies from conventional .NET programs. In this way, Zhi# programs are fully interoperable with standard .NET assemblies with full consideration of the stipulated XML Schema Definition value space constraints.

The Zhi# approach is unique in making constrained XML data types first class citizens of a general-purpose object-oriented programming language. Existing approaches focus either on content models of complex types, or on embedding programming language code in XML documents or inserting XML literals into programming language source code.

Ontological data reside in knowledge bases that are subject to ontological reasoning. The author devised a pluggable architectural model of an ontological knowledge base server. The CHIL Knowledge Base Server can be used to adapt off-the-shelf ontology management systems for the Web Ontology Language (OWL DL). In the current Java-based implementation, the Jena Semantic Web Framework is adapted and configured to use the Pellet OWL DL reasoner with in-memory and database backed ontology models. The CHIL Knowledge Base Server implements the formally specified CHIL OWL API, which was defined based on a combination of Floyd-Hoare logic and formal Description Logics terminology. Using Floyd-Hoare logic rules, the formal API specification can be used to reason on the meta-level about the effects of sequences of operations on OWL DL knowledge bases. Regression test code was automatically generated from the formal specification. Complementary to the formal specification, the CHIL OWL API definition is given as an XML instance document based on a twofold XML Schema Definition that was used to automatically generate client components for programming languages such as Java and C#. The CHIL OWL API comprises 91 formally specified methods for telling and asking the TBox and ABox of OWL DL knowledge bases plus 32 auxiliary methods.

Its formal specification sets the CHIL OWL API apart from existing DL knowledge base interface specifications. Moreover, in contrast to existing ontology management systems, the CHIL Knowledge Base Server reveals excellent connectivity capabilities by supporting the concurrent use of a number of different remoting protocols.

The CHIL OWL API is used by the OWL DL plug-in for the Zhi# compiler framework. Zhi#'s OWL DL compiler plug-in makes the property centric modeling features of the Web Ontology Language available via C#'s object-oriented notation and integrates ontological reasoning with features of the programming language (i.e. *OWL aware compilation*). Ontological concept and role descriptions are subject to a combination of static typing and dynamic checking. Referenced ontological concepts and roles are checked at compile time to exist in the imported ontology. Static type checks include disjoint concepts, and cardinalities and disjoint property domain and range restrictions of ontological roles. OWL concept descriptions are permitted in Zhi# programs in all places where .NET types are admissible except for type declarations. In particular, OWL concepts can be used for formal parameters of methods, user defined operators, and indexers. The power of the “.” can be used to declare ad hoc relationships between ontological individuals and to declare members on a per instance basis. Property domain and range restrictions can be used for ontological reasoning. The *checked*-operator enforces frame-like semantics of OWL concepts and roles in Zhi# programs. Auxiliary properties were implemented for ontological individuals, roles, and static concept references in Zhi# programs in order to make it particularly easy to, for example, get all individuals in the extension of a specified concept description or to get all RDF types of a specified individual. A mapping of CHIL OWL API methods to auxiliary properties of ontology elements in Zhi# programs was presented that makes half of the defined preconditions of the CHIL OWL API amenable to static checking. The RDF types of ontological individuals in Zhi# programs are dynamically checked at runtime. The *is*-operator can be used for dynamic RDF type checks of ontological individuals. The OWL DL compiler plug-in can be used cooperatively with

the XSD compiler plug-in in order to support OWL datatype properties. Zhi# code that uses elements of an ontology is compiled into conventional C#. In the current implementation, the OWL DL component of the Zhi# runtime library utilizes the CHIL OWL API in order to manage the ontological knowledge base. Eventually, the OWL DL plug-in reduces the dependency on particular OWL APIs since it is possible to substitute the CHIL OWL API in the Zhi# runtime library without recompilation of Zhi# programs.

The Zhi# solution to provide programming language inherent support for ontologies is the first of its kind. Earlier attempts either lack ABox reasoning, concurrent access to a shared ontological knowledge base, or fall short in fully supporting OWL DL's modeling features. Previous and current approaches to embed a variety of dynamic typing schemes in statically typed programming languages include the concept of a *Dynamic* type contrived by Abadi et al. [ACP89, ACP91], *quasi-static* type systems developed by Thatte [Tha90], *gradual typing* [ST06], and Tobin-Hochstadt and Felleisen's *occurrence typing* [TF08]. These approaches – though well defined and carefully evaluated – are not tailored to an ontology language such as the Web Ontology Language (OWL DL).

The Zhi# tool suite includes the Zhi# compiler, which was integrated with the MSBuild build system for Microsoft Visual Studio by means of an MSBuild task component. The Eclipse-based frontend boasts a Zhi# editor with syntax highlighting and autocompletion.

The current state of ontological reasoning technology may not yet be sufficient to manage very large ontological knowledge bases. However, significant performance gains have been achieved in recent years. These developments are likely to continue since the advantages of the convenient and compact OWL notation have been widely acknowledged. As a consequence, it appears to be reasonable to make this important form of ontologies available in a widely used programming language. Microscopic examinations give evidence for the usefulness and applicability of the proposed compiler framework. Its extensibility and universal practicability are indicated by the suggested implementation of OCL invariants.

## 8.2 Outlook

Up to now, the Zhi# language extensions to C# are only visible on the class level. On the whole, external type information is lost when Zhi# programs are compiled into plain C# code. It is planned to extend the scope of the Zhi# language extensions to the component level by adding annotations to the generated C# code, which will make it possible to reconstruct external type information from compiled Zhi# assemblies. Thus, it would be possible to not only reference conventional .NET assemblies but also Zhi# libraries with the original extent of typing information. The C# programming language provides for user-defined declarative information in form of attributes. Attribute classes may be used to describe Zhi# type information in the output C# code.

Approaches that allow for syntactic extensions of programming languages usually feature a debugger that is aware of the programming language extensions. Similarly, it is planned to augment the Zhi# tool suite with an extensible debugger that is aware of the original Zhi# source code and of the implementation of operations on external objects (e.g., assignments to ontological roles). In general, modifications of, for example, a shared ontological knowledge base cannot be assumed to be atomic since the underlying infrastructure cannot be completely made transparent. Still, it appears reasonable to facilitate stepping through a Zhi# program on the Zhi# language level for all cases where operations do not fail due to, for example, networking or ontology management errors.

The long term technical vision for the Zhi# approach is to become an inherent feature of an augmented .NET runtime. Instead of piggybacking on conventional C#, constrained atomic data types and ontological reasoning rules may be integrated with the MSIL type system. Under the hood, the most recent version 3.0 of the C# programming language already utilizes features of structural type systems (e.g., internal class definitions are reused for identical structural types of LINQ expressions). It would be a natural amalgamation to further develop structural typing techniques with value space constraints and ontological reasoning rules.

Future work will include the transformation of OMG Ontology Definition Metamodels [Obj05b] into Zhi# programs. Currently, on the meta-level, ontologies and UML class models can be related to each other in one unified model, which even includes a certain degree of MOF-based static data type analysis. Still, once the ontological and non-ontological parts of the unified model are transformed into an ontology markup and a programming language, respectively, the unified model again falls apart into two distinct class hierarchies. With ontological class descriptions being first-class citizens the complete MOF [Obj05a] modeling space can be translated into the Zhi# programming language.

The Zhi# compiler infrastructure has shown to be a viable approach to solving the OWL-OO integration problem. It is conceivable to implement additional plug-ins for different logics such as, for example, default logic. Combined keywords such as, for example, *possibly new* and *possibly not* may provide for elements of modality in modal logics.

In Zhi#, greater dynamicity could be achieved for OWL object properties. The other way round, dynamically typed programming languages might be extended with external type systems to enforce frame-like semantics of ontologies to achieve greater rigor.

In the long run, the author plans to investigate the interplay of closed world semantics in an ontology with autoepistemic features (e.g., the epistemological  $\mathcal{K}$ -operator) with the type checking provided in the Zhi# programming language in order to enable a higher degree of static type checking for ontological concept descriptions. Also, the recent integration of XML data types with ontological concept descriptions in the draft of OWL 2 may be considered by Zhi#'s compiler plug-ins for XSD and OWL DL. Eventually, safe ontological reasoning rules may be embedded with Zhi# programs.

The author believes that in the near future, programming language compilers will become increasingly configurable and programmable in order to provide for a more and more dynamic world. This includes support for dynamic checking of objects whose structure is unknown at compile time and support for content models with different conceptual bases than the built-in type systems of general-purpose programming languages.

In fact, in many cases, the structure of data can only be assumed unknown at compile time. For example, content models of XML documents such as XHTML Web pages for which no schema exist can only be discovered at runtime. The same goes for comma separated files, spreadsheets, and contingent ABoxes of Description Logic knowledge bases. Moreover, data may need to be processed that could (at least partially) be statically typed but whose definition languages are conceptually different from the static type system of the programming language in use. Eventually, combinations of static typing and dynamic checking may be appropriate in scenarios that are similar to ontological knowledge bases.

As a consequence, the time-honored workflow to apply a black box compiler to input source files in order to get output object files will be complemented by type system and application specific compiler programming and configuration tasks. Most notably, the up to now entirely statically typed C# programming language will include in its next version the new *dynamic* keyword, which can be used to declare objects whose member lookups and method calls will be resolved at runtime (by the Dynamic Language Runtime). For *dynamic* variables the C# compiler will generate code that will allow dynamic lookups and invocations whose actual meanings are deferred until runtime and will depend on the semantics of the *IDynamicObject* implementation. The *IDynamicObject* interface can be implemented to provide type system specific member lookup and method invocation.

This work showed that it is possible to not conceal different type systems behind the façade of a common *dynamic* type. Instead, external type systems and their different typing, subtyping, and type inference facets can be made visible in widely used general-purpose programming languages. Thus, the rigor of a number of static type systems can be combined to achieve a higher degree of flexibility (e.g., the .NET type system plus the XSD type system in Zhi#). Combinations of static typing and dynamic checking may be used when the structure of data is only partially known (e.g., the OWL type system in Zhi#). Dynamic typing can be used when content models are completely unknown at compile time (e.g., the type *dynamic* in C# 4.0).





## APPENDIX A

### The Zhi# Compiler Application

The Zhi# compiler was implemented in C# 3.0 as a .NET Framework 3.5 assembly, which can be used locally from the command line and be invoked as an MSBuild [Mic07d] task. Also, the compiler assembly can be hosted by a Windows Forms-based server application (see Fig. A.1), which exposes the API of the Zhi# compiler via an XML-over-TCP interface. In this way, the Zhi# compiler is available locally and remotely to .NET and non-.NET clients. The Zhi# server application itself provides various visualizations of the recent compiler run. For example, the abstract syntax tree of input Zhi# code is displayed, compiled Zhi# code is displayed in a tree view that shows type information of source code elements when hovering with the mouse pointer over these elements, one can browse the tree view presentation of the constructed type table, and XML data types and generated anonymous OWL concepts are displayed in a list view.

The actual Zhi# developer frontend was implemented as an Eclipse 3.5 plug-in as shown in Fig. A.2. A Zhi# Eclipse project type was defined that supports an MSBuild-based build process. The Zhi# editor boasts syntax highlighting and autocompletion of XML data types and OWL concept and role names. An error list view can be used to navigate to erroneous lines of code. Zhi# source files (.zs), XML schema files (.xsd) and files that contain OWL ontologies (.rdf, .owl) are managed by the Eclipse project environment and are passed into the Zhi# server via its XML-over-TCP interface. The outline view of Zhi# source files displays XML data type and OWL concept information in a tree view. Metadata about XML data types, OWL concepts and roles are displayed as tooltips when hovering with the mouse pointer over these elements in Zhi# programs.

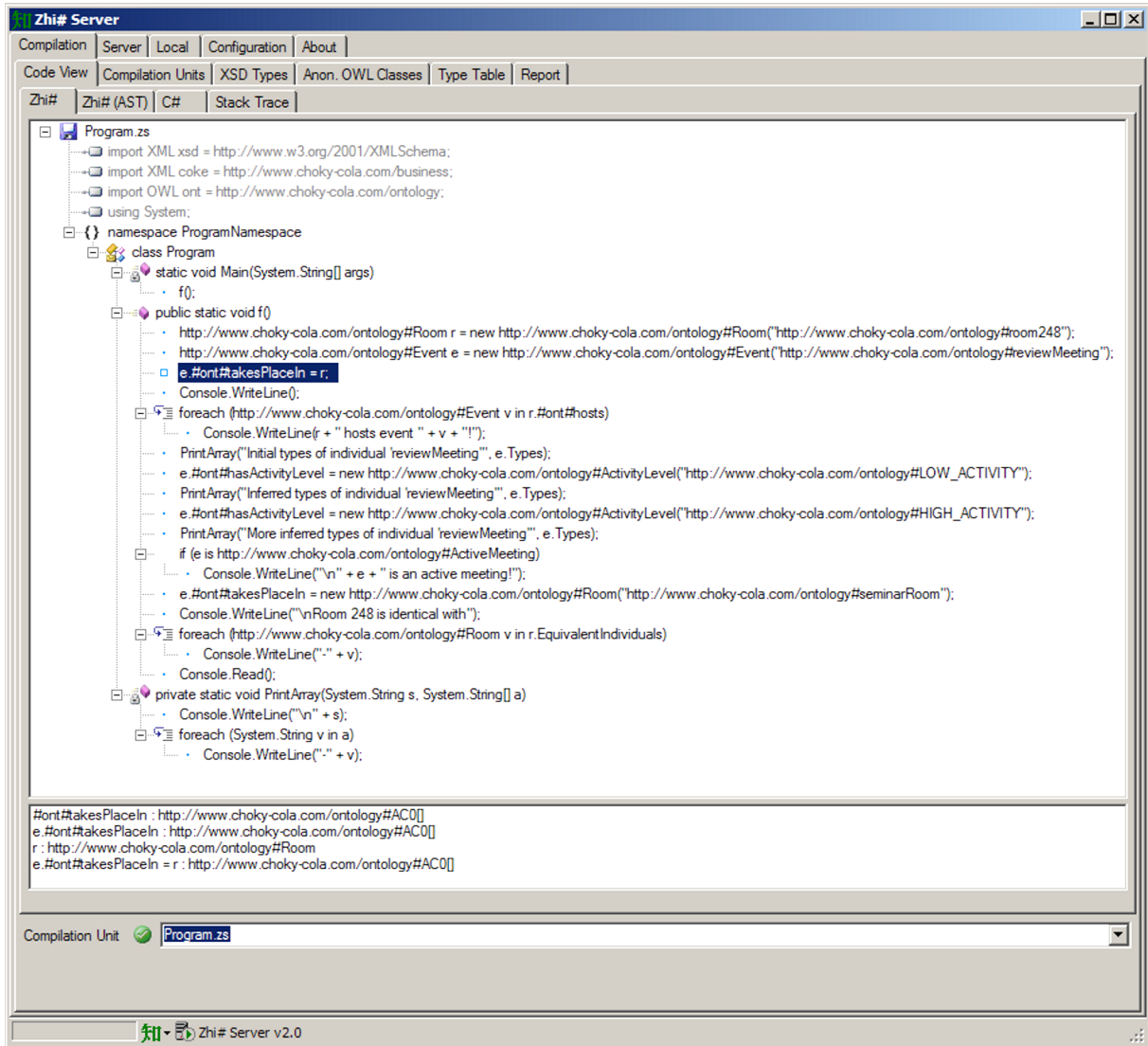


Figure A.1: Zhi# server application

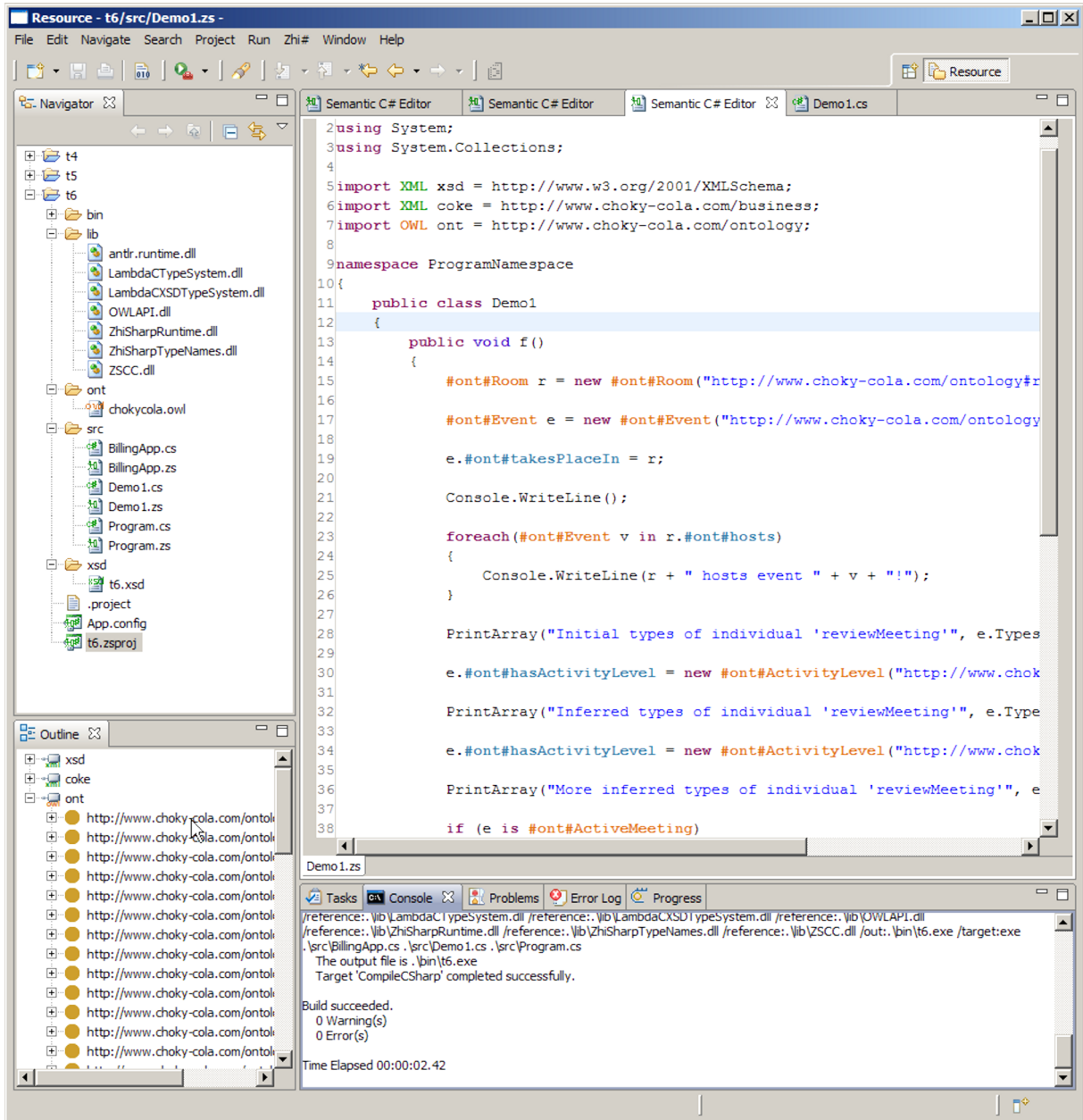


Figure A.2: Zhi# Eclipse-based frontend

## APPENDIX B

### Zhi# External Program Transformation Functions

This appendix elucidates the implementations of the external program transformation functions that are provided by the Zhi# compiler plug-ins for XML Schema Definition and the Web Ontology Language. The program transformation functions were specified in the premise-conclusion form as introduced in Section 2.3.

In the following arguments, monospace font indicates generated C# output code (e.g., “ $\mathbf{x} = \mathbf{v}$ ”). For the sake of conciseness, in the specifications of the program transformation functions the following three operator definitions are used to specify different cases. The binary *is*-operator performs a type-check of its operands. It yields true if the left operand is an instance of the right operand; otherwise false. The binary casts-exists operator  $v \triangleright_{\Gamma} T$  determines whether in the typing context  $\Gamma$  the specified object  $v$  can be casted to the specified type  $T$ . The binary implicit-casts-exists operator  $v \triangleright_{\Gamma}^{impl} T$  determines whether in the typing context  $\Gamma$  an implicit cast exists from  $v$  to the specified type  $T$ .

Zhi# source code is processed in the form of code DOM objects. In the following specifications, the function  $o(t)$  yields the code DOM object that represents the term  $t$  in the current context. The “dot”-operator can be used to access fields of code DOM objects. For example, code DOM objects expose a field *ZhiSharpType*, which corresponds with the type of the represented Zhi# term (e.g.,  $t : T \Rightarrow o(t).ZhiSharpType = T$ ).

Field updates of code DOM objects are given in square brackets. For example, given an object creation expression  $t = \text{“new byte}(3)\text{”}$  the field update  $o(t)[ZhiSharpType = int]$  changes the value of the field *ZhiSharpType* of the *ObjectCreationExpression*, which then represents the term  $t = \text{“new int}(3)\text{”}$ . Function  $o(t)$  is invertible (i.e.  $o(t)$  allows

roundtrips from Zhi# terms to code DOM objects to Zhi# terms). The inverse function  $o^{-1}(o(t))$  returns the Zhi# term that is represented by the code DOM object  $o(t)$  (e.g.,  $o^{-1}(o(\text{new byte}(3))[\text{ZhiSharpType} = \text{int}]) = \text{new int}(3)$ ).

The implementing C# source code of the compiler plug-ins for XSD and OWL was annotated with comments of the form `//CF XSD 'Function' 'case no.'` and `//CF OWL 'Function' 'case no.'`, respectively. By setting up a user defined token “CF” for the Visual Studio task list [MSD09] one can easily navigate to the implementation of each case of the specified transformation functions.

## B.1 XSD Program Transformation Functions

This section specifies the implementations of the external program transformation functions that are provided by the Zhi# compiler plug-in for XML Schema Definition. The external program transformation methods *PropertyAccess*, *PropertyUpdate*, *IndexerAccess*, *IndexerUpdate*, *MethodInvocation*, and *ForEach* are not applicable to XML data types.

### UseNamespace

$$\frac{\emptyset \vdash \text{namespace} \in \blacktriangle_{\text{XSD}}}{\text{UseNamespace}(\text{namespace}) = \text{using zhisharpSimpleTypes} = \text{Zhimantic.LambdaC.Typesystem.Runtime};}$$

### GetProxyType

$$\frac{\emptyset \vdash T \in \Delta_{\text{XSD}}}{\text{GetProxyType}(T) = \text{RTSimpleType} + \text{array ranks of } T}$$

### Cast

$$\frac{\Gamma, v : V \vdash T \in \Delta_{\text{XSD}}}{\text{Cast}(T, v) = \left\{ \begin{array}{l} (\text{GetProxyType}(T)) \mathbf{v} \quad ; \curvearrowright 1 : T \in \Delta_{\text{XSD}}; \end{array} \right.}$$

## CreateObject

$$\begin{array}{c}
 \Gamma, v_i : V_i \vdash (T \in \Delta_{\text{XSD}}^\perp \vee \bigvee V_i \in \Delta_{\text{XSD}}^\perp) \\
 \hline
 \text{CreateObject}(T, \{v_i^{i \in 1..n}\}) = \\
 \left\{ \begin{array}{l}
 \text{New}(\text{"T"}, (\text{GetProxyType}(T)) \ v_1) \quad ; \curvearrow 1 : \begin{array}{l} n = 1, \\ v_1 \triangleright_{\Gamma}^{\text{impl}} T; \end{array} \\
 \\
 \text{New}(\text{"T"}, o^{-1}(o(v_1).\text{Arguments}[0])) \quad ; \curvearrow 2 : \begin{array}{l} n = 1, \\ o(v_1) \text{ is } \text{ObjectCreationExpression}, \\ V_1 \in \Delta_{\text{XSD}}^\perp, \\ T \in \Delta_{\text{XSD}}^\perp \cup \Delta_{\text{NET prim.}}^\perp; \end{array} \\
 \\
 \text{New}(\text{"T"}, v_1) \quad ; \curvearrow 3 : \begin{array}{l} o(v_1) \text{ is } \text{ObjectCreationExpression}, \\ V_1 \in \Delta_{\text{NET prim.}}^\perp, \\ T \in \Delta_{\text{XSD}}^\perp \cup \Delta_{\text{NET prim.}}^\perp; \end{array} \\
 \\
 \text{CreateObject}(V_1, v_1) \quad ; \curvearrow 4 : \begin{array}{l} T = \text{System.Object}, \\ V_1 \in \Delta^\perp; \\ n = 1, \end{array} \\
 \\
 v_1 \quad ; \curvearrow 5 : \begin{array}{l} T = \text{System.Object}, \\ V_1 \in \Delta^{\perp}; \\ n = 1, \end{array} \\
 \\
 \text{Convert.ToBoolean}((v_1).\text{ToString}()) \quad ; \curvearrow 6a : \begin{array}{l} T = \text{System.Boolean}, \\ V_1 = \text{xsd\#boolean};^1 \end{array} \\
 \\
 \text{New}(\text{"T"}, v_1) \quad ; \curvearrow 7 : \begin{array}{l} n = 1, \\ T \in \Delta_{\text{XSD}}^\perp \cup \Delta_{\text{NET prim.}}^\perp; \end{array}
 \end{array}
 \right.
 \end{array}$$

<sup>1</sup>Cases 6a – 6m are implemented analogously for all 13 .NET primitive types and compatible XML data types.

## GetObject

$$\frac{\Gamma, v_i : V_i \vdash (T \in \Delta_{\text{XSD}}^\perp \vee \bigvee V_i \in \Delta_{\text{XSD}}^\perp)}{\text{GetObject}(T, \{v_i^{i \in 1..n}\}) = \text{CreateObject}(T, \{v_i^{i \in 1..n}\})}$$

## CreateArray

$$\frac{\Gamma, v_{ij} : V_{ij} \vdash T \in \Delta_{\text{XSD}}^{[\ ]}}{\text{CreateArray}(T, \{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}) = \left\{ \begin{array}{l} n = 1, \\ o^{-1}(o(v_{11})[\text{ZhiSharpType} = \text{GetProxyType}(T)]) \quad ; \curvearrow 1 : m = 1, \\ v_{11} \text{ is } \text{ArrayCreationExpression}^2; \end{array} \right.}$$

## Assign

$$\frac{\Gamma, x : T, v : V \vdash (T \in \Delta_{\text{XSD}} \vee V \in \Delta_{\text{XSD}})}{\text{Assign}(T, x, v) = \left\{ \begin{array}{l} x = \text{null} \quad ; \curvearrow 1 : T = \text{null}; \\ x = v \quad ; \curvearrow 2 : T \in \Delta_{\text{XSD}}^{[\ ]}; \\ \text{SetValue}(\text{"T"}, \text{out } x, (\text{GetProxyType}(T)) \ v) \quad ; \curvearrow 3 : (T \in \Delta_{\text{XSD}}^\perp) \wedge (v \triangleright_\Gamma^{\text{impl}} T); \\ \text{SetValue}(\text{"T"}, \text{out } x, v) \quad ; \curvearrow 4 : (T \in \Delta_{\text{XSD}}^\perp) \wedge \overline{(v \triangleright_\Gamma^{\text{impl}} T)}; \\ x = v \quad ; \curvearrow 5 : T \in \{\Delta^{[\ ]} \setminus \Delta_{\text{XSD}}^{[\ ]}\}; \\ x = v \quad ; \curvearrow 6 : (T \in \{\Delta^\perp \setminus \Delta_{\text{XSD}}^\perp\}) \wedge (v \triangleright_\Gamma^{\text{impl}} T); \\ x = \text{CreateObject}(T, v) \quad ; \curvearrow 7 : (T \in \{\Delta^\perp \setminus \Delta_{\text{XSD}}^\perp\}) \wedge \overline{(v \triangleright_\Gamma^{\text{impl}} T)}; \end{array} \right.$$

<sup>2</sup>The *ArrayCreationExpression* must have parameters that are compatible with the base type of array type  $T$ .

## Compute

$$\begin{array}{c}
 \Gamma, x : X, y : Y \vdash (X \in \Delta_{\text{XSD}} \vee Y \in \Delta_{\text{XSD}}) \\
 \hline
 \text{Compute}(T, x, \tau, y) = \\
 \left\{ \begin{array}{l}
 \text{AreFractionDigitsOkL}(x, o(y).Literal) \ ; \curvearrowright 1a) : \begin{array}{l}
 \tau = '\%.', \\
 X \in \Delta_{\text{NET}}^{\perp} \text{ prim.} \cup \Delta_{\text{XSD}}^{\perp}, \\
 Y \in \Delta_{\text{NET}}^{\perp} \text{ prim.} \cup \Delta_{\text{XSD}}^{\perp}, \\
 o(y) \text{ is } \textit{LiteralExpression}^3; ^4
 \end{array} \\
 \\
 x \ \&\& \ y \ ; \curvearrowright 8 : \begin{array}{l}
 \tau = "\&\&", \\
 X \in \Delta_{\text{NET}}^{\perp} \text{ prim.} \cup \Delta_{\text{XSD}}^{\perp}, \\
 Y \in \Delta_{\text{NET}}^{\perp} \text{ prim.} \cup \Delta_{\text{XSD}}^{\perp}; ^5
 \end{array} \\
 \\
 x \ || \ y \ ; \curvearrowright 9 : \begin{array}{l}
 \tau = "||", \\
 X \in \Delta_{\text{NET}}^{\perp} \text{ prim.} \cup \Delta_{\text{XSD}}^{\perp}, \\
 Y \in \Delta_{\text{NET}}^{\perp} \text{ prim.} \cup \Delta_{\text{XSD}}^{\perp}; ^5
 \end{array} \\
 \\
 \text{Is}(x, "Y") \ ; \curvearrowright 10a) : \begin{array}{l}
 \tau = "is", \\
 X \in \Delta_{\text{XSD}}^{\perp};
 \end{array} \\
 \\
 \text{Is}(\text{New}("X", x), "Y") \ ; \curvearrowright 10b) : \begin{array}{l}
 \tau = "is", \\
 X \notin \Delta_{\text{XSD}}^{\perp};
 \end{array}
 \end{array} \right.
 \end{array}$$

<sup>3</sup>The literal must represent a valid value for the used relational operator (i.e. constraining facet).

<sup>4</sup>Cases 1a – 6a are implemented analogously for the %, %, >, =, <, and ?? operator.

<sup>5</sup>In the current implementation only the *xsd#boolean* data type implements the .NET *true* and *false* operators.



## Compute (continued)

$$\begin{array}{c}
 \Gamma, x : X, y : Y \vdash (X \in \Delta_{\text{XSD}} \vee Y \in \Delta_{\text{XSD}}) \\
 \hline
 \text{Compute}(T, x, \tau, y) = \\
 \left\{ \begin{array}{l}
 \begin{array}{l}
 \tau = '+', \\
 \text{Addition}(\text{"T"}, x, y) \quad ; \rightsquigarrow 11a) : X \in \Delta_{\text{XSD}}^{\perp}, \\
 Y \in \Delta_{\text{XSD}}^{\perp};^6
 \end{array} \\
 \begin{array}{l}
 \tau = '+', \\
 \text{Addition}(\text{"T"}, x, \text{New}(Y, y)) \quad ; \rightsquigarrow 11b) : X \in \Delta_{\text{XSD}}^{\perp}, \\
 Y \notin \Delta_{\text{XSD}}^{\perp};^6
 \end{array} \\
 \begin{array}{l}
 \tau = '+', \\
 \text{Addition}(\text{"T"}, \text{New}(X, x), y) \quad ; \rightsquigarrow 11c) : X \notin \Delta_{\text{XSD}}^{\perp}, \\
 Y \in \Delta_{\text{XSD}}^{\perp};^6
 \end{array} \\
 \begin{array}{l}
 \tau = "\%", \\
 \text{AreFractionDigitsOkT}(x, o(y).Type) \quad ; \rightsquigarrow 1b) : X \in \Delta_{\text{NET prim.}}^{\perp} \cup \Delta_{\text{XSD}}^{\perp}, \\
 o(y) \text{ is TypeOfExpression};^7
 \end{array} \\
 \begin{array}{l}
 \tau = "\$=", \\
 \text{IsEnumValidValueT}(x, o(y).Type) \quad ; \rightsquigarrow 7 : X \in \Delta_{\text{NET prim.}}^{\perp} \cup \Delta_{\text{XSD}}^{\perp}, \\
 o(y) \text{ is TypeOfExpression}^8;
 \end{array}
 \end{array} \right.
 \end{array}$$

---

<sup>6</sup>Cases 11a,b,c – 32a,b,c are implemented analogously for Zhi#'s binary arithmetic and comparison operators. Assignment expressions of the form  $x \tau = y$  are compiled into an explicit assignment expression of the form  $x = x \tau y$ .

<sup>7</sup>Cases 1b – 6b are implemented analogously for the %, %, ?, =, ?<, and ?? operator.

<sup>8</sup>The denoted type must be an enumerated XML data type.

## B.2 OWL Program Transformation Functions

This section specifies the implementations of the external program transformation functions that are provided by the *Zhi#* compiler plug-in for the Web Ontology Language. The external program transformation methods *Indexer Access* and *Indexer Update* are not applicable to OWL concepts, roles, and individuals. For a *Zhi#* term  $t$  and an OWL concept name  $T$  the expression  $t^T$  stands for the generated C# code `AssertKindOf(t, "T")`, which dynamically checks that the ontological individual referred to by  $t$  is in the extension of concept  $T$ .

### GetProxyType

$$\frac{\emptyset \vdash T \in \Delta_{\text{OWL}}}{\text{GetProxyType}(T) = \text{OWLIndividual} + \text{array ranks of } T}$$

### Cast

$$\frac{\Gamma, v : V \vdash T \in \Delta_{\text{OWL}} \vee V \in \Delta_{\text{OWL}}}{\text{Cast}(T, v) = \left\{ \begin{array}{ll} \text{CastOWLIndividual}(v^V, "T") & ; \curvearrowright 1a : \begin{array}{l} T \in \Delta_{\text{OWL}}^{\perp}, \\ V \in \Delta_{\text{OWL}}^{\perp}; \end{array} \\ \text{CastOWLIndividual}(v, "T") & ; \curvearrowright 1b : \begin{array}{l} T \in \Delta_{\text{OWL}}^{\perp}, \\ V \in \Delta_{\text{OWL}}^{[]}; \end{array} \\ (T) v^V & ; \curvearrowright 2a : \begin{array}{l} T \notin \Delta_{\text{OWL}}, \\ V \in \Delta_{\text{OWL}}^{\perp}, \\ v \triangleright_{\Gamma} T; \end{array} \\ (T) v & ; \curvearrowright 2b : \begin{array}{l} T \notin \Delta_{\text{OWL}}, \\ V \in \Delta_{\text{OWL}}^{[]}, \\ v \triangleright_{\Gamma} T; \end{array} \end{array} \right.$$

## CreateObject

$$\begin{array}{c}
 \Gamma, v_i : V_i \vdash (T \in \Delta_{\text{OWL}}^\perp \vee \bigvee V_i \in \Delta_{\text{OWL}}^\perp) \\
 \hline
 \text{CreateObject}(T, \{v_i^{i \in 1..n}\}) = \\
 \left\{ \begin{array}{l}
 n = 1, \\
 (\text{New}(o^{-1}(o(v_1).\text{Arguments}[0]), \text{"T"})) : T \quad ; \curvearrowright 1a : o(v_1) \text{ is } \text{ObjectCreationExpression}^9, \\
 T \in \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 \text{New}(o^{-1}(o(v_1).\text{Arguments}[0]), \text{"T"}) \quad ; \curvearrowright 1b : o(v_1) \text{ is } \text{ObjectCreationExpression}^9, \\
 T \notin \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 ((\text{GetProxyType}(T)) v_1) : T \quad ; \curvearrowright 2a : v_1 \triangleright_\Gamma^{\text{impl}} T, \\
 T \in \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 (\text{GetProxyType}(T)) v_1 \quad ; \curvearrowright 2b : v_1 \triangleright_\Gamma^{\text{impl}} T, \\
 T \notin \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 (\text{New}(v_1, \text{"T"})) : T \quad ; \curvearrowright 3a : \overline{v_1 \triangleright_\Gamma^{\text{impl}} T}, \\
 T \in \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 \text{New}(v_1, \text{"T"}) \quad ; \curvearrowright 3b : \overline{v_1 \triangleright_\Gamma^{\text{impl}} T}, \\
 T \notin \Delta_{\text{OWL}}^\perp;
 \end{array} \right.
 \end{array}$$

<sup>9</sup>The *ObjectCreationExpression* must have one parameter of type *System.String* and create an object of type  $T \in \Delta_{\text{OWL}}^\perp$ .

## GetObject

$$\frac{\Gamma, v_i : V_i \vdash (T \in \Delta_{\text{OWL}}^\perp \vee \bigvee V_i \in \Delta_{\text{OWL}}^\perp)}{\text{GetObject}(T, \{v_i^{i \in 1..n}\}) =}$$

$$\left\{ \begin{array}{l}
 n = 1, \\
 ((\text{GetProxyType}(T)) \mathbf{v}_1)^{:T} \quad ; \curvearrow 1a : v_1 \triangleright_{\Gamma}^{\text{impl}} T, \\
 T \in \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 (\text{GetProxyType}(T)) \mathbf{v}_1 \quad ; \curvearrow 1b : v_1 \triangleright_{\Gamma}^{\text{impl}} T, \\
 T \notin \Delta_{\text{OWL}}^\perp; \\
 v_1^{:V_1} \quad ; \curvearrow 2a : \frac{}{v_1 \triangleright_{\Gamma}^{\text{impl}} T}, \\
 V_1 \in \Delta_{\text{OWL}}^\perp; \\
 n = 1, \\
 \mathbf{v}_1 \quad ; \curvearrow 2b : \frac{}{v_1 \triangleright_{\Gamma}^{\text{impl}} T}, \\
 V_1 \notin \Delta_{\text{OWL}}^\perp;
 \end{array} \right.$$

## CreateArray

$$\frac{\Gamma, v_{ij} : V_{ij} \vdash T \in \Delta_{\text{OWL}}^{[\ ]}}{\text{CreateArray}(T, \{\{v_i^{i \in 1..n}\}_j^{j \in 1..m}\}) =}$$

$$\left\{ \begin{array}{l}
 n = 1, \\
 o^{-1}(o(v_{11})[\text{ZhiSharpType} = \text{GetProxyType}(T)]) \quad ; \curvearrow 1 : m = 1, \\
 v_{11} \text{ is } \text{ArrayCreationExpression}^{10};
 \end{array} \right.$$

<sup>10</sup>The *ArrayCreationExpression* must have parameters that are compatible with the base type of array type  $T$ .

## Compute

$$\begin{array}{c}
 \Gamma, x : X, y : Y \vdash (X \in \Delta_{\text{OWL}} \vee Y \in \Delta_{\text{OWL}}) \\
 \hline
 \text{Compute}(T, x, \tau, y) = \\
 \left\{ \begin{array}{l}
 \tau = \text{"is"}, \\
 \text{Is}(x, \text{"Y"}) \quad ; \curvearrow 1 : X \in \Delta_{\text{OWL}}^{\perp}, \\
 \quad \quad \quad Y \in \Delta_{\text{OWL}}^{\perp}; \\
 x == y \quad ; \curvearrow 2 : \tau = \text{"=="}, \\
 \quad \quad \quad (X = \text{null}) \vee (Y = \text{null}); \\
 \text{Identity}(x, y) \quad ; \curvearrow 3a : X \in \Delta_{\text{OWL}}^{\perp}, \\
 \quad \quad \quad Y \in \Delta_{\text{OWL}}; \\
 \quad \quad \quad \tau = \text{"=="}, \\
 \text{Identity}(y, x) \quad ; \curvearrow 3b : X \in \Delta_{\text{OWL}}, \\
 \quad \quad \quad Y \in \Delta_{\text{OWL}}^{\perp}; \\
 x == y \quad ; \curvearrow 4 : \tau = \text{"=="}; \\
 x == y \quad ; \curvearrow 5 : \tau = \text{"!="}, \\
 \quad \quad \quad (X = \text{null}) \vee (Y = \text{null}); \\
 \quad \quad \quad \tau = \text{"!="}, \\
 \text{NIdentity}(x, y) \quad ; \curvearrow 6a : X \in \Delta_{\text{OWL}}^{\perp}, \\
 \quad \quad \quad Y \in \Delta_{\text{OWL}}; \\
 \quad \quad \quad \tau = \text{"!="}, \\
 \text{NIdentity}(y, x) \quad ; \curvearrow 6b : X \in \Delta_{\text{OWL}}, \\
 \quad \quad \quad Y \in \Delta_{\text{OWL}}^{\perp}; \\
 x == y \quad ; \curvearrow 7 : \tau = \text{"!="};
 \end{array} \right.
 \end{array}$$

## Assign

$$\frac{\Gamma, x : T, v : V \vdash (T \in \Delta_{OWL} \vee V \in \Delta_{OWL})}{\text{Assign}(T, x, v) = \begin{cases} \mathbf{x} = v^V & ; \curvearrow 1 : V \in \Delta_{OWL}^\perp; \\ \mathbf{x} = \mathbf{v} & ; \curvearrow 2 : V \notin \Delta_{OWL}^\perp; \end{cases}}$$

## PropertyAccess

$$\frac{\Gamma, h : H \vdash H \in \Delta_{OWL}^\perp}{\text{PropertyAccess}(T, h, p, q) = \begin{cases} \text{Exists("H")} & ; \curvearrow 1a : \begin{array}{l} o(h).IsStaticTypeReference, \\ p = \text{"Exists"}; \end{array} \\ \text{Count("H")} & ; \curvearrow 1b : \begin{array}{l} o(h).IsStaticTypeReference, \\ p = \text{"Count"}; \end{array} \\ \text{Individuals("H")} & ; \curvearrow 1c : \begin{array}{l} o(h).IsStaticTypeReference, \\ p = \text{"Individuals"}; \end{array} \\ \text{ExistsObjectPropertyValue}(h, "p") & ; \curvearrow 2a : \begin{array}{l} \overline{o(h).IsStaticTypeReference}, \\ p \in \Delta_{OWL}^\perp \times \Delta_{OWL}^\perp, \\ q = \text{"Exists"}; \end{array} \\ \text{ExistsDatatypePropertyValue}(h, "p") & ; \curvearrow 2b : \begin{array}{l} \overline{o(h).IsStaticTypeReference}, \\ p \in \Delta_{OWL}^\perp \times \Delta_{XSD}^\perp, \\ q = \text{"Exists"}; \end{array} \end{cases}}$$

PropertyAccess (continued)

$$\begin{array}{c}
 \Gamma, h : H \vdash H \in \Delta_{\text{OWL}}^{\perp} \\
 \hline
 \text{PropertyAccess}(T, h, p, q) = \\
 \left\{ \begin{array}{ll}
 \text{CountObjectPropertyValues}(h, "p") & ; \rightsquigarrow 3a : \overline{o(h).IsStaticTypeReference}, \\
 & p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \\
 & q = \text{"Count"}; \\
 \text{CountDatatypePropertyValues}(h, "p") & ; \rightsquigarrow 3b : \overline{o(h).IsStaticTypeReference}, \\
 & p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}, \\
 & q = \text{"Count"}; \\
 \text{GetTypesOfIndividual}(h) & ; \rightsquigarrow 4a : \overline{o(h).IsStaticTypeReference}, \\
 & p = \text{"Types"}; \\
 \text{GetEquivalentIndividuals}(h) & ; \rightsquigarrow 4b : \overline{o(h).IsStaticTypeReference}, \\
 & p = \text{"EquivalentIndividuals"}; \\
 \text{GetObjectPropertyValues}(h, "p") & ; \rightsquigarrow 5 : \overline{o(h).IsStaticTypeReference}, \\
 & p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}; \\
 \text{GetDatatypePropertyValue}(h, "p") & ; \rightsquigarrow 6a : \overline{o(h).IsStaticTypeReference}, \\
 & p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}, \\
 & \top \sqsubseteq \leq 1p; \\
 \text{GetDatatypePropertyValues}(h, "p") & ; \rightsquigarrow 6b : \overline{o(h).IsStaticTypeReference}, \\
 & p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp};
 \end{array} \right.
 \end{array}$$

## PropertyUpdate

$$\begin{array}{c}
 \Gamma, h : H, v : V \vdash H \in \Delta_{\text{OWL}}^{\perp} \\
 \hline
 \text{PropertyUpdate}(T, h, p, v) = \\
 \left\{ \begin{array}{l}
 \text{AddObjectPropertyValuesChecked}(h^{:H}, \text{"p"}, v^{:V}) \quad ; \curvearrowright 1a : \begin{array}{l} V \in \Delta_{\text{OWL}}^{\perp}, \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \\ v \text{ is checked}; \end{array} \\
 \text{AddObjectPropertyValuesChecked}(h^{:H}, \text{"p"}, v) \quad ; \curvearrowright 1b : \begin{array}{l} V \in \Delta_{\text{OWL}}^{[\ ]}, \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \\ v \text{ is checked}; \end{array} \\
 \text{AddObjectPropertyValues}(h^{:H}, \text{"p"}, v^{:V}) \quad ; \curvearrowright 2a : \begin{array}{l} V \in \Delta_{\text{OWL}}^{\perp}, \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \\ v \text{ is not checked}; \end{array} \\
 \text{AddObjectPropertyValues}(h^{:H}, \text{"p"}, v) \quad ; \curvearrowright 2b : \begin{array}{l} V \in \Delta_{\text{OWL}}^{[\ ]}, \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \\ v \text{ is not checked}; \end{array} \\
 \text{SetDatatypePropertyValue}(h^{:H}, \text{"p"}, v) \quad ; \curvearrowright 3a : \begin{array}{l} V \in \Delta_{\text{XSD}}^{\perp}, \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}, \\ \top \sqsubseteq \leq 1p; \end{array} \\
 \text{AddDatatypePropertyValues}(h^{:H}, \text{"p"}, v) \quad ; \curvearrowright 3b : \begin{array}{l} V \in \Delta_{\text{XSD}}^{[\ ]}, \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}; \end{array}
 \end{array}
 \right.
 \end{array}$$



## MethodInvocation

$$\begin{array}{c}
 \Gamma, h : H, v_i : V_i \vdash H \in \Delta_{\text{OWL}}^{\perp} \\
 \hline
 \text{MethodInvocation}(T, h, p, m, \{v_i\}_{i \in 1..n}) = \\
 \left\{ \begin{array}{ll}
 \text{ClearObjectPropertyValues}(h, \text{"p"}) & ; \curvearrow 1a : \begin{array}{l} p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \\ m = \text{"Clear"}; \end{array} \\
 \text{ClearDatatypePropertyValues}(h, \text{"p"}) & ; \curvearrow 1b : \begin{array}{l} p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}, \\ m = \text{"Clear"}; \end{array} \\
 \text{RemoveObjectPropertyValues}(h, \text{"p"}, v_1) & ; \curvearrow 2a : \begin{array}{l} n = 1, \\ V_1 \in \Delta_{\text{OWL}}^{[\ ]}, \\ m = \text{"Remove"}; \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{OWL}}^{\perp}, \end{array} \\
 \text{RemoveObjectPropertyValues}(h, \text{"p"}, v_1, \dots, v_n) & ; \curvearrow 2b : \begin{array}{l} n \geq 1, \\ V_i\_{i \in 1..n} \in \Delta_{\text{OWL}}^{\perp}, \\ m = \text{"Remove"}; \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}, \end{array} \\
 \text{RemoveDatatypePropertyValues}(h, \text{"p"}, v_1) & ; \curvearrow 3a : \begin{array}{l} n = 1, \\ V_1 \in \Delta_{\text{XSD}}^{[\ ]}, \\ m = \text{"Remove"}; \\ p \in \Delta_{\text{OWL}}^{\perp} \times \Delta_{\text{XSD}}^{\perp}, \end{array} \\
 \text{RemoveDatatypePropertyValues}(h, \text{"p"}, v_1, \dots, v_n) & ; \curvearrow 3b : \begin{array}{l} n \geq 1, \\ V_i\_{i \in 1..n} \in \Delta_{\text{XSD}}^{\perp}, \\ m = \text{"Remove"}; \end{array}
 \end{array}
 \right.
 \end{array}$$

## MethodInvocation (continued)

$$\frac{\Gamma, h : H, v_i : V_i \vdash H \in \Delta_{\text{OWL}}^\perp}{\text{MethodInvocation}(T, h, p, m, \{v_i^{i \in 1..n}\}) =}$$

{	SameAs(h, v <sub>1</sub> )	; $\curvearrowright$ 4a : $V_1 \in \Delta_{\text{OWL}}^{[]}$ ,
		$n = 1,$
		$m = \text{"SameAs"};$
		$n \geq 1,$
	SameAs(h, v <sub>1</sub> , ..., v <sub>n</sub> )	; $\curvearrowright$ 4b : $V_i^{i \in 1..n} \in \Delta_{\text{OWL}}^\perp,$
		$m = \text{"SameAs"};$
	DifferentFrom(h, v <sub>1</sub> )	; $\curvearrowright$ 5a : $V_1 \in \Delta_{\text{OWL}}^{[]}$ ,
		$n = 1,$
		$m = \text{"DifferentFrom"};$
		$n \geq 1,$
	DifferentFrom(h, v <sub>1</sub> , ..., v <sub>n</sub> )	; $\curvearrowright$ 5b : $V_i^{i \in 1..n} \in \Delta_{\text{OWL}}^\perp,$
		$m = \text{"DifferentFrom"};$

## ForEach

$$\frac{\Gamma, v : V \vdash V \in \Delta_{\text{OWL}}}{\text{ForEach}(T, v) = \text{ForEach}(\text{"T"}, v)^{11}}$$

## UseNamespace

$$\frac{\emptyset \vdash \text{namespace} \in \blacktriangle_{\text{OWL}}}{\text{UseNamespace}(\text{namespace}) = \text{null}}$$

<sup>11</sup>The object  $v$  must provide a *GetEnumerator()* method, which returns an *IEnumerator* object.

# APPENDIX C

## The $\lambda_C$ -Calculus

Table C.1 summarizes the syntax of  $\lambda_C$ -terms and type definitions. Tables C.2, C.3, C.4, and C.5 summarize the evaluation, typing, subtyping, and type derivation rules of the  $\lambda_C$ -calculus. Gray backgrounds highlight extensions of the  $\lambda_{<}$ -calculus.

Table C.1: Constrained types calculus ( $\lambda_C$ ) syntax

---

### *Syntax*

$t ::=$

$x$

$\lambda x : T.t$

$tt$

*terms:*

*variable*

*abstraction*

*application*

$v ::=$

$\lambda x : T.t$

*true*

*false*

*values:*

*abstraction value*

*true value*

*false value*

over, please

---

Table C.1: Constrained types calculus ( $\lambda_C$ ) syntax

---

**Syntax**

$T ::=$

*types:*

$T \rightarrow T$

*type of functions*

$Top$

*maximum type*

$Bool$

*type of booleans*

$P_{\text{xsd}\#\text{boolean}}, \dots, P_{\text{xsd}\#\text{string}}$

*primitive base types*

$T.c$

*constraint application*

$c ::=$

*constraints:*

$c = \phi(TV)\{x|x \in v(TV)\} \bigcap_{k \in 1..m} \{x|x \prec literal_k\}$

*constraint*

$\Gamma ::=$

*contexts:*

$\emptyset$

*empty context*

$\Gamma, x : T$

*term variable binding*

---

Table C.2: Constrained types calculus ( $\lambda_C$ ) evaluation

---

<i>Evaluation</i>	$t \rightarrow t'$
$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$	(E-APP1)
$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2}$	(E-APP2)
$(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$	(E-APPABS)
<i>if true then <math>t_2</math> else <math>t_3 \rightarrow t_2</math></i>	(E-IFTRUE)
<i>if false then <math>t_2</math> else <math>t_3 \rightarrow t_3</math></i>	(E-IFTRUE)
$\frac{t_1 \rightarrow t'_1}{\textit{if } t_1 \textit{ then } t_2 \textit{ else } t_3 \rightarrow \textit{if } t'_1 \textit{ then } t_2 \textit{ else } t_3}$	(E-IF)

---

Table C.3: Constrained types calculus ( $\lambda_C$ ) typing

---

<b><i>Typing</i></b>	$\Gamma \vdash t : T$
$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)
$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2}$	(T-ABS)
$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(T-APP)
$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$	(T-SUB)
$\mathit{true} : \mathit{Bool}$	(T-TRUE)
$\mathit{false} : \mathit{Bool}$	(T-FALSE)
$\frac{t_1 : \mathit{Bool} \quad t_2 : T \quad t_3 : T}{\mathit{if } t_1 \mathit{ then } t_2 \mathit{ else } t_3 : T}$	(T-IF)

---

Table C.4: Constrained types calculus ( $\lambda_C$ ) subtyping

---

<i>Subtyping</i>	$S <: T$
$S <: S$	(S-REFL)
$\frac{S <: U \quad U <: T}{S <: T}$	(S-TRANS)
$S <: Top$	(S-TOP)
$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$	(S-ARROW)
$\frac{v(S) \subseteq v(T)}{S <: T}$	(S-VSPACE)
$\frac{S = \bigcap_{i \in 1..n+k}^T c_i \quad U = \bigcap_{i \in 1..n}^T c_i}{S <: U}$	(S-WIDTH)
$\frac{\text{for each } i \ c_i <:: d_i \quad S = \bigcap_{i \in 1..n}^T c_i \quad U = \bigcap_{i \in 1..n}^T d_i}{S <: U}$	(S-DEPTH)
$\frac{S <: T}{S.c <: T}$	(S-APP)

over, please

Table C.4: Constrained types calculus ( $\lambda_C$ ) subtyping

---

<i>Subtyping</i>	$S <: T$
$\frac{v(c_1\{\{TV \leftarrow T\}\}) \subseteq v(c_2\{\{TV \leftarrow T\}\})}{c_1 <:: c_2}$	(S-CSTRVSPACE)
$\phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n+k} \{x x < y_i\} <:: \phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n} \{x x < y_i\}$	(S-CSTRWIDTH)
$\frac{\text{for each } i \{x x < y_i\} \subseteq \{x x < z_i\}}{\phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n} \{x x < y_i\} <:: \phi(TV)\{x x \in v(TV)\} \bigcap_{i \in 1..n} \{x x < z_i\}}$	(S-CSTRDEPTH)

---

Table C.5: Constrained types calculus ( $\lambda_C$ ) type derivation and substitution

---

<i>Type derivation and substitution</i>	$T \rightarrow T'$
$\frac{T' = T.c}{v(T') = v(c\{\{TV \leftarrow T\}\})}$	(TD-CSTRAPP)
$\frac{T \rightarrow T'}{T.c \rightarrow T'.c}$	(TD-SUBS)

---



# APPENDIX D

## XSD Type Inference Rules

Table D.1: TI-IFADD

$$\frac{\Gamma \vdash a : A \quad \text{if } \left( \bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i) \right) \text{ then } \diamond}{\Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x \mid x \in v(A)\} \cap \{x \mid x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

Table D.2: TI-FORADD

$$\frac{\Gamma \vdash a : A \quad \text{for } ([\text{initializers}]; \bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i); [\text{iterators}]) \text{ then } \diamond}{\Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x \mid x \in v(A)\} \cap \{x \mid x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

Table D.3: TI-WHILEADD

$$\frac{\Gamma \vdash a : A \quad \text{while } \left( \bigwedge_{i \in 1..n} (a \prec_i \text{literal}_i) \right) \text{ then } \diamond}{\Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n}^A c_i \quad \text{where } c_i = \{x \mid x \in v(A)\} \cap \{x \mid x \prec_i \text{literal}_i\}^{i \in 1..n}}$$

Table D.4: TI-ASSIGNREM

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : A \quad \Gamma_{\diamond} \vdash a : \bigcap_{i \in 1..n} c_i \quad a := t}{\Gamma_{\diamond\diamond} \vdash a : A}$$

# APPENDIX E

## XSD Compile-Time Arithmetic

### E.1 XSD Type Arithmetic

Table E.1: XSD type arithmetic

<b>Type</b> $T_1$	<b>Operator</b> $\tau$	<b>Type</b> $T_2$	$T = T_1 \tau T_2$
<i>anyURI</i>	==	<i>anyURI</i>	<i>boolean</i>
<i>base64Binary</i>	==	<i>base64Binary</i>	<i>boolean</i>
<i>boolean</i>	==	<i>boolean</i>	<i>boolean</i>
<i>boolean</i>	&	<i>boolean</i>	<i>boolean</i>
<i>boolean</i>		<i>boolean</i>	<i>boolean</i>
<i>boolean</i>	^	<i>boolean</i>	<i>boolean</i>
<i>date</i>	<	<i>date</i>	<i>boolean</i>
<i>date</i>	==	<i>date</i>	<i>boolean</i>
<i>date</i>	>	<i>date</i>	<i>boolean</i>
<i>date</i>	+	<i>duration</i>	<i>dateTime</i>
<i>dateTime</i>	<	<i>dateTime</i>	<i>boolean</i>
<i>dateTime</i>	==	<i>dateTime</i>	<i>boolean</i>
<i>dateTime</i>	>	<i>dateTime</i>	<i>boolean</i>
<i>dateTime</i>	+	<i>duration</i>	<i>dateTime</i>
<i>decimal</i>	<	<i>decimal</i>	<i>boolean</i>
<i>decimal</i>	==	<i>decimal</i>	<i>boolean</i>

Table E.1: XSD type arithmetic

<b>Type</b> $T_1$	<b>Operator</b> $\tau$	<b>Type</b> $T_2$	$T = T_1 \tau T_2$
<i>decimal</i>	>	<i>decimal</i>	<i>boolean</i>
<i>decimal</i>	+	<i>decimal</i>	<i>decimal</i>
<i>decimal</i>	-	<i>decimal</i>	<i>decimal</i>
<i>decimal</i>	*	<i>decimal</i>	<i>decimal</i>
<i>decimal</i>	/	<i>decimal</i>	<i>decimal</i>
<i>decimal</i>	%	<i>decimal</i>	<i>decimal</i>
<i>double</i>	<	<i>double</i>	<i>boolean</i>
<i>double</i>	==	<i>double</i>	<i>boolean</i>
<i>double</i>	>	<i>double</i>	<i>boolean</i>
<i>double</i>	+	<i>double</i>	<i>double</i>
<i>double</i>	-	<i>double</i>	<i>double</i>
<i>double</i>	*	<i>double</i>	<i>double</i>
<i>double</i>	/	<i>double</i>	<i>double</i>
<i>double</i>	%	<i>double</i>	<i>double</i>
<i>duration</i>	<	<i>duration</i>	<i>boolean</i>
<i>duration</i>	==	<i>duration</i>	<i>boolean</i>
<i>duration</i>	>	<i>duration</i>	<i>boolean</i>
<i>duration</i>	+	<i>duration</i>	<i>duration</i>
<i>duration</i>	+	<i>date</i>	<i>dateTime</i>
<i>duration</i>	+	<i>dateTime</i>	<i>dateTime</i>
<i>float</i>	<	<i>float</i>	<i>boolean</i>
<i>float</i>	==	<i>float</i>	<i>boolean</i>
<i>float</i>	>	<i>float</i>	<i>boolean</i>
<i>float</i>	+	<i>float</i>	<i>float</i>
<i>float</i>	-	<i>float</i>	<i>float</i>
<i>float</i>	*	<i>float</i>	<i>float</i>

Table E.1: XSD type arithmetic

<b>Type</b> $T_1$	<b>Operator</b> $\tau$	<b>Type</b> $T_2$	$T = T_1 \tau T_2$
<i>float</i>	/	<i>float</i>	<i>float</i>
<i>float</i>	%	<i>float</i>	<i>float</i>
<i>gDay</i>	<	<i>gDay</i>	<i>boolean</i>
<i>gDay</i>	==	<i>gDay</i>	<i>boolean</i>
<i>gDay</i>	>	<i>gDay</i>	<i>boolean</i>
<i>gMonth</i>	<	<i>gMonth</i>	<i>boolean</i>
<i>gMonth</i>	==	<i>gMonth</i>	<i>boolean</i>
<i>gMonth</i>	>	<i>gMonth</i>	<i>boolean</i>
<i>gMonthDay</i>	<	<i>gMonthDay</i>	<i>boolean</i>
<i>gMonthDay</i>	==	<i>gMonthDay</i>	<i>boolean</i>
<i>gMonthDay</i>	>	<i>gMonthDay</i>	<i>boolean</i>
<i>gYear</i>	<	<i>gYear</i>	<i>boolean</i>
<i>gYear</i>	==	<i>gYear</i>	<i>boolean</i>
<i>gYear</i>	>	<i>gYear</i>	<i>boolean</i>
<i>gYearMonth</i>	<	<i>gYearMonth</i>	<i>boolean</i>
<i>gYearMonth</i>	==	<i>gYearMonth</i>	<i>boolean</i>
<i>gYearMonth</i>	>	<i>gYearMonth</i>	<i>boolean</i>
<i>hexBinary</i>	==	<i>hexBinary</i>	<i>boolean</i>
<i>NOTATION</i>	==	<i>NOTATION</i>	<i>boolean</i>
<i>QName</i>	==	<i>QName</i>	<i>boolean</i>
<i>string</i>	==	<i>string</i>	<i>boolean</i>
<i>string</i>	+	<i>string</i>	<i>string</i>
<i>time</i>	<	<i>time</i>	<i>boolean</i>
<i>time</i>	==	<i>time</i>	<i>boolean</i>
<i>time</i>	>	<i>time</i>	<i>boolean</i>

## E.2 XSD Constraint Arithmetic

In the following table, constraints are given in the form  $\{operator\ variable\}$ , where a set of  $n$  constraints is denoted as  $\{\{operator_1\ variable_1\}, \dots, \{operator_n\ variable_n\}\}$ . Given on the left sides of the following propositions are initially existing constraints and facts about used variables. A bar over a constraint indicates that this constraint does not exist in the initial constraint set. Given on the right sides are homogenized (rules CMR-H-XSD-X) and inferred (rules CMR-I-XSD-X) sets of XML Schema Definition constraints.

Table E.2: XSD constraint materialization rules

Rule name	Rule
CMR-H-XSD-MINEX-MININ	$\{\{> a\}, \{\geq b\}\} \wedge a \geq b \implies \{\{> a\}, \{\geq a\}\}$
CMR-H-XSD-MINEX-MININ-FD1	$\{\{> a\}, \{\geq b\}, \{\%.c\}\} \wedge a \geq b \implies \{\{> a\}, \{\geq a + 10^{-c}\}, \{\%.c\}\}$
CMR-H-XSD-MINEX-MININ-FD2	$\{\{> a\}, \{\geq b\}, \{\%.c\}\} \wedge a < b \implies \{\{> b - 10^{-c}\}, \{\geq b\}, \{\%.c\}\}$
CMR-H-XSD-MAXEX-MAXIN	$\{\{\< a\}, \{\leq b\}\}, a \leq b \implies \{\{\< a\}, \{\leq a\}\}$
CMR-H-XSD-MAXEX-MAXIN-FD1	$\{\{\< a\}, \{\leq b\}, \{\%.c\}\} \wedge a \leq b \implies \{\{\< a\}, \{\leq a - 10^{-c}\}, \{\%.c\}\}$
CMR-H-XSD-MAXEX-MAXIN-FD2	$\{\{\< a\}, \{\leq b\}, \{\%.c\}\} \wedge a > b \implies \{\{\< b + 10^{-c}\}, \{\leq b\}, \{\%.c\}\}$
CMR-H-XSD-TD-FD	$\{\{\%\% a\}, \{\%.b\}\} \wedge a < b \implies \{\{\%\% a\}, \{\%.a\}\}$
CMR-I-XSD-MINEX-MININ	$\{\{> a\}, \{\overline{\geq b}\}\} \implies \{\{> a\}, \{\geq a\}\}$
CMR-I-XSD-MINEX-MININ-FD	$\{\{> a\}, \{\overline{\geq b}\}, \{\%.c\}\} \implies \{\{> a\}, \{\geq a + 10^{-c}\}, \{\%.c\}\}$
CMR-I-XSD-MININ-MINEX-FD	$\{\{>= a\}, \{\overline{> b}\}, \{\%.c\}\} \implies \{\{>= a\}, \{> a - 10^{-c}\}, \{\%.c\}\}$
CMR-I-XSD-MAXEX-MAXIN	$\{\{\< a\}, \{\overline{\leq b}\}\} \implies \{\{\< a\}, \{\leq a\}\}$
CMR-I-XSD-MAXEX-MAXIN-FD	$\{\{\< a\}, \{\overline{\leq b}\}, \{\%.c\}\} \implies \{\{\< a\}, \{\leq a - 10^{-c}\}, \{\%.c\}\}$
CMR-I-XSD-MAXIN-MAXEX-FD	$\{\{\leq a\}, \{\overline{\< b}\}, \{\%.c\}\} \implies \{\{\leq a\}, \{\< a + 10^{-c}\}, \{\%.c\}\}$
CMR-I-XSD-TD-FD	$\{\{\%\% a\}, \{\overline{\%.b}\}\} \implies \{\{\%\% a\}, \{\%.a\}\}$

In the following table, constraints are given in the form  $\{operator\ variable\}$ , where a capitalized variable name indicates a set of elements. Operations on two sets (e.g.,  $A + B$ ) stand for the cross product of the two sets with the used operator. The given rules are to be interpreted as follows. If instances of two constrained types  $T.c_1$  and  $T.c_2$  are related by operator  $\tau$  in a term  $T.c_1 \tau T.c_2$  then the result is a value of the constrained type  $T.(c_1 \tau c_2)$ .

Table E.3: XSD constraint arithmetic rules

Rule name	$c_1$	$\tau$	$c_2$	$c = c_1 \tau c_2$
CA-XSD-MINLENGTH-PLUS-MINLENGTH	$\{?> a\}$	+	$\{?> b\}$	$\{?> a + b\}$
CA-XSD-MAXLENGTH-PLUS-MAXLENGTH	$\{?< a\}$	+	$\{?< b\}$	$\{?< a + b\}$
CA-XSD-ENUM-PLUS-ENUM	$\{\$ = A\}$	+	$\{\$ = B\}$	$\{\$ = A + B\}$
CA-XSD-ENUM-MINUS-ENUM	$\{\$ = A\}$	-	$\{\$ = B\}$	$\{\$ = A - B\}$
CA-XSD-ENUM-MULT-ENUM	$\{\$ = A\}$	*	$\{\$ = B\}$	$\{\$ = A * B\}$
CA-XSD-ENUM-DIV-ENUM	$\{\$ = A\}$	/	$\{\$ = B\}$	$\{\$ = A/B\}$
CA-XSD-MININ-PLUS-MININ	$\{\geq a\}$	+	$\{\geq b\}$	$\{\geq a + b\}$
CA-XSD-MININ-PLUS-MINEX	$\{\geq a\}$	+	$\{> b\}$	$\{> a + b\}$
CA-XSD-MININ-MINUS-MAXIN	$\{\geq a\}$	-	$\{\leq b\}$	$\{\geq a - b\}$
CA-XSD-MININ-MINUS-MAXEX	$\{\geq a\}$	-	$\{< b\}$	$\{> a - b\}$
CA-XSD-MINEX-PLUS-MINEX	$\{> a\}$	+	$\{> b\}$	$\{> a + b\}$
CA-XSD-MINEX-PLUS-MININ	$\{> a\}$	+	$\{\geq b\}$	$\{> a + b\}$
CA-XSD-MINEX-MINUS-MAXEX	$\{> a\}$	-	$\{< b\}$	$\{> a - b\}$
CA-XSD-MINEX-MINUS-MAXIN	$\{> a\}$	-	$\{\leq b\}$	$\{> a - b\}$
CA-XSD-MAXEX-PLUS-MAXEX	$\{< a\}$	+	$\{< b\}$	$\{< a + b\}$
CA-XSD-MAXEX-PLUS-MAXIN	$\{< a\}$	+	$\{\leq b\}$	$\{< a + b\}$

Table E.3: XSD constraint arithmetic rules

<b>Rule name</b>	$c_1$	$\tau$	$c_2$	$c = c_1 \tau c_2$
CA-XSD-MAXEX-MINUS-MINEX	$\{< a\}$	$-$	$\{> b\}$	$\{< a - b\}$
CA-XSD-MAXEX-MINUS-MININ	$\{< a\}$	$-$	$\{\geq b\}$	$\{< a - b\}$
CA-XSD-MAXIN-PLUS-MAXIN	$\{\leq a\}$	$+$	$\{\leq b\}$	$\{\leq a - b\}$
CA-XSD-MAXIN-PLUS-MAXEX	$\{\leq a\}$	$+$	$\{< b\}$	$\{< a + b\}$
CA-XSD-MAXIN-MINUS-MININ	$\{\leq a\}$	$-$	$\{\geq b\}$	$\{\leq a - b\}$
CA-XSD-MAXIN-MINUS-MINEX	$\{\leq a\}$	$-$	$\{> b\}$	$\{< a - b\}$
CA-XSD-TD-PLUS-TD	$\{\% \% a\}$	$+$	$\{\% \% b\}$	$\{\% \% a + b\}$
CA-XSD-TD-MINUS-TD	$\{\% \% a\}$	$-$	$\{\% \% b\}$	$\{\% \% a + b\}$
CA-XSD-TD-MULT-TD	$\{\% \% a\}$	$*$	$\{\% \% b\}$	$\{\% \% a + b\}$
CA-XSD-FD-PLUS-FD	$\{\% . a\}$	$+$	$\{\% . b\}$	$\{\% . a + b\}$
CA-XSD-FD-MINUS-FD	$\{\% . a\}$	$-$	$\{\% . b\}$	$\{\% . a + b\}$
CA-XSD-FD-MULT-FD	$\{\% . a\}$	$*$	$\{\% . b\}$	$\{\% . a + b\}$



## APPENDIX F

### CHIL OWL API Preconditions

The CHIL OWL API preconditions are given as conjunctions of Boolean expressions, which must be true before the execution of particular methods of the CHIL OWL API. Each clause in the Boolean expressions is a statement about, for example, the ontology under consideration. Preconditions are parameterized with formal parameters, which can be used in the Boolean expressions in the *Requires* field of the precondition specification. In addition, the precondition specifications comprise protocols to create elements in the ontology under consideration and to bind the precondition parameters to these elements such that, the specified conditions initially evaluate to false and eventually to true. The specified exceptions are expected if preconditions are violated. Required predecessors are given to allow for chained up preconditions whose Boolean expressions in the *Requires* field assume a particular context (e.g., an individual with a particular name in the ontology). The provided information proved to be sufficient to automatically generate regression test code for the formally specified methods of the CHIL OWL API. In the following specifications, the formal parameters  $C$ ,  $D$ ,  $R$ ,  $U$ , and  $o$  stand for ontological concepts, data types, abstract roles, datatype roles, and individuals. The sets  $N_C$ ,  $N_D$ ,  $N_R$ ,  $N_U$ , and  $N_I$  contain the names of the concepts, data types, abstract roles, datatype roles, and individuals of the ontology under consideration (e.g.,  $C \in N_C$  means that a concept description named  $C$  is declared in the ontology under consideration). The names  $C_{\text{fresh}}$ ,  $D_{\text{fresh}}$ ,  $R_{\text{fresh}}$ ,  $U_{\text{fresh}}$ , and  $o_{\text{fresh}}$  stand for fresh ontological concepts, data types, abstract roles, datatype roles, and individuals that have neither been declared in the ontology under consideration and have not been assigned to formal precondition parameters.

---

PC-OWLAPI-DECLARED-CONCEPT( $C$ )

---

Requires:  $C \in N_C$

---

Required predecessor: —

---

Creates: —

Binds:  $C = C_{\text{fresh}}$

Expects: *UndeclaredConceptException*

Creates:  $C_A \sqsubseteq \top$

Binds:  $C = C_A$

---

PC-OWLAPI-DECLARED-CONCEPTS( $C_1, C_2$ )

---

Requires:  $C_1 \in N_C, C_2 \in N_C$

---

Required predecessor: —

---

Creates: —

Binds:  $C_1 = C_{\text{fresh}_1}, C_2 = C_{\text{fresh}_2}$

Expects: *UndeclaredConceptException*

Creates:  $C_A \sqsubseteq \top, C_B \sqsubseteq \top$

Binds:  $C_1 = C_A, C_2 = C_B$

---

PC-OWLAPI-DECLARED-DATATYPE( $D$ )

---

Tests:  $D \in N_D$

---

Required predecessor: —

---

Creates: —

Binds:  $D = D_{\text{fresh}}$

Expects: *UndeclaredDatatypeException*

Creates: —

Binds:  $D = xsd\#\text{integer}$

---

PC-OWLAPI-IS-VALID-DATAVALUE( $v, D$ )

---

Requires:	$v^{\mathbf{D}} \in D^{\mathbf{D}}$
Required predecessor:	PC-OWLAPI-DECLARED-DATATYPE( $D$ )
Creates:	—
Binds:	$v = \text{"NaN"}, D = \text{xsd\#integer}$
Expects:	<i>InvalidDataValueException</i>
Creates:	—
Binds:	$v = 100, D = \text{xsd\#integer}$

---

PC-OWLAPI-DECLARED-INDIVIDUAL( $o$ )

---

Requires:	$o \in N_{\mathbf{I}}$
Required predecessor:	—
Creates:	—
Binds:	$o = o_{\text{fresh}}$
Expects:	<i>UndeclaredIndividualException</i>
Creates:	$C_A \sqsubseteq \top, C_A(o_A)$
Binds:	$o = o_A$

---

PC-OWLAPI-DECLARED-INDIVIDUALS( $o_1, o_2$ )

---

Requires:	$o_1 \in N_{\mathbf{I}}, o_2 \in N_{\mathbf{I}}$
Required predecessor:	—
Creates:	—
Binds:	$o_1 = o_{\text{fresh}_1}, o_2 = o_{\text{fresh}_2}$
Expects:	<i>UndeclaredIndividualException</i>
Creates:	$C_A \sqsubseteq \top, C_A(o_A), C_B \sqsubseteq \top, C_B(o_B)$
Binds:	$o_1 = o_A, o_2 = o_B$

---

PC-OWLAPI-DECLARED-OBJECTPROPERTY( $R$ )

---

Requires:	$R \in N_R$
Required predecessor:	—
Creates:	—
Binds:	$R = R_{\text{fresh}}$
Expects:	<i>UndeclaredObjectPropertyException</i>
Creates:	$C_A \sqsubseteq \top, C_B \sqsubseteq \top, R_{AB} \sqsubseteq C_A \times C_B$
Binds:	$R = R_{AB}$

---

PC-OWLAPI-UNDECLARED-OBJECTPROPERTY( $R$ )

---

Requires:	$R \notin N_R$
Required predecessor:	—
Creates:	$C_A \sqsubseteq \top, C_B \sqsubseteq \top, R_{AB} \sqsubseteq C_A \times C_B$
Binds:	$R = R_{AB}$
Expects:	<i>InvalidPropertyNameException</i>
Creates:	—
Binds:	$R = R_{\text{fresh}}$

---

PC-OWLAPI-DECLARED-DATATYPEPROPERTY( $U$ )

---

Requires:	$U \in N_U$
Required predecessor:	—
Creates:	—
Binds:	$U = U_{\text{fresh}}$
Expects:	<i>UndeclaredDatatypePropertyException</i>
Creates:	$C_A \sqsubseteq \top, U_{\text{Abyte}} \sqsubseteq C_A \times \text{xsd\#byte}$
Binds:	$U = U_{\text{Abyte}}$

---

PC-OWLAPI-UNDECLARED-DATATYPEPROPERTY( $U$ )

---

Requires:	$U \notin N_U$
Required predecessor:	—
Creates:	$C_A \sqsubseteq \top, U_{\text{Abyte}} \sqsubseteq C_A \times \text{xsd}\#\text{byte}$
Binds:	$U = U_{\text{Abyte}}$
Expects:	<i>InvalidPropertyNameException</i>
Creates:	—
Binds:	$U = U_{\text{fresh}}$

---

PC-OWLAPI-HAS-OBJECTPROPERTY( $o, R$ )

---

Requires:	$\mathcal{O} \vdash o^{\mathcal{I}} \in \bigcup_{i \in 1..n} \{C_i \mid \geq 1R \sqsubseteq C_i\}^{\mathcal{I}}$
Required predecessor:	PC-OWLAPI-DECLARED-INDIVIDUAL( $o$ ), PC-OWLAPI-DECLARED-OBJECTPROPERTY( $R$ )
Creates:	$C_{\text{fresh}_1} \sqsubseteq \top, C_{\text{fresh}_2} \sqsubseteq \top, R_{\text{fresh}_{12}} \sqsubseteq C_{\text{fresh}_1} \times C_{\text{fresh}_2}$
Binds:	$o = o_A, R = R_{\text{fresh}_{12}}$
Expects:	<i>InvalidPropertyDomainException</i>
Creates:	—
Binds:	$o = o_A, R = R_{AB}$

---

PC-OWLAPI-HAS-DATATYPEPROPERTY( $o, U$ )

---

Requires:	$\mathcal{O} \vdash o^{\mathcal{I}} \in \bigcup_{i \in 1..n} \{C_i \mid \geq 1U \sqsubseteq C_i\}^{\mathcal{I}}$
Required predecessor:	PC-OWLAPI-DECLARED-INDIVIDUAL( $o$ ), PC-OWLAPI-DECLARED-DATATYPEPROPERTY( $U$ )
Creates:	$C_{\text{fresh}_1} \sqsubseteq \top, U_{\text{fresh}_{1\text{byte}}} \sqsubseteq C_{\text{fresh}_1} \times \text{xsd}\#\text{byte}$
Binds:	$o = o_A, U = U_{\text{fresh}_{1\text{byte}}}$
Expects:	<i>InvalidPropertyDomainException</i>
Creates:	—
Binds:	$o = o_A, U = U_{\text{Abyte}}$

PC-OWLAPI-IS-RANGE-OBJECT( $o, R$ )

Requires:	$\mathcal{O} \vdash o^{\mathcal{I}} \in \bigcup_{i \in 1..n} \{C_i   \top \sqsubseteq \forall R.C_i\}^{\mathcal{I}}$
Required predecessor:	PC-OWLAPI-HAS-OBJECTPROPERTY( $o, R$ )
Creates:	$C_{\text{fresh}} \sqsubseteq \top, C_{\text{fresh}}(o_{\text{fresh}})$
Binds:	$o = o_{\text{fresh}}, R = R_{\text{AB}}$
Expects:	<i>InvalidPropertyRangeException</i>
Creates:	$C_{\text{B}}(o_{\text{B}})$
Binds:	$o = o_{\text{B}}, R = R_{\text{AB}}$

PC-OWLAPI-IS-RANGE-TYPE( $D, U$ )

Requires:	$D <: \text{range}(U)$
Required predecessor:	PC-OWLAPI-DECLARED-DATATYPEPROPERTY( $U$ )
Creates:	—
Binds:	$D = \text{xsd}\#\text{string}, U = U_{\text{Abyte}}$
Expects:	<i>InvalidPropertyRangeException</i>
Creates:	—
Binds:	$D = \text{xsd}\#\text{byte}, U = U_{\text{Abyte}}$

PC-OWLAPI-IS-OBJECTPROPERTY-VALUE-OF-INDIVIDUAL( $o_1, R, o_2$ )

Requires:	$\mathcal{O} \vdash \langle o_1^{\mathcal{I}}, o_2^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$
Required predecessor:	PC-OWLAPI-DECLARED-INDIVIDUAL( $o$ ), PC-OWLAPI-DECLARED-OBJECTPROPERTY( $R$ )
Creates:	$C_{\text{fresh}} \sqsubseteq \top, C_{\text{fresh}}(o_{\text{fresh}})$
Binds:	$o_1 = o_{\text{A}}, R = R_{\text{AB}}, o_2 = o_{\text{fresh}}$
Expects:	<i>UndeclaredPropertyValueException</i>
Creates:	$C_{\text{B}} \sqsubseteq \top, C_{\text{B}}(o_{\text{B}}), R_{\text{AB}}(o_{\text{A}}, o_{\text{B}})$
Binds:	$o_1 = o_{\text{A}}, R = R_{\text{AB}}, o_2 = o_{\text{B}}$

---

PC-OWLAPI-IS-DATATYPEPROPERTY-VALUE-OF-INDIVIDUAL( $o, U, v$ )

---

Tests:  $\mathcal{O} \vdash \langle o^{\mathcal{I}}, v^{\mathbf{D}} \rangle \in U^{\mathcal{I}}$

---

Required predecessor: PC-OWLAPI-DECLARED-INDIVIDUAL( $o$ ),  
PC-OWLAPI-DECLARED-DATATYPEPROPERTY( $U$ )

---

Creates: —

Binds:  $o = o_A, U = U_{\text{Abyte}}, v = 100$

Expects: *UndeclaredPropertyValueException*

Creates:  $U_{\text{Abyte}}(o_A, 100)$

Binds:  $o = o_A, U = U_{\text{Abyte}}, v = 100$

---

PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE( $D, U$ )

---

Tests:  $\mathcal{O} \vdash D^{\mathbf{D}} \cap \text{range}(U)^{\mathbf{D}} \neq \emptyset$

---

Required predecessor: PC-OWLAPI-DECLARED-DATATYPEPROPERTY( $U$ )

---

Creates: —

Binds:  $D = \text{xsd}\#\text{string}, U = U_{\text{Abyte}}$

Expects: *InvalidDataValueException*

Creates: —

Binds:  $D = \text{xsd}\#\text{byte}, U = U_{\text{Abyte}}$

---

PC-OWLAPI-IS-NATURAL-NUMBER( $n$ )

---

Tests:  $n \in \mathbb{N}$

---

Required predecessor: —

---

Creates: —

Binds:  $n = -1$

Expects: *NotANaturalNumberException*

Creates: —

Binds:  $n = 1$

# APPENDIX G

## The CHIL OWL API

### G.1 The *ITellingABox* Interface

#### **addIndividual(*o*, *C*)**

(HT-OWLAPI-ADDINDIVIDUAL)

$$\begin{array}{l} \text{addIndividual}(o, C) \doteq \\ \{P\}: \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{addIndividual}(o, C) \\ \{R\}: \left\{ (\mathcal{O} \vdash o : C) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

#### **deleteIndividual(*o*)**

Note: This method removes the specified named individual from the ontology by deleting any statements that refer to it, as either statement-subject or statement-object.

(HT-OWLAPI-DELETEINDIVIDUAL)

$$\begin{array}{l} \text{deleteIndividual}(o) \doteq \\ \{P\}: \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{deleteIndividual}(o) \\ \{R\}: \left\{ (\mathcal{O} \not\vdash o \in N_I) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$



**declareSameAs**( $o_1, o_2$ )

(HT-OWLAPI-DECLARESAMEAS)

$$\begin{array}{l} \text{declareSameAs}(o_1, o_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareSameAs}(o_1, o_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash o_1 = o_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**revokeSameAs**( $o_1, o_2$ )

(HT-OWLAPI-REVOKESAMEAS)

$$\begin{array}{l} \text{revokeSameAs}(o_1, o_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{revokeSameAs}(o_1, o_2) \\ \{R\}: \left\{ (\mathcal{O} \not\vdash o_1 = o_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**declareDifferentFrom**( $o_1, o_2$ )

(HT-OWLAPI-DECLAREDIFFERENTFROM)

$$\begin{array}{l} \text{declareDifferentFrom}(o_1, o_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareDifferentFrom}(o_1, o_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash o_1 \neq o_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**revokeDifferentFrom**( $o_1, o_2$ )

(HT-OWLAPI-REVOKEDIFFERENTFROM)

$$\begin{array}{l}
 \text{revokeDifferentFrom}(o_1, o_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{revokeDifferentFrom}(o_1, o_2) \\
 \{R\}: \left\{ (\mathcal{O} \not\vdash o_1 \neq o_2) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

**addObjectPropertyValue**( $o_1, R, o_2$ )

(HT-OWLAPI-ADDOBJECTPROPERTYVALUE)

$$\begin{array}{l}
 \text{addObjectPropertyValue}(o_1, R, o_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addObjectPropertyValue}(o_1, R, o_2) \\
 \{R\}: \left\{ (\mathcal{O} \vdash R(o_1, o_2)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addObjectPropertyValueChecked( $o_1, R, o_2$ )

(HT-OWLAPI-ADDOBJECTPROPERTYVALUECHECKED)

$$\begin{array}{l}
 \text{addObjectPropertyValueChecked}(o_1, R, o_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-HAS-OBJECTPROPERTY}(o_1, R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-RANGE-OBJECT}(o_2, R) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addObjectPropertyValueChecked}(o_1, R, o_2) \\
 \{R\}: \left\{ (\mathcal{O} \vdash R(o_1, o_2)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### deleteObjectPropertyValue( $o_1, R, o_2$ )

(HT-OWLAPI-DELETEOBJECTPROPERTYVALUE)

$$\begin{array}{l}
 \text{deleteObjectPropertyValue}(o_1, R, o_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-OBJECTPROPERTY-VALUE-OF-INDIVIDUAL}(o_1, R, o_2) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{deleteObjectPropertyValue}(o_1, R, o_2) \\
 \{R\}: \left\{ (\mathcal{O} \not\vdash R(o_1, o_2)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### assignObjectPropertyValue( $o_1, R, o_2$ )

(HT-OWLAPI-ASSIGNOBJECTPROPERTYVALUE)

$$\begin{array}{l}
 \text{assignObjectPropertyValue}(o_1, R, o_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{assignObjectPropertyValue}(o_1, R, o_2) \\
 \{R\}: \left\{ \begin{array}{l} (\mathcal{O} \vdash R(o_1, o_2)) \wedge \\ (\mathcal{O} \vdash \forall o. (\langle o_1^{\mathcal{I}}, o^{\mathcal{I}} \rangle \in R^{\mathcal{I}}) \Rightarrow (o = o_2)) \wedge \\ (\mathcal{R} = \emptyset) \end{array} \right\}
 \end{array}$$

### assignObjectPropertyValueChecked( $o_1, R, o_2$ )

(HT-OWLAPI-ASSIGNOBJECTPROPERTYVALUECHECKED)

$$\begin{array}{l}
 \text{assignObjectPropertyValueChecked}(o_1, R, o_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_2), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-HAS-OBJECTPROPERTY}(o_1, R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-RANGE-OBJECT}(o_2, R) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{assignObjectPropertyValueChecked}(o_1, R, o_2) \\
 \{R\}: \left\{ \begin{array}{l} (\mathcal{O} \vdash R(o_1, o_2)) \wedge \\ (\mathcal{O} \vdash \forall o. (\langle o_1^{\mathcal{I}}, o^{\mathcal{I}} \rangle \in R^{\mathcal{I}}) \Rightarrow (o = o_2)) \wedge \\ (\mathcal{R} = \emptyset) \end{array} \right\}
 \end{array}$$

### addDatatypePropertyValue( $o, U, v, D$ )

(HT-OWLAPI-ADDDATATYPEPROPERTYVALUE)

$$\begin{array}{l}
 \text{addDatatypePropertyValue}(o, U, v, D) \doteq \\
 \{P\}: \left\{ \begin{array}{l}
 \eta(\mathcal{O}), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-VALID-DATAVALUE}(v, D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE}(D, U)
 \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addDatatypePropertyValue}(o, U, v, D) \\
 \{R\}: \left\{ (\mathcal{O} \vdash U(o, v)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addDatatypePropertyValueChecked( $o, U, v, D$ )

(HT-OWLAPI-ADDDATATYPEPROPERTYVALUECHECKED)

$$\begin{array}{l}
 \text{addDatatypePropertyValueChecked}(o, U, v, D) \doteq \\
 \{P\}: \left\{ \begin{array}{l}
 \eta(\mathcal{O}), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-VALID-DATAVALUE}(v, D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE}(D, U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-HAS-DATATYPEPROPERTY}(o, U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-RANGE-TYPE}(D, U)
 \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addDatatypePropertyValueChecked}(o, U, v, D) \\
 \{R\}: \left\{ (\mathcal{O} \vdash U(o, v)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

## deleteDatatypePropertyValue( $o, U, v, D$ )

Note: For defined datatype property values this method deletes all values whose types are derived from the same primitive base type like data type  $D$ . For example, datatype property values `"23"^^http://www.w3.org/2001/XMLSchema#integer` and `"23"^^http://www.w3.org/2001/XMLSchema#long` are both deleted for either parameter `"23"^^http://www.w3.org/2001/XMLSchema#integer` and `"23"^^http://www.w3.org/2001/XMLSchema#long`).

An *InvalidDataValueException* is thrown if a property value of the given type does not exist (e.g., `"v"^^D` must not be `"23"^^http://www.w3.org/2001/XMLSchema#string` for datatype property values `"23"^^http://www.w3.org/2001/XMLSchema#integer` and `"23"^^http://www.w3.org/2001/XMLSchema#long`).

(HT-OWLAPI-DELETEDATATYPEPROPERTYVALUE)

$$\begin{array}{l}
 \text{deleteDatatypePropertyValue}(o, U, v, D) \doteq \\
 \left. \begin{array}{l}
 \eta(\mathcal{O}), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-VALID-DATAVALUE}(v, D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE}(D, U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-DATATYPEPROPERTY-VALUE-OF-INDIVIDUAL}(o, U, v)
 \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{deleteDatatypePropertyValue}(o, U, v, D) \\
 \{R\}: \quad \left\{ (\mathcal{O} \not\vdash U(o, v)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

**assignDatatypePropertyValue( $o, U, v, D$ )**

(HT-OWLAPI-ASSIGNDATATYPEPROPERTYVALUE)

$$\begin{array}{l}
 \text{assignDatatypePropertyValue}(o, U, v, D) \doteq \\
 \left. \begin{array}{l}
 \eta(\mathcal{O}), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-VALID-DATAVALUE}(v, D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE}(D, U)
 \end{array} \right\} \{P\}: \\
 \{Q\}: \quad \mathcal{O}, \text{assignDatatypePropertyValue}(o, U, v, D) \\
 \{R\}: \quad \left\{ \begin{array}{l}
 (\mathcal{O} \vdash U(o, v)) \wedge (\mathcal{O} \vdash \forall x. (\langle o^{\mathcal{I}}, x^{\mathbf{D}} \rangle \in U^{\mathcal{I}}) \Rightarrow (x^{\mathbf{D}} = v^{\mathbf{D}})) \wedge \\
 (\mathcal{R} = \emptyset)
 \end{array} \right\}
 \end{array}$$

**assignDatatypePropertyValueChecked( $o, U, v, D$ )**

(HT-OWLAPI-ASSIGNDATATYPEPROPERTYVALUECHECKED)

$$\begin{array}{l}
 \text{assignDatatypePropertyValueChecked}(o, U, v, D) \doteq \\
 \left. \begin{array}{l}
 \eta(\mathcal{O}), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-VALID-DATAVALUE}(v, D), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE}(D, U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-HAS-DATATYPEPROPERTY}(o, U), \\
 \mathcal{O} \vdash \text{PC-OWLAPI-IS-RANGE-TYPE}(D, U)
 \end{array} \right\} \{P\}: \\
 \{Q\}: \quad \mathcal{O}, \text{assignDatatypePropertyValueChecked}(o, U, v, D) \\
 \{R\}: \quad \left\{ \begin{array}{l}
 (\mathcal{O} \vdash U(o, v)) \wedge (\mathcal{O} \vdash \forall x. (\langle o^{\mathcal{I}}, x^{\mathbf{D}} \rangle \in U^{\mathcal{I}}) \Rightarrow (x^{\mathbf{D}} = v^{\mathbf{D}})) \wedge \\
 (\mathcal{R} = \emptyset)
 \end{array} \right\}
 \end{array}$$

## G.2 The *IAskingABox* Interface

### **listSameIndividuals(*o*)**

(HT-OWLAPI-LISTSAMEINDIVIDUALS)

$$\begin{aligned} & \text{listSameIndividuals}(o) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listSameIndividuals}(o) \\ \{R\}: & \left\{ \mathcal{R} = \{x \mid \mathcal{O} \vdash x^{\mathcal{I}} = o^{\mathcal{I}}\} \right\} \end{aligned}$$

### **listDifferentIndividuals(*o*)**

(HT-OWLAPI-LISTDIFFERENTINDIVIDUALS)

$$\begin{aligned} & \text{listDifferentIndividuals}(o) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listDifferentIndividuals}(o) \\ \{R\}: & \left\{ \mathcal{R} = \{x \mid \mathcal{O} \vdash x^{\mathcal{I}} \neq o^{\mathcal{I}}\} \right\} \end{aligned}$$

### **listIndividualsOfClass(*C*)**

(HT-OWLAPI-LISTINDIVIDUALSOFCCLASS)

$$\begin{aligned} & \text{listIndividualsOfClass}(C) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listIndividualsOfClass}(C) \\ \{R\}: & \left\{ \mathcal{R} = \{o \mid \mathcal{O} \vdash o^{\mathcal{I}} \in C^{\mathcal{I}}\} \right\} \end{aligned}$$



### **getCardinalityOfClass( $C$ )**

(HT-OWLAPI-GETCARDINALITYOFCLASS)

$$\begin{aligned} & \text{getCardinalityOfClass}(C) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{getCardinalityOfClass}(C) \\ \{R\}: & \quad \left\{ \mathcal{R} = |\{o \mid \mathcal{O} \vdash o^{\mathcal{I}} \in C^{\mathcal{I}}\}| \right\} \end{aligned}$$

### **listRDFTypesOfIndividual( $o$ )**

Note: For declared individuals, the list of RDF types returned by this method always includes the top level concept <http://www.w3.org/2002/07/owl#Thing>; the list does not contain <http://www.w3.org/2000/01/rdf-schema#Resource>.

(HT-OWLAPI-LISTRDFTYPESOFINDIVIDUAL)

$$\begin{aligned} & \text{listRDFTypesOfIndividual}(o) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listRDFTypesOfIndividual}(o) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{C \mid \mathcal{O} \vdash o^{\mathcal{I}} \in C^{\mathcal{I}}\} \right\} \end{aligned}$$

### **listDirectRDFTypesOfIndividual( $o$ )**

(HT-OWLAPI-LISTDIRECTRDFTYPESOFINDIVIDUAL)

$$\begin{aligned} & \text{listDirectRDFTypesOfIndividual}(o) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDirectRDFTypesOfIndividual}(o) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{C \mid \mathcal{O} \vdash (o^{\mathcal{I}} \in C^{\mathcal{I}}) \wedge \nexists E.((\mathcal{O} \vdash E^{\mathcal{I}} \subseteq C^{\mathcal{I}}) \wedge (\mathcal{O} \vdash o^{\mathcal{I}} \in E^{\mathcal{I}}))\} \right\} \end{aligned}$$

### listObjectPropertiesOfIndividual( $o$ )

(HT-OWLAPI-LISTOBJECTPROPERTIESOFINDIVIDUAL)

$$\begin{aligned} & \text{listObjectPropertiesOfIndividual}(o) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listObjectPropertiesOfIndividual}(o) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{R \mid \mathcal{O} \vdash (\geq 1R \sqsubseteq C) \wedge \mathcal{O} \vdash (o^{\mathcal{I}} \in C^{\mathcal{I}})\} \right\} \end{aligned}$$

### listDatatypePropertiesOfIndividual( $o$ )

(HT-OWLAPI-LISTDATATYPEPROPERTIESOFINDIVIDUAL)

$$\begin{aligned} & \text{listDatatypePropertiesOfIndividual}(o) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDatatypePropertiesOfIndividual}(o) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{U \mid \mathcal{O} \vdash (\geq 1U \sqsubseteq C) \wedge \mathcal{O} \vdash (o^{\mathcal{I}} \in C^{\mathcal{I}})\} \right\} \end{aligned}$$

### listObjectPropertyValuesOfIndividual( $o, R$ )

(HT-OWLAPI-LISTOBJECTPROPERTYVALUESOFINDIVIDUAL)

$$\begin{aligned} & \text{listObjectPropertyValuesOfIndividual}(o, R) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listObjectPropertyValuesOfIndividual}(o, R) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{x \mid \mathcal{O} \vdash \langle o^{\mathcal{I}}, x^{\mathcal{I}} \rangle \in R^{\mathcal{I}}\} \right\} \end{aligned}$$

### getCardinalityOfObjectPropertyValuesOfIndividual( $o, R$ )

(HT-OWLAPI-GETCARDINALITYOFOBJECTPROPERTYVALUESOFINDIVIDUAL)

$$\begin{aligned} & \text{getCardinalityOfObjectPropertyValuesOfIndividual}(o, R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{getCardinalityOfObjectPropertyValuesOfIndividual}(o, R) \\ \{R\}: & \left\{ \mathcal{R} = |\{x \mid \mathcal{O} \vdash \langle o^{\mathcal{I}}, x^{\mathcal{I}} \rangle \in R^{\mathcal{I}}\}| \right\} \end{aligned}$$

### listDatatypePropertyValuesOfIndividual( $o, U$ )

(HT-OWLAPI-LISTDATATYPEPROPERTYVALUESOFINDIVIDUAL)

$$\begin{aligned} & \text{listDatatypePropertyValuesOfIndividual}(o, U) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listDatatypePropertyValuesOfIndividual}(o, U) \\ \{R\}: & \left\{ \mathcal{R} = \{x \mid \mathcal{O} \vdash \langle o^{\mathcal{I}}, x^{\mathbf{D}} \rangle \in U^{\mathcal{I}}\} \right\} \end{aligned}$$

### getCardinalityOfDatatypePropertyValuesOfIndividual( $o, U$ )

(HT-OWLAPI-GETCARDINALITYOFDATATYPEPROPERTYVALUESOFINDIVIDUAL)

$$\begin{aligned} & \text{getCardinalityOfDatatypePropertyValuesOfIndividual}(o, U) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{getCardinalityOfDatatypePropertyValuesOfIndividual}(o, U) \\ \{R\}: & \left\{ \mathcal{R} = |\{x \mid \mathcal{O} \vdash \langle o^{\mathcal{I}}, x^{\mathbf{D}} \rangle \in U^{\mathcal{I}}\}| \right\} \end{aligned}$$

### G.3 The *ITellingTBox* Interface

#### **declareClass**( $C$ )

(HT-OWLAPI-DECLARECLASS)

$$\begin{array}{l} \text{declareClass}(A) \doteq \\ \{P\}: \quad \left\{ \eta(\mathcal{O}) \right\} \\ \{Q\}: \quad \mathcal{O}, \text{declareClass}(C) \\ \{R\}: \quad \left\{ (\mathcal{O} \vdash C \in N_C) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

#### **removeClass**( $C$ )

Note: This method removes the specified named concept description from the ontology by deleting any statements that refer to it, as either statement-subject or statement-object.

(HT-OWLAPI-REMOVECLASS)

$$\begin{array}{l} \text{removeClass}(C) \doteq \\ \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: \quad \mathcal{O}, \text{removeClass}(C) \\ \{R\}: \quad \left\{ (\mathcal{O} \not\vdash C \in N_C) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

#### **declareEquivalentClass**( $C_1, C_2$ )

(HT-OWLAPI-DECLAREEQUIVALENTCLASS)

$$\begin{array}{l} \text{declareEquivalentClass}(C_1, C_2) \doteq \\ \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: \quad \mathcal{O}, \text{declareEquivalentClass}(C_1, C_2) \\ \{R\}: \quad \left\{ (\mathcal{O} \vdash C_1 \equiv C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**declareDisjointClass**( $C_1, C_2$ )

(HT-OWLAPI-DECLAREDISJOINTCLASS)

$$\begin{array}{l} \text{declareDisjointClass}(C_1, C_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareDisjointClass}(C_1, C_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash C_1 \sqsubseteq \neg C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**declareSubClass**( $C_1, C_2$ )

(HT-OWLAPI-DECLARESUBCLASS)

$$\begin{array}{l} \text{declareSubClass}(C_1, C_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareSubClass}(C_1, C_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash C_1 \sqsubseteq C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**declareSuperClass**( $C_1, C_2$ )

(HT-OWLAPI-DECLARESUPERCLASS)

$$\begin{array}{l} \text{declareSuperClass}(C_1, C_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareSuperClass}(C_1, C_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash C_1 \supseteq C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

**declareIntersectionClass**( $C, \{C_i\}^{i \in 1..n}$ )

(HT-OWLAPI-DECLAREINTERSECTIONCLASS)

$$\begin{aligned} & \text{declareIntersectionClass}(C, \{C_i\}^{i \in 1..n}) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_i)^{i \in 1..n} \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareIntersectionClass}(C, \{C_i\}^{i \in 1..n}) \\ \{R\}: & \left\{ (\mathcal{O} \vdash C \equiv C_1 \sqcap \dots \sqcap C_n) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

**declareUnionClass**( $C, \{C_i\}^{i \in 1..n}$ )

(HT-OWLAPI-DECLAREUNIONCLASS)

$$\begin{aligned} & \text{declareUnionClass}(C, \{C_i\}^{i \in 1..n}) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_i)^{i \in 1..n} \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareUnionClass}(C, \{C_i\}^{i \in 1..n}) \\ \{R\}: & \left\{ (\mathcal{O} \vdash C \equiv C_1 \sqcup \dots \sqcup C_n) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

**declareComplementClass**( $C_1, C_2$ )

(HT-OWLAPI-DECLARECOMPLEMENTCLASS)

$$\begin{aligned} & \text{declareComplementClass}(C_1, C_2) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareComplementClass}(C_1, C_2) \\ \{R\}: & \left\{ (\mathcal{O} \vdash C_1 \equiv \neg C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

**declareEnumeratedClass**( $C, \{o_i\}^{i \in 1..n}$ )

(HT-OWLAPI-DECLAREENUMERATEDCLASS)

$$\begin{aligned} & \text{declareEnumeratedClass}(C, \{o_i\}^{i \in 1..n}) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o_i)^{i \in 1..n} \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareEnumeratedClass}(C, \{o_i\}^{i \in 1..n}) \\ \{R\}: & \left\{ (\mathcal{O} \vdash C \equiv \{o_1, \dots, o_n\}) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

**declareObjectProperty**( $C_1, R, C_2$ )

(HT-OWLAPI-DECLAREOBJECTPROPERTY)

$$\begin{aligned} & \text{declareObjectProperty}(C_1, R, C_2) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareObjectProperty}(C_1, R, C_2) \\ \{R\}: & \left\{ (\mathcal{O} \vdash (\geq 1R) \sqsubseteq C_1) \wedge (\mathcal{O} \vdash \top \sqsubseteq \forall R.C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

**addObjectPropertyDomain**( $C, R$ )

(HT-OWLAPI-ADDOBJECTPROPERTYDOMAIN)

$$\begin{aligned} & \text{addObjectPropertyDomain}(C, R) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{addObjectPropertyDomain}(C, R) \\ \{R\}: & \left\{ (\mathcal{O} \vdash (\geq 1R) \sqsubseteq C) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

### **addObjectPropertyRange( $R, C$ )**

(HT-OWLAPI-ADDOBJECTPROPERTYRANGE)

$$\begin{aligned} & \text{addObjectPropertyRange}(R, C) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{addObjectPropertyRange}(R, C) \\ \{R\}: & \left\{ (\mathcal{O} \vdash \top \sqsubseteq \forall R.C_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

### **declareAsFunctionalObjectProperty( $R$ )**

(HT-OWLAPI-DECLAREASFUNCTIONALOBJECTPROPERTY)

$$\begin{aligned} & \text{declareAsFunctionalObjectProperty}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareAsFunctionalObjectProperty}(R) \\ \{R\}: & \left\{ (\mathcal{O} \vdash \top \sqsubseteq (\leq 1R)) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

### **declareAsSymmetricObjectProperty( $R$ )**

(HT-OWLAPI-DECLAREASSYMMETRICOBJECTPROPERTY)

$$\begin{aligned} & \text{declareAsSymmetricObjectProperty}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareAsSymmetricObjectProperty}(R) \\ \{R\}: & \left\{ (\mathcal{O} \vdash R \sqsubseteq R^-) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$



### **declareAsTransitiveObjectProperty( $R$ )**

(HT-OWLAPI-DECLAREASTRANSITIVEOBJECTPROPERTY)

$$\begin{array}{l}
 \text{declareAsTransitiveObjectProperty}(R) \doteq \\
 \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{declareAsTransitiveObjectProperty}(R) \\
 \{R\}: \quad \left\{ (\mathcal{O} \vdash \mathbf{Trans}(R)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### **declareAsInverseFunctionalObjectProperty( $R$ )**

(HT-OWLAPI-DECLAREASINVERSEFUNCTIONALOBJECTPROPERTY)

$$\begin{array}{l}
 \text{declareAsInverseFunctionalObjectProperty}(R) \doteq \\
 \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{declareAsInverseFunctionalObjectProperty}(R) \\
 \{R\}: \quad \left\{ (\mathcal{O} \vdash \top \sqsubseteq (\leq 1R^-)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### **declareAsInverseObjectProperties( $R_1, R_2$ )**

(HT-OWLAPI-DECLAREASINVERSEOBJECTPROPERTIES)

$$\begin{array}{l}
 \text{declareAsInverseObjectProperties}(R_1, R_2) \doteq \\
 \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_2) \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{declareAsInverseObjectProperties}(R_1, R_2) \\
 \{R\}: \quad \left\{ (\mathcal{O} \vdash R_1 \sqsubseteq R_2^-) \wedge (\mathcal{O} \vdash R_2 \sqsubseteq R_1^-) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### **declareDatatypeProperty( $C, U, D$ )**

(HT-OWLAPI-DECLAREDATATYPEPROPERTY)

$$\begin{aligned} & \text{declareDatatypeProperty}(C, U, D) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareDatatypeProperty}(C, U, D) \\ \{R\}: & \left\{ (\mathcal{O} \vdash (\geq 1U) \sqsubseteq C) \wedge (\mathcal{O} \vdash \top \sqsubseteq \forall U.D) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

### **declareAsFunctionalDatatypeProperty( $U$ )**

(HT-OWLAPI-DECLAREASFUNCTIONALDATATYPEPROPERTY)

$$\begin{aligned} & \text{declareAsFunctionalDatatypeProperty}(U) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareAsFunctionalDatatypeProperty}(U) \\ \{R\}: & \left\{ (\mathcal{O} \vdash \top \sqsubseteq (\leq 1U)) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

### **declareSubObjectProperty( $R_1, R_2$ )**

(HT-OWLAPI-DECLARESUBOBJECTPROPERTY)

$$\begin{aligned} & \text{declareSubObjectProperty}(R_1, R_2) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_2) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{declareSubObjectProperty}(R_1, R_2) \\ \{R\}: & \left\{ (\mathcal{O} \vdash R_1 \sqsubseteq R_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{aligned}$$

### **declareEquivalentObjectProperty( $R_1, R_2$ )**

(HT-OWLAPI-DECLAREEQUIVALENTOBJECTPROPERTY)

$$\begin{array}{l} \text{declareEquivalentObjectProperty}(R_1, R_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareEquivalentObjectProperty}(R_1, R_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash R_1 \equiv R_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

### **declareSuperObjectProperty( $R_1, R_2$ )**

(HT-OWLAPI-DECLARESUPEROBJECTPROPERTY)

$$\begin{array}{l} \text{declareSuperObjectProperty}(R_1, R_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareSuperObjectProperty}(R_1, R_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash R_1 \sqsupseteq R_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

### **declareSubDatatypeProperty( $U_1, U_2$ )**

(HT-OWLAPI-DECLARESUBDATATYPEPROPERTY)

$$\begin{array}{l} \text{declareSubDatatypeProperty}(U_1, U_2) \doteq \\ \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U_2) \end{array} \right\} \\ \{Q\}: \mathcal{O}, \text{declareSubDatatypeProperty}(U_1, U_2) \\ \{R\}: \left\{ (\mathcal{O} \vdash U_1 \sqsubseteq U_2) \wedge (\mathcal{R} = \emptyset) \right\} \end{array}$$

### **declareEquivalentDatatypeProperty( $U_1, U_2$ )**

(HT-OWLAPI-DECLAREEQUIVALENTDATATYPEPROPERTY)

$$\begin{array}{l}
 \text{declareEquivalentDatatypeProperty}(U_1, U_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U_2) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{declareEquivalentDatatypeProperty}(U_1, U_2) \\
 \{R\}: \left\{ (\mathcal{O} \vdash U_1 \equiv U_2) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### **declareSuperDatatypeProperty( $U_1, U_2$ )**

(HT-OWLAPI-DECLARESUPERDATATYPEPROPERTY)

$$\begin{array}{l}
 \text{declareSuperDatatypeProperty}(U_1, U_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U_2) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{declareSuperDatatypeProperty}(U_1, U_2) \\
 \{R\}: \left\{ (\mathcal{O} \vdash U_1 \sqsupseteq U_2) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### **addObjectPropertyHasValueRestriction( $C, R, o$ )**

(HT-OWLAPI-ADDOBJECTPROPERTYHASVALUERESTRICTION)

$$\begin{array}{l}
 \text{addObjectPropertyHasValueRestriction}(C, R, o) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-INDIVIDUAL}(o) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addObjectPropertyHasValueRestriction}(C, R, o) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\exists R.\{o\})) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

**addSomeValuesFromConceptRestriction( $C_1, R, C_2$ )**

(HT-OWLAPI-ADDSOMEVALUESFROMCONCEPTRESTRICTION)

$$\begin{array}{l}
 \text{addSomeValuesFromConceptRestriction}(C_1, R, C_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addSomeValuesFromConceptRestriction}(C_1, R, C_2) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C_1 \sqsubseteq (\exists R.C_2)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

**addSomeValuesFromDatatypeRestriction( $C, U, D$ )**

(HT-OWLAPI-ADDSOMEVALUESFROMDATATYPERESTRICTION)

$$\begin{array}{l}
 \text{addSomeValuesFromDatatypeRestriction}(C, U, D) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addSomeValuesFromDatatypeRestriction}(C, U, D) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\exists U.D)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

**addAllValuesFromConceptRestriction( $C_1, R, C_2$ )**

(HT-OWLAPI-ADDALLVALUESFROMCONCEPTRESTRICTION)

$$\begin{array}{l}
 \text{addAllValuesFromConceptRestriction}(C_1, R, C_2) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_1), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C_2) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addAllValuesFromConceptRestriction}(C_1, R, C_2) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C_1 \sqsubseteq (\forall R.C_2)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

**addAllValuesFromDatatypeRestriction( $C, U, D$ )**

(HT-OWLAPI-ADDALLVALUESFROMDATATYPERESTRICTION)

$$\begin{array}{l}
 \text{addAllValuesFromDatatypeRestriction}(C, U, D) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPE}(D) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addAllValuesFromDatatypeRestriction}(C, U, D) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\forall U.D)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addObjectPropertyMinCardinalityRestriction( $C, R, n$ )

(HT-OWLAPI-ADDOBJECTPROPERTYMINCARDINALITYRESTRICTION)

$$\begin{array}{l}
 \text{addObjectPropertyMinCardinalityRestriction}(C, R, n) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-NATURAL-NUMBER}(n) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addObjectPropertyMinCardinalityRestriction}(C, R, n) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\geq nR)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addObjectPropertyCardinalityRestriction( $C, R, n$ )

(HT-OWLAPI-ADDOBJECTPROPERTYCARDINALITYRESTRICTION)

$$\begin{array}{l}
 \text{addObjectPropertyCardinalityRestriction}(C, R, n) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-NATURAL-NUMBER}(n) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addObjectPropertyCardinalityRestriction}(C, R, n) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (=nR)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addObjectPropertyMaxCardinalityRestriction( $C, R, n$ )

(HT-OWLAPI-ADDOBJECTPROPERTYMAXCARDINALITYRESTRICTION)

$$\begin{array}{l}
 \text{addObjectPropertyMaxCardinalityRestriction}(C, R, n) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-NATURAL-NUMBER}(n) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addObjectPropertyMaxCardinalityRestriction}(C, R, n) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\leq nR)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addDatatypePropertyMinCardinalityRestriction( $C, U, n$ )

(HT-OWLAPI-ADDDATATYPEPROPERTYMINCARDINALITYRESTRICTION)

$$\begin{array}{l}
 \text{addDatatypePropertyMinCardinalityRestriction}(C, U, n) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-NATURAL-NUMBER}(n) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addDatatypePropertyMinCardinalityRestriction}(C, U, n) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\geq nU)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$



### addDatatypePropertyCardinalityRestriction( $C, U, n$ )

(HT-OWLAPI-ADDDATATYPEPROPERTYCARDINALITYRESTRICTION)

$$\begin{array}{l}
 \text{addDatatypePropertyCardinalityRestriction}(C, U, n) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-NATURAL-NUMBER}(n) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addDatatypePropertyCardinalityRestriction}(C, U, n) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (=nU)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

### addDatatypePropertyMaxCardinalityRestriction( $C, U, n$ )

(HT-OWLAPI-ADDDATATYPEPROPERTYMAXCARDINALITYRESTRICTION)

$$\begin{array}{l}
 \text{addDatatypePropertyMaxCardinalityRestriction}(C, U, n) \doteq \\
 \{P\}: \left\{ \begin{array}{l} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C), \\ \mathcal{O} \vdash \text{PC-OWLAPI-IS-NATURAL-NUMBER}(n) \end{array} \right\} \\
 \{Q\}: \mathcal{O}, \text{addDatatypePropertyMaxCardinalityRestriction}(C, U, n) \\
 \{R\}: \left\{ (\mathcal{O} \vdash C \sqsubseteq (\leq nU)) \wedge (\mathcal{R} = \emptyset) \right\}
 \end{array}$$

## G.4 The *IAskingTBox* Interface

### **listClasses()**

(HT-OWLAPI-LISTCLASSES)

$$\begin{array}{l}
 \text{listClasses()} \doteq \\
 \{P\}: \quad \left\{ \eta(\mathcal{O}) \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{listClasses}() \\
 \{R\}: \quad \left\{ \mathcal{R} = \{C \mid \mathcal{O} \vdash C \in N_C\} \right\}
 \end{array}$$

### **listSubClasses(*C*)**

(HT-OWLAPI-LISTSUBCLASSES)

$$\begin{array}{l}
 \text{listSubClasses}(C) \doteq \\
 \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{listSubClasses}(C) \\
 \{R\}: \quad \left\{ \mathcal{R} = \{E \mid \mathcal{O} \vdash E \sqsubseteq C\} \right\}
 \end{array}$$

### **listDirectSubClasses(*C*)**

(HT-OWLAPI-LISTDIRECTSUBCLASSES)

$$\begin{array}{l}
 \text{listDirectSubClasses}(C) \doteq \\
 \{P\}: \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\
 \{Q\}: \quad \mathcal{O}, \text{listDirectSubClasses}(C) \\
 \{R\}: \quad \left\{ \begin{array}{c} \mathcal{R} = \{E \mid \mathcal{O} \vdash E \sqsubseteq C \wedge \forall F. ((\mathcal{O} \vdash E \sqsubseteq F \wedge \mathcal{O} \vdash F \sqsubseteq C) \Rightarrow \\ \mathcal{O} \vdash (E \equiv F \vee F \equiv C))\} \end{array} \right\}
 \end{array}$$

### **listEquivalentClasses( $C$ )**

(HT-OWLAPI-LISTEQUIVALENTCLASSES)

$$\begin{aligned} & \text{listEquivalentClasses}(C) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listEquivalentClasses}(C) \\ \{R\}: & \left\{ \mathcal{R} = \{E \mid \mathcal{O} \vdash E \equiv C\} \right\} \end{aligned}$$

### **listDisjointClasses( $C$ )**

(HT-OWLAPI-LISTDISJOINTCLASSES)

$$\begin{aligned} & \text{listDisjointClasses}(C) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listDisjointClasses}(C) \\ \{R\}: & \left\{ \mathcal{R} = \{E \mid \mathcal{O} \vdash E \sqsubseteq \neg C\} \right\} \end{aligned}$$

### **listSuperClasses( $C$ )**

(HT-OWLAPI-LISTSUPERCLASSES)

$$\begin{aligned} & \text{listSuperClasses}(C) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listSuperClasses}(C) \\ \{R\}: & \left\{ \mathcal{R} = \{E \mid \mathcal{O} \vdash E \sqsupseteq C\} \right\} \end{aligned}$$

### **listDirectSuperClasses( $C$ )**

(HT-OWLAPI-LISTDIRECTSUPERCLASSES)

$$\begin{aligned} & \text{listDirectSubClasses}(C) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDirectSubClasses}(C) \\ \{R\}: & \quad \left\{ \begin{array}{c} \mathcal{R} = \{E \mid \mathcal{O} \vdash E \sqsupseteq C \wedge \forall F. ((\mathcal{O} \vdash E \sqsupseteq F \wedge \mathcal{O} \vdash F \sqsupseteq C) \Rightarrow \\ \mathcal{O} \vdash (F \equiv E \vee F \equiv C))\} \end{array} \right\} \end{aligned}$$

### **listDatatypes()**

(HT-OWLAPI-LISTDATATYPES)

$$\begin{aligned} & \text{listDatatypes}() \doteq \\ \{P\}: & \quad \left\{ \eta(\mathcal{O}) \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDatatypes}() \\ \{R\}: & \quad \left\{ \mathcal{R} = \{D \mid \mathcal{O} \vdash D \in N_D\} \right\} \end{aligned}$$

### **listObjectProperties()**

(HT-OWLAPI-LISTOBJECTPROPERTIES)

$$\begin{aligned} & \text{listObjectProperties}() \doteq \\ \{P\}: & \quad \left\{ \eta(\mathcal{O}) \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listObjectProperties}() \\ \{R\}: & \quad \left\{ \mathcal{R} = \{R \mid \mathcal{O} \vdash R \in N_R\} \right\} \end{aligned}$$

### **listObjectPropertiesOfClass( $C$ )**

(HT-OWLAPI-LISTOBJECTPROPERTIESOFCLASS)

$$\begin{aligned} & \text{listObjectPropertiesOfClass}(C) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listObjectPropertiesOfClass}(C) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{R \mid \mathcal{O} \vdash (\geq 1R) \sqsubseteq C\} \right\} \end{aligned}$$

### **listDatatypeProperties()**

(HT-OWLAPI-LISTDATATYPEPROPERTIES)

$$\begin{aligned} & \text{listDatatypeProperties}() \doteq \\ \{P\}: & \quad \left\{ \eta(\mathcal{O}) \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDatatypeProperties}() \\ \{R\}: & \quad \left\{ \mathcal{R} = \{U \mid \mathcal{O} \vdash U \in N_U\} \right\} \end{aligned}$$

### **listDatatypePropertiesOfClass( $C$ )**

(HT-OWLAPI-LISTDATATYPEPROPERTIESOFCLASS)

$$\begin{aligned} & \text{listDatatypePropertiesOfClass}(C) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-CONCEPT}(C) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDatatypePropertiesOfClass}(C) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{U \mid \mathcal{O} \vdash (\geq 1U) \sqsubseteq C\} \right\} \end{aligned}$$

### **listSubObjectProperties( $R$ )**

(HT-OWLAPI-LISTSUBOBJECTPROPERTIES)

$$\begin{aligned} & \text{listSubObjectProperties}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listSubObjectProperties}(R) \\ \{R\}: & \left\{ \mathcal{R} = \{S \mid \mathcal{O} \vdash S \sqsubseteq R\} \right\} \end{aligned}$$

### **listEquivalentObjectProperties( $R$ )**

(HT-OWLAPI-LISTEQUIVALENTOBJECTPROPERTIES)

$$\begin{aligned} & \text{listEquivalentObjectProperties}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listEquivalentObjectProperties}(R) \\ \{R\}: & \left\{ \mathcal{R} = \{S \mid \mathcal{O} \vdash S \equiv R\} \right\} \end{aligned}$$

### **listInverseObjectProperties( $R$ )**

(HT-OWLAPI-LISTINVERSEOBJECTPROPERTIES)

$$\begin{aligned} & \text{listInverseObjectProperties}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listInverseObjectProperties}(R) \\ \{R\}: & \left\{ \mathcal{R} = \{S \mid \mathcal{O} \vdash S \equiv R^{-}\} \right\} \end{aligned}$$

### **listSubDatatypeProperties( $U$ )**

(HT-OWLAPI-LISTSUBDATATYPEPROPERTIES)

$$\begin{aligned} & \text{listSubDatatypeProperties}(U) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listSubDatatypeProperties}(U) \\ \{R\}: & \left\{ \mathcal{R} = \{S \mid \mathcal{O} \vdash S \sqsubseteq U\} \right\} \end{aligned}$$

### **listEquivalentDatatypeProperties( $U$ )**

(HT-OWLAPI-LISTEQUIVALENTDATATYPEPROPERTIES)

$$\begin{aligned} & \text{listEquivalentDatatypeProperties}(U) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listEquivalentDatatypeProperties}(U) \\ \{R\}: & \left\{ \mathcal{R} = \{S \mid \mathcal{O} \vdash S \equiv U\} \right\} \end{aligned}$$

### **listDomainOfObjectProperty( $R$ )**

Note: This list always contains the top level concept <http://www.w3.org/2002/07/owl-#Thing>.

(HT-OWLAPI-LISTDOMAINOFOBJECTPROPERTY)

$$\begin{aligned} & \text{listDomainOfObjectProperty}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{listDomainOfObjectProperty}(R) \\ \{R\}: & \left\{ \mathcal{R} = \{C \mid \mathcal{O} \vdash (\geq 1R) \sqsubseteq C\} \right\} \end{aligned}$$

### **listRangeOfObjectProperty( $R$ )**

Note: This list always contains the top level concept <http://www.w3.org/2002/07/owl-#Thing>.

$$\begin{aligned} & \text{(HT-OWLAPI-LISTRANGEOFOBJECTPROPERTY)} \\ & \text{listRangeOfObjectProperty}(R) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listRangeOfObjectProperty}(R) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{C \mid \mathcal{O} \vdash \top \sqsubseteq \forall R.C\} \right\} \end{aligned}$$

### **listDomainOfDatatypeProperty( $U$ )**

Note: This list always contains the top level concept <http://www.w3.org/2002/07/owl-#Thing>.

$$\begin{aligned} & \text{(HT-OWLAPI-LISTDOMAINOFDATATYPEPROPERTY)} \\ & \text{listDomainOfDatatypeProperty}(U) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listDomainOfDatatypeProperty}(U) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{C \mid \mathcal{O} \vdash (\geq 1U) \sqsubseteq C\} \right\} \end{aligned}$$

### **listRangeOfDatatypeProperty( $U$ )**

$$\begin{aligned} & \text{(HT-OWLAPI-LISTRANGEOFDATATYPEPROPERTY)} \\ & \text{listRangeOfDatatypeProperty}(U) \doteq \\ \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\ \{Q\}: & \quad \mathcal{O}, \text{listRangeOfDatatypeProperty}(U) \\ \{R\}: & \quad \left\{ \mathcal{R} = \{D \mid \top \sqsubseteq \forall U.D\} \right\} \end{aligned}$$



### **isFunctionalObjectProperty( $R$ )**

(HT-OWLAPI-ISFUNCTIONALOBJECTPROPERTY)

$$\begin{aligned} & \text{isFunctionalObjectProperty}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{isFunctionalObjectProperty}(R) \\ \{R\}: & \left\{ \mathcal{R} = \begin{cases} \text{true} & ; \text{ if } \mathcal{O} \vdash \top \sqsubseteq (\leq 1R) \\ \text{false} & ; \text{ otherwise} \end{cases} \right\} \end{aligned}$$

### **isInverseFunctionalObjectProperty( $R$ )**

(HT-OWLAPI-ISINVERSEFUNCTIONALOBJECTPROPERTY)

$$\begin{aligned} & \text{isInverseFunctionalObjectProperty}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{isInverseFunctionalObjectProperty}(R) \\ \{R\}: & \left\{ \mathcal{R} = \begin{cases} \text{true} & ; \text{ if } \mathcal{O} \vdash \top \sqsubseteq (\leq 1R^-) \\ \text{false} & ; \text{ otherwise} \end{cases} \right\} \end{aligned}$$

### **isSymmetricObjectProperty( $R$ )**

(HT-OWLAPI-ISSYMMETRICOBJECTPROPERTY)

$$\begin{aligned} & \text{isSymmetricObjectProperty}(R) \doteq \\ \{P\}: & \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\ \{Q\}: & \mathcal{O}, \text{isSymmetricObjectProperty}(R) \\ \{R\}: & \left\{ \mathcal{R} = \begin{cases} \text{true} & ; \text{ if } \mathcal{O} \vdash R \sqsubseteq R^- \\ \text{false} & ; \text{ otherwise} \end{cases} \right\} \end{aligned}$$

### isTransitiveObjectProperty( $R$ )

(HT-OWLAPI-ISTRANSITIVEOBJECTPROPERTY)

$$\begin{aligned}
 & \text{isTransitiveObjectProperty}(R) \doteq \\
 \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-OBJECTPROPERTY}(R) \end{array} \right\} \\
 \{Q\}: & \quad \mathcal{O}, \text{isTransitiveObjectProperty}(R) \\
 \{R\}: & \quad \mathcal{R} = \left\{ \begin{array}{ll} \text{true} & ; \text{ if } \mathcal{O} \vdash \mathbf{Trans}(R) \\ \text{false} & ; \text{ otherwise} \end{array} \right\}
 \end{aligned}$$

### isFunctionalDatatypeProperty( $U$ )

(HT-OWLAPI-ISFUNCTIONALDATATYPEPROPERTY)

$$\begin{aligned}
 & \text{isFunctionalDatatypeProperty}(U) \doteq \\
 \{P\}: & \quad \left\{ \begin{array}{c} \eta(\mathcal{O}), \\ \mathcal{O} \vdash \text{PC-OWLAPI-DECLARED-DATATYPEPROPERTY}(U) \end{array} \right\} \\
 \{Q\}: & \quad \mathcal{O}, \text{isFunctionalDatatypeProperty}(U) \\
 \{R\}: & \quad \mathcal{R} = \left\{ \begin{array}{ll} \text{true} & ; \text{ if } \mathcal{O} \vdash \top \sqsubseteq (\leq 1U) \\ \text{false} & ; \text{ otherwise} \end{array} \right\}
 \end{aligned}$$

## APPENDIX H

### Mapping of the CHIL OWL API

The following three sections present mappings from interfaces of the CHIL OWL API to auxiliary methods and properties of ontological individuals, roles, and static concept references in Zhi# programs.

The first row of each of the following tables contains conventional C# method invocations on the CHIL OWL API (the *Zhimantic.OWL.API.IOWLAPI* host object is omitted for brevity), where concept, role, and individual names are given as quoted string literals. The second row of each table contains corresponding Zhi# code, where the *IOWLAPI* host object is substituted by ontology elements. Also, string literals denoting concept, role, and individual *names* will partially be replaced by concept, role, and individual *objects* (qualifying namespace aliases are omitted for brevity). Thus, defined method preconditions that are related to the introduced ontology elements can now be statically checked, which eliminates dynamic checks and possible exceptions at runtime.

The functionality of CHIL OWL API methods marked with an asterisk has already been implemented in the current version of the OWL DL plug-in for the Zhi# compiler (see Subsection 6.1.3). Note that in the following sections the naming of the suggested auxiliary methods and properties may differ from the actual implementation. Also, the functionality of particular CHIL OWL API methods may be provided in Zhi# by means of object-oriented member access and operators instead of auxiliary methods and properties. Also recall that the TBox of managed knowledge bases is assumed immutable! Furthermore, note that preconditions marked with an asterisk can only be checked at compile time if modifications of the ABox are monotonic (e.g., individuals must not be deleted).

## H.1 The *ITellingABox* Interface

CHIL OWL API:	<i>addIndividual("o", "C")*</i>
Zhi#code:	<i>C.New("o")</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>deleteIndividual("o")*</i>
Zhi#code:	<i>o.Delete()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> ) <sup>(*)</sup>

CHIL OWL API:	<i>declareSameAs("o<sub>1</sub>", "o<sub>2</sub>")</i>
Zhi#code:	<i>o<sub>1</sub>.SameAs(o<sub>2</sub>)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>1</sub></i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>2</sub></i> )*

CHIL OWL API:	<i>revokeSameAs("o<sub>1</sub>", "o<sub>2</sub>")</i>
Zhi#code:	<i>o<sub>1</sub>.RevokeSameAs(o<sub>2</sub>)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>1</sub></i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>2</sub></i> )*

CHIL OWL API:	<i>declareDifferentFrom("o<sub>1</sub>", "o<sub>2</sub>")</i>
Zhi#code:	<i>o<sub>1</sub>.DifferentFrom(o<sub>2</sub>)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>1</sub></i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>2</sub></i> )*

CHIL OWL API:	<i>revokeDifferentFrom("o<sub>1</sub>", "o<sub>2</sub>")</i>
Zhi#code:	<i>o<sub>1</sub>.RevokeDifferentFrom(o<sub>2</sub>)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>1</sub></i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o<sub>2</sub></i> )*

CHIL OWL API:	<i>addObjectPropertyValue("o1", "R", "o2")*</i>
Zhi#code:	<i>o1.R.Add(o2)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o1</i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o2</i> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>addObjectPropertyValueChecked("o1", "R", "o2")*</i>
Zhi#code:	<i>o1.R.AddChecked(o2)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o1</i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o2</i> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> ) PC-OWLAPI-HAS-OBJECTPROPERTY( <i>o1</i> , <i>R</i> ) PC-OWLAPI-IS-RANGE-OBJECT( <i>o2</i> , <i>R</i> )

CHIL OWL API:	<i>deleteObjectPropertyValue("o1", "R", "o2")*</i>
Zhi#code:	<i>o1.R.Delete(o2)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o1</i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o2</i> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>assignObjectPropertyValue("o1", "R", "o2")*</i>
Zhi#code:	<i>o1.R.Assign(o2)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o1</i> )* PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o2</i> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>assignObjectPropertyValueChecked</i> ("o <sub>1</sub> ", "R", "o <sub>2</sub> ")*
Zhi#code:	<i>o<sub>1</sub>.R.AssignChecked</i> (o <sub>2</sub> )
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL(o <sub>1</sub> )* PC-OWLAPI-DECLARED-INDIVIDUAL(o <sub>2</sub> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY(R) PC-OWLAPI-HAS-OBJECTPROPERTY(o <sub>1</sub> , R) PC-OWLAPI-IS-RANGE-OBJECT(o <sub>2</sub> , R)

CHIL OWL API:	<i>addDatatypePropertyValue</i> ("o", "U", "v", "D")*
Zhi#code:	<i>o<sub>1</sub>.U.Add</i> (v, D)
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL(o)* PC-OWLAPI-DECLARED-DATATYPE(D) PC-OWLAPI-DECLARED-DATATYPEPROPERTY(U) PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE(D, U)

CHIL OWL API:	<i>addDatatypePropertyValueChecked</i> ("o", "U", "v", "D")*
Zhi#code:	<i>o<sub>1</sub>.U.AddChecked</i> (v, D)
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL(o)* PC-OWLAPI-DECLARED-DATATYPE(D) PC-OWLAPI-DECLARED-DATATYPEPROPERTY(U) PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE(D, U) PC-OWLAPI-HAS-DATATYPEPROPERTY(o, U) PC-OWLAPI-IS-RANGE-TYPE(D, U)

CHIL OWL API:	<i>deleteDatatypePropertyValue("o", "U", "v", "D")*</i>
Zhi#code:	<i>o<sub>1</sub>.U.Delete(v, D)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-DATATYPE( <i>D</i> ) PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> ) PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE( <i>D, U</i> )

CHIL OWL API:	<i>assignDatatypePropertyValue("o", "U", "v", "D")*</i>
Zhi#code:	<i>o<sub>1</sub>.U.Assign(v, D)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-DATATYPE( <i>D</i> ) PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> ) PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE( <i>D, U</i> )

CHIL OWL API:	<i>assignDatatypePropertyValueChecked("o", "U", "v", "D")*</i>
Zhi#code:	<i>o<sub>1</sub>.U.AssignChecked(v, D)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-DATATYPE( <i>D</i> ) PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> ) PC-OWLAPI-IS-COMPATIBLE-PRIMITIVE-BASE-TYPE( <i>D, U</i> ) PC-OWLAPI-HAS-DATATYPEPROPERTY( <i>o, U</i> ) PC-OWLAPI-IS-RANGE-TYPE( <i>D, U</i> )

## H.2 The *IAskingABox* Interface

CHIL OWL API:	<i>listSameIndividuals("o")*</i>
Zhi#code:	<i>o.ListSame()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )*

CHIL OWL API:	<i>listDifferentIndividuals("o")*</i>
Zhi#code:	<i>o.ListDifferent()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )*

CHIL OWL API:	<i>listIndividualsOfClass("C")*</i>
Zhi#code:	<i>C.List()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>getCardinalityOfClass("C")*</i>
Zhi#code:	<i>C.Cardinality</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listRDFTypesOfIndividual("o")*</i>
Zhi#code:	<i>o.ListRDFTypes()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )*

CHIL OWL API:	<i>listDirectRDFTypesOfIndividual("o")*</i>
Zhi#code:	<i>o.ListDirectRDFTypes()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )*

CHIL OWL API:	<i>listObjectPropertiesOfIndividual("o")</i>
Zhi#code:	<i>o.ListObjectProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )*

CHIL OWL API:	<i>listDatatypePropertiesOfIndividual("o")</i>
Zhi#code:	<i>o.ListDatatypeProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )*



CHIL OWL API:	<i>listObjectPropertyValuesOfIndividual("o", "R")</i>
Zhi#code:	<i>o.ListObjectPropertyValues(R)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )
CHIL OWL API:	<i>getCardinalityOfObjectPropertyValuesOfIndividual("o", "R")</i>
Zhi#code:	<i>o.GetObjectPropertyValuesCount(R)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )
CHIL OWL API:	<i>listDatatypePropertyValuesOfIndividual("o", "U")</i>
Zhi#code:	<i>o.ListDatatypePropertyValues(U)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )
CHIL OWL API:	<i>getCardinalityOfDatatypePropertyValuesOfIndividual("o", "U")</i>
Zhi#code:	<i>o.GetDatatypePropertyValuesCount(U)</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-INDIVIDUAL( <i>o</i> )* PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )

### H.3 The *IAskingTBox* Interface

CHIL OWL API:	<i>listSubClasses("C")</i>
Zhi#code:	<i>C.ListSubClasses()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listDirectSubClasses("C")</i>
Zhi#code:	<i>C.ListDirectSubClasses()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listEquivalentClasses("C")</i>
Zhi#code:	<i>C.ListEquivalentClasses()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listDisjointClasses("C")</i>
Zhi#code:	<i>C.ListDisjointClasses()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listSuperClasses("C")</i>
Zhi#code:	<i>C.ListSuperClasses()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listDirectSuperClasses("C")</i>
Zhi#code:	<i>C.ListDirectSuperClasses()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listObjectPropertiesOfClass("C")</i>
Zhi#code:	<i>C.ListObjectProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listDatatypePropertiesOfClass("C")</i>
Zhi#code:	<i>C.ListDatatypeProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-CONCEPT( <i>C</i> )

CHIL OWL API:	<i>listSubObjectProperties("R")</i>
Zhi#code:	<i>R.ListSubProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>listEquivalentObjectProperties("R")</i>
Zhi#code:	<i>R.ListEquivalentProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>listInverseObjectProperties("R")</i>
Zhi#code:	<i>R.ListInverseProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>listSubDatatypeProperties("U ")</i>
Zhi#code:	<i>U.ListSubProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )

CHIL OWL API:	<i>listEquivalentDatatypeProperties("U ")</i>
Zhi#code:	<i>U.ListEquivalentProperties()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )

CHIL OWL API:	<i>listDomainOfObjectProperty("R")</i>
Zhi#code:	<i>R.ListDomain()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>listRangeOfObjectProperty("R")</i>
Zhi#code:	<i>R.ListRange()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>listDomainOfDatatypeProperty("U")</i>
Zhi#code:	<i>U.ListDomain()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )

CHIL OWL API:	<i>listRangeOfDatatypeProperty("U")</i>
Zhi#code:	<i>U.ListRange()</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )

CHIL OWL API:	<i>isFunctionalObjectProperty("R")</i>
Zhi#code:	<i>R.IsFunctional</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>isInverseFunctionalObjectProperty("R")</i>
Zhi#code:	<i>R.IsInverseFunctional</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>isSymmetricObjectProperty("R")</i>
Zhi#code:	<i>R.IsSymmetric</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>isTransitiveObjectProperty("R")</i>
Zhi#code:	<i>R.IsTransitive</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-OBJECTPROPERTY( <i>R</i> )

CHIL OWL API:	<i>isFunctionalDatatypeProperty("U")</i>
Zhi#code:	<i>U.IsFunctional</i>
Checkable preconditions:	PC-OWLAPI-DECLARED-DATATYPEPROPERTY( <i>U</i> )

# APPENDIX I

## Zhi# Syntax and Semantics

The Zhi# programming language is a proper superset of ECMA standard C# version 1.0 [HWG02]. Zhi# includes a modicum of syntactical extensions to accommodate for the import and use of external type definitions in Zhi# programs. The syntactical extensions provided by the Zhi# compiler framework are fixed (i.e. compiler plug-ins cannot augment the syntax of the Zhi# programming language). Zhi#'s syntactical extensions to the conventional C# language grammar comprise the *import*-directive for including external namespaces, the naming scheme of external types in Zhi# programs, and seven additional binary operators, which correspond to constraining facets in XML Schema Definition [BM04b]. The syntactical extensions are described in Section I.1.

Compiler plug-ins contribute external (sub-)typing and type inference mechanisms of external type systems. External type systems determine the semantics of object creation and conversion, object-oriented member access, operators, assignments, and runtime type checks in Zhi# programs that involve external type definitions. In particular, compiler plug-ins can provide for property access and method invocations on external types on the class and instance level. External types can be indexed and be subject to iterator expressions. Compiler plug-ins can implement meta-properties of external types that do not exist in the external type system but improve the programmability.

Section I.2 specifies the semantics of the *import*-directive and XSD and OWL-specific semantics of object creations and conversions, object-oriented member access, operators, assignments, and runtime type checks. Both syntax as well as semantics are specified complementary to the C# language specification.

## I.1 Syntax

This section describes Zhi#'s extensions to the C# 1.0 lexical and syntactic grammars. Extensions to the original C# grammar productions are underlined. The C# 1.0 lexical and syntactic grammar is given in Appendix A.13 and A.14, respectively, in the ECMA 334 standard specification [HWG02]. The following paragraph names correspond to paragraph names in the ECMA specification. Omitted production rules in paragraphs are indicated with ellipses. Monospace font indicates Zhi# keywords.

### Basic concepts (syntactic)

type-name:

namespace-or-type-name

external-type-name

external-type-name:

#identifier#identifier

...

### Namespaces (syntactic)

compilation-unit:

using-directives<sub>opt</sub> import-directives<sub>opt</sub> global-attributes<sub>opt</sub> ↪

namespace-member-declarations<sub>opt</sub>

namespace-body:

{ using-directives<sub>opt</sub> import-directives<sub>opt</sub> namespace-member-declarations<sub>opt</sub> }

import-directives:

import-directive

import-directives import-directive

import-directive:

**import** identifier identifier = external-namespace-name ;

...

## Keywords (lexical)

keyword: *one of*

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	false	finally	float	for	foreach	goto
if	implicit	<u>import</u>	in	int	interface	internal
is	lock	long	namespace	new	null	object
operator	out	override	params	private	protected	public
readonly	ref	return	sbyte	sealed	short	sizeof
static	string	struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked	ushort	using
virtual	void	volatile	while			

## Operators and punctuators (lexical)

operator-or-punctuator: *one of*

{	}	[	]	(	)	.	,	:	;	+	-	*
/	%	&		^	!	~	=	<	>	?	++	--
&&		<<	>>	==	!=	<=	>=	+=	-=	*=	/=	%=
&=	=	^=	<<=	>>=	<u>?=</u>	<u>?&lt;</u>	<u>?&gt;</u>	<u>??</u>	<u>\$=</u>	<u>%%</u>	<u>%.</u>	

Note: In C# 3.0 the “null coalescing” operator (??) was introduced, which checks whether the value provided on the left side of the expression is *null*, and if so it returns an alternative value indicated by the right side of the expression.

## External namespace name (additional lexical)

external-namespace-name:

*An RFC 2396<sup>1</sup> compliant Uniform Resource Identifier (URI)*

---

<sup>1</sup>see [BFM98]

## I.2 Semantics

Following the specification of the Zhi# syntax by means of a formal grammar, this section specifies the semantics of Zhi# expressions that involve external types. Programming language semantics can be defined in a number of ways. The behavior of programming languages can be made explicit by model (or reference) implementations. *Reference implementations* are designated as authoritative. The meaning of a program can simply be determined by running the program on the reference implementation, which, of course, needs to behave deterministically for the given input. The model implementation approach has been used for, for example, the Perl programming language. The implementation of the Zhi# programming language described in this work is, however, not intended to be a reference implementation since this would conflate bugs of the current implementation with the language specification. Notably, in case of Perl, the existence of a reference implementation has prevented any other implementation of the language. In contrast, the characteristic features of the Zhi# programming language may prospectively be adopted in other programming languages as well. In the same vein, Zhi# is not completely defined by the *test suite* mentioned in Section 7.1. Obviously, language implementations that can *only* run the test suite would be near to useless. Still, the approximately 12 KLOC regression test code may certainly be used for testing different implementations of the Zhi# programming language. In particular, similar test code could be used if Zhi# language features are integrated with other statically typed object-oriented programming languages such as, for example, Java. *Formal semantics* of programming languages are grounded in mathematics. Formal semantics allow for mathematical proofs of program correctness and the soundness of type systems. However, mathematical rigor to devise unambiguous and uniform language standards is mostly applicable only to simple programming languages, such as, for example, the  $\lambda_C$ -calculus (see Chapter 3). In practice, formal semantics of real life programming languages quickly become too difficult for practical use and are therefore often accompanied by natural language descriptions. Semantic definitions of, for exam-



ple, the C language are given in *natural language*. In the following two subsections, the semantics of Zhi# expressions that involve XML data types and OWL DL ontologies will be explained. Based on the discussion above, semantic definitions are provided in natural language. In particular, semantics are specified for object creations and conversions, object-oriented member access, runtime type checks, and method invocations.

## I.2.1 XML Schema Definition

The following specification of the behavior of the use of XML data types in Zhi# programs is complementary to the ECMA 334 standard specification of the C# programming language [HWG02], which applies for all cases that are not considered below. In particular, Zhi# grammar production rules can be found in Section I.1 above and in [HWG02]. For the sake of brevity, in this section, citations of paragraphs denoted by § or §§ refer to sections in the C# language specification [HWG02].

### I.2.1.1 Import directives

Import directives facilitate the use of XML data types that are defined in XML namespaces (cf. § 9.3). The scope of an *import-directive* extends over the *namespace-member-declarations* of its immediately containing compilation unit or Zhi# namespace body. The scope of an *import-directive* specifically does not include its peer *import-directives*. Thus, peer *import-directives* do not affect each other, and the order in which they are written is insignificant.

import-directives:

import-directive

import-directives import-directive

import-directive:

**import** identifier identifier = external-namespace-name ;

The first identifier following the *import* keyword serves as a type system evidence, which determines the Zhi# compiler plug-in that processes the *import-directive* (*XML* is used for XML Schema Definition). A compile-time error occurs if no plug-in is available for the specified type system. The second identifier serves as an alias for an XML namespace within the immediately enclosing compilation unit or Zhi# namespace body. Within member declarations in a compilation unit or Zhi# namespace body that contains an *import-directive*, the identifier introduced by the *import-directive* can be used to reference the given XML namespace. For example:

```
1 import XML xsd = http://www.w3.org/2001/XMLSchema;
2 class C {
3     public static void Main() {
4         #xsd#int = 23;
5     }}
```

The identifier of an *import-directive* must be unique within the declaration space of the compilation unit or Zhi# namespace that immediately contains the *import-directive*. It is a compile-time error for two or more *import-directives* in the same compilation unit or Zhi# namespace body to declare aliases by the same name. For example:

```
1 import XML xsd = namespace 1;
2 import XML xsd = namespace 2; // Error, xsd already exists
```

An *import-directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *import-directive* is not transitive but rather affects only the compilation unit or Zhi# namespace body in which it occurs. In the example

```
1 namespace N1 {
2     import XML xsd = http://www.w3.org/2001/XMLSchema;
3 }
4 namespace N1 {
```

```

5 class C {
6     public static void Main() {
7         #xsd#int = 23;
8     }}}

```

the scope of the *import-directive* that introduces *xsd* only extends to member declarations in the *Zhi#* namespace body in which it is contained, so *xsd* is unknown in the second namespace declaration. However, placing the *import-directive* in the containing compilation unit causes the alias to become available within both namespace declarations:

```

1 import XML xsd = http://www.w3.org/2001/XMLSchema;
2 namespace N1 {
3     [...]
4 }
5 namespace N1 {
6     class C {
7         public static void Main() {
8             #xsd#int = 23;
9         }}}

```

Just like regular members, aliases introduced by *import-directives* are hidden by similarly named members in nested scopes. In the example

```

1 import XML xsd = http://www.w3.org/2001/XMLSchema;
2 namespace N1 {
3     class C {
4         public static void Main() {
5             int xsd = 0;
6             #xsd#int = 23; // Error, xsd has no member int
7         }}}

```

the reference to *#xsd#int* in line 6 causes a compile-time error because *xsd* refers to the *System.Int32* variable declared in line 5.

The order in which *import-directives* are written has no significance, and resolution of the *external-namespace-name* referenced by an *import-directive* is not affected by the *import-directive* itself or by other *import-directives* in the immediately containing compilation unit or Zhi# namespace body. In other words, the *external-namespace-name* of an *import-directive* is resolved as if the immediately containing compilation unit or Zhi# namespace body had no *import-directives*.

### I.2.1.2 XML data types and variables

XML data types can be used in Zhi# programs in all places where .NET types are admissible except for type declarations. XML data types appear as nullable values types in Zhi# programs (i.e. XML variables can be assigned the *null* value). The default value of XML variables is *null*. Hence, XML variables must be *definitely assigned* (see § 5.3) before use. Variables are definitely assigned if the compiler can statically prove that the variable has been initialized or has been assigned a value.

Reads and writes of XML data types are not guaranteed to be atomic (because reads and writes of the .NET types *long*, *ulong*, *double*, and *decimal*, as well as user-defined .NET types, are not guaranteed to be atomic either).

The value of an XML variable is the XML data type value that has been assigned to the variable or the default value *null* if no value has been assigned (cf. § 5.3.1). XML variables expose a *Value* property of type *string*. The *Value* property can be used both in Zhi# as well as in compiled Zhi# programs (i.e. conventional C#) to obtain and to set the value of an XML variable. The use of the *Value* property can be omitted in Zhi# as shown below. Both in Zhi# as well as in C# assigned values are always checked to be valid in respect of the underlying XML schema definition.

```
1 #xsd#int i;  
2 i = 23;  
3 i.Value = 23;
```

### I.2.1.3 Conversions

**Implicit conversions** A conversion enables an expression of one type to be treated as another type (cf. § 6). XML data types implicitly convert to themselves. This conversion exists only such that an entity that already has a required type can be said to be convertible to that type. Implicit conversions from built-in XML Schema Definition data types to built-in .NET value types and the .NET type *string* are:

- From *xsd#string* to *string*.
- From *xsd#boolean* to *bool*.
- From *xsd#float* to *float* or *double*.
- From *xsd#double* to *double*.
- From *xsd#byte* to *sbyte*, *short*, *int*, *long*, *float*, *double*, or *decimal*.
- From *xsd#unsignedByte* to *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, or *decimal*.
- From *xsd#short* to *short*, *int*, *long*, *float*, *double*, or *decimal*.
- From *xsd#unsignedShort* to *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, or *decimal*.
- From *xsd#int* to *int*, *long*, *float*, *double*, or *decimal*.
- From *xsd#unsignedInt* to *uint*, *long*, *ulong*, *float*, *double*, or *decimal*.
- From *xsd#long* to *long*, *float*, *double*, or *decimal*.
- From *xsd#unsignedLong* to *ulong*, *float*, *double*, or *decimal*.

Conversions from *xsd#int*, *xsd#unsignedInt*, *xsd#long*, or *xsd#unsignedLong* to *float* and from *xsd#long* or *xsd#unsignedLong* to *double* may cause a loss of precision, but will never cause a loss of magnitude. The other implicit conversions never lose any information.

Implicit conversions from built-in .NET value types and type *string* to built-in XML Schema Definition data types are:

- From *string* to *xsd#string*.
- From *bool* to *xsd#boolean*.
- From *float* to *xsd#float* or *xsd#double*.
- From *double* to *xsd#double*.
- From *sbyte* to *xsd#byte*, *xsd#short*, *xsd#int*, *xsd#long*, *xsd#integer*, *xsd#decimal*, *xsd#float*, or *xsd#double*.
- From *byte* to *xsd#unsignedByte*, *xsd#short*, *xsd#unsignedShort*, *xsd#int*, *xsd#unsignedInt*, *xsd#long*, *xsd#unsignedLong*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.
- From *short* to *xsd#short*, *xsd#int*, *xsd#long*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.
- From *ushort* to *xsd#unsignedShort*, *xsd#int*, *xsd#unsignedInt*, *xsd#long*, *xsd#unsignedLong*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.
- From *int* to *xsd#int*, *xsd#long*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.
- From *uint* to *xsd#unsignedInt*, *xsd#long*, *xsd#unsignedLong*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.
- From *long* to *xsd#long*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.
- From *ulong* to *xsd#unsignedLong*, *xsd#integer*, *xsd#decimal*, *xsd#float*, *xsd#double*.

Conversions from *int*, *uint*, *long*, or *ulong* to *xsd#float* and from *long* or *ulong* to *xsd#double* may cause a loss of precision, but will never cause a loss of magnitude. The other implicit conversions never lose any information.

**Standard explicit conversions** Standard explicit conversions between XML data types and .NET types are defined exactly like standard explicit conversions between .NET types (see § 6.3.2). The standard explicit conversions between XML data types and .NET types are all standard implicit conversions plus the subset of the explicit conversions between XML data types and .NET types for which an opposite standard implicit conversion exists. If a standard implicit conversion exists from a type  $A$  to a type  $B$ , then a standard explicit conversion exists from type  $A$  to type  $B$  and from type  $B$  to type  $A$ .

**User-defined conversions** Zhi# allows the pre-defined implicit and explicit conversions between XML data types and .NET types to be augmented by certain user-defined conversions. User-defined conversions are introduced by declaring conversion operators in .NET class and struct types as described in § 10.9.3. It is not possible to redefine an already existing implicit or explicit conversion between an XML data type and a .NET type. A .NET class or struct is permitted to declare a conversion (implicit or explicit) from an XML source type  $S$  to a .NET target type  $T$  or from a .NET source type  $S$  to an XML target type  $T$  only if all of the following are true:

- Either  $S$  or  $T$  is the .NET class or struct type in which the operator declaration takes place.
- Either  $S$  or  $T$  is an XML data type.
- Neither  $S$  nor  $T$  is *object* or a .NET interface-type.

User-defined conversions between XML data types and .NET types are evaluated according to §§ 6.4.2-6.4.4 based on the above stipulations for XML data types.

#### I.2.1.4 Expressions

**Operators** An expression is a sequence of operators and operands (cf. § 7). XML data types can be used with binary operators and the ternary operator (?:). Binary operators

applicable to XML data types are listed in Section E.1. Binary operators can be used with an XML data type and a .NET type if an implicit conversion exists between the XML data type and the .NET type. The type of logical expressions involving XML data types is *bool*. The type of arithmetic expressions is an XML data type, which is computed based on the  $\lambda_C$ -constraint arithmetic algorithm (see Fig. 3.3) and the XSD constraint arithmetic (see Section E.2). No binary assignment operators can be used with XML data types. No unary operators can be used with XML data types. The semantics of the XML-specific operators *?=*, *?<*, *?>*, *\$=*, *??*, *%%*, and *%.* is as defined in Section 5.1.3. The precedence of the XML-specific operators is the same as of C#'s relational and type testing operators (see § 7.9). When an operand occurs between two operators with the same precedence, the associativity of the operators as defined in the C# language specification controls the order in which the operations are performed. Binary operators that relate two XML data types cannot be overloaded. Binary operators are processed according to §§ 7.2.4 and 7.2.5. Cast operations are overloaded by providing user-defined conversions.

**Object creation** XML variables are declared like .NET variables (see § 8.5). The initial value of a variable can be an object for which an implicit conversion is defined to the type of the XML variable. The initial value can be provided as is or as the argument of an *object-creation-expression* (see § 7.5.10).

object-creation-expression:

```
new type ( argument-listopt )
```

For XML data types, exactly one *argument* must be provided for which an implicit conversion is defined to the type of the XML variable. Alternatively, a *string* object can be provided that denotes an element of the lexical space of the XML data type. The use of the *new*-operator is not required and can be omitted as shown below. There is no default constructor for XML data types. The default value of uninitialized XML variables is *null*. The following example shows how an *xsd#int* variable can be initialized with an



*int* literal or *object-creation-expressions*. The *string* value that is provided to the *object-creation-expression* in line 3 is interpreted as a lexical representation of an element of the *xsd#int* value space. Variable *l*, which is declared in line 4, remains uninitialized.

```
1 #xsd#int i = 23;           // i == 23
2 #xsd#int j = new #xsd#int (23); // j == 23
3 #xsd#int k = new #xsd#int ("23"); // k == 23
4 #xsd#int l;                // l == null
```

**Arrays** An array is a data structure that contains a number of elements that are accessed through computed indices (see § 12). In *Zhi#*, the element type of an array can be an XML data type. For arrays that contain XML variables, the same rules for array types, array creation, element access, array members, and array initializers are effective as specified in § 12. Arrays of XML data types behave differently from arrays of .NET types in terms of array covariance. *C#* allows covariant subtyping of arrays (see § 12.5). There is no array covariance for arrays of XML data types in *Zhi#*. For example, the following assignment causes a compile-time error despite the fact that an implicit conversion exists from *xsd#byte* to *xsd#int*.

```
#xsd#int [] a = new #xsd#byte [1]; // Error, no array covariance
```

**The *is*-Operator** The *is*-operator is used to dynamically check if the runtime type of an object is compatible with a given type (see § 7.9.9). The type of the operation *e is T*, where *e* is an expression and *T* is a type, is *bool*. When used with an XML data type, the operation is evaluated as shown below. No user-defined conversions are considered by the *is*-operator.

- If the value of *e* is *null*, the result is *false*.
- Otherwise, an anonymous object of type *T* is created and initialized with value *e*. If the initialization succeeds, the result is *true*; *false* otherwise.

**Assignments** The simple assignment operator ( $=$ ) (see § 7.13.1) assigns a new value to an XML variable. Zhi# admits covariant coercions for XML data types. An assignment of the form  $x = y$ , where  $x$  or  $y$  are XML data types, is valid if the type of the rvalue (i.e. the right operand) is subsumed by the type of the lvalue (i.e. the left operand) or if an implicit conversion is defined from the type of  $y$  to the type of  $x$ . If the lvalue is an XML data type, only the *value* (and not the *type*) of the source operand is assigned such that, a fresh instance of the XML data type of the lvalue is created and initialized with the rvalue. The rvalue evaluates to itself if its type is subsumed by the type of the lvalue or to the result of a defined implicit conversion operator, which converts the rvalue to the type of the lvalue. The type of the source operand may have an appropriate value space that was inferred based on static flow analysis as described below. In contrast to arrays of .NET types, there is no array covariance for arrays of XML data types in Zhi# (see above). Compound assignments of the form  $x \text{ op} = y$  are evaluated for XML data types as specified in § 7.13.2. In particular,  $x$  is evaluated only once.

**Type inference** XML data types and .NET data types for which built-in implicit conversions to XML data types exist are subject to type inference as described in Section 5.1.3. The XSD type inference rules are specified in Appendix D. The compile-time XSD type and constraint arithmetic rules are specified in Appendix E.

**Method overriding** The variance rules for method overriding in Zhi# are as follows. Formal XSD input parameters can be contravariant. Formal XSD output parameters can be covariant. For .NET types the variance rules are unchanged to the C# specification.

**Member lookup** XML variables in Zhi# programs expose the public *Type* and *Value* members. The read-only *string* property *Type* yields the XML data type of the XML variable. The read-write *string* property *Value* can be used to obtain and to set the value of an XML variable.

## I.2.2 Web Ontology Language

The following specification of the behavior of the use of OWL classes in Zhi# programs is complementary to the ECMA 334 standard specification of the C# programming language. Citations of paragraphs denoted by § or §§ refer to sections in the C# language specification [HWG02]. Zhi# grammar production rules can be found in Section I.1. In the example programs, fully qualifying namespace aliases are omitted for brevity.

### I.2.2.1 Import directives

Import directives work for OWL namespaces like for XML namespaces. The type system evidence *OWL* specifies an OWL namespace.

### I.2.2.2 OWL classes and variables

OWL classes can be used in Zhi# programs in all places where .NET types are admissible except for type declarations. OWL variables refer to ontological individuals in a shared ontological knowledge base, which is configured in the *App.config* configuration file of a Zhi# application. OWL variables can be assigned the *null* value. The default value of OWL variables is *null*. OWL variables must be *definitely assigned* (see § 5.3) before use. The value of an OWL variable is the ontological individual that has been assigned to the variable or the default value *null* if no value has been assigned (cf. § 5.3.1).

Before each single use of an OWL non-array variable the ontological individual that is referred to by the variable is dynamically checked to be in the extension of the declared OWL class and the knowledge base is checked to be consistent. An *InvalidCastException* and *InconsistentOntologyException* is thrown, respectively, if either is not the case.

Reads and writes of OWL variables are not guaranteed to be atomic because 1) the ontological knowledge base is likely to be remote and 2) the ontology management system cannot be assumed to provide for atomic operations.

### I.2.2.3 Conversions

**Implicit conversions** OWL classes implicitly convert to themselves. There are no implicit conversions between OWL classes and .NET types. OWL class instances implement the ubiquitous *ToString()* method, which returns the URI of the ontological individual as a .NET *string*. Hence, OWL variables can be sensibly used with a variety of .NET Base Class Library (BCL) methods that require the string representation of objects (e.g., *Console.WriteLine()*).

**Standard explicit conversions** A standard explicit conversion between two OWL classes exists if both classes are not declared to be disjoint.

**User-defined conversions** A .NET class or struct is permitted to declare a conversion (implicit or explicit) from an OWL source type *S* to a .NET target type *T* or from a .NET source type *S* to an OWL target type *T* only if all of the following are true.

- Either *S* or *T* is the .NET class or struct type in which the operator declaration takes place.
- Either *S* or *T* is an OWL class.
- Neither *S* nor *T* is *object* or a .NET interface-type.

User-defined conversions between OWL classes and .NET types are evaluated like user-defined conversions between .NET types as specified in §§ 6.4.2-6.4.4.

### I.2.2.4 Expressions

**Object creation** OWL variables are declared like .NET variables (see § 8.5). The initial value of an OWL variable can be a .NET object for which an implicit conversion to the declared type exists, a defined OWL variable, or an *object-creation-expression* (see §

7.5.10) that answers an ontological individual (i.e. OWL class instance) in the ontological knowledge base. Existing individuals in the knowledge base with the same name are reused, following Semantic Web standards. `Zhi#` provides a constructor for OWL class instances that takes the URI of the individual. There is no default constructor. As in `C#`, the *new*-operator cannot be overloaded. The answered ontological individuals are subject to ontological reasoning. Individual answering with the *new*-operator corresponds to the use of the `addIndividual()` method of the CHIL OWL API. For constant constructor parameters (e.g., string literals) the provided class name is statically checked to be declared in the ontology. A compile-time error occurs for undeclared class names.

**Arrays** In `Zhi#`, the element type of an array can be an OWL class. For arrays that contain OWL variables, the same rules for array types, array creation, element access, array members, array covariance, and array initializers are effective as specified in § 12.

**The *is*-Operator** The *is*-operator can be used with an OWL class instance and an OWL class name to check the runtime RDF type of the class instance. The type of the operation *e is T*, where *e* is an OWL class instance and *T* is an OWL class name, is *bool*. The operation is evaluated as shown below. No user-defined conversions are considered by the *is*-operator. Compile-time errors occur for disjoint compile-time types of *e* and *T*.

- If the value of *e* is *null*, the result is *false*.
- Otherwise, the result is *true* if the ontological individual referred to by *e* is in the extension of the named class *T*; *false* otherwise.

**Object equality** The equality (`==`) and inequality (`!=`) operator can be used to test if two OWL class instances are equal or different, respectively. The `==` and `!=`-operator evaluate on the ontology level. In the following example, the expression `a == b` in line 3 evaluates to *true* if the ontological individuals A and B, which are referred to by variables

$a$  and  $b$ , refer to the same entities in the described world;  $a == b$  evaluates to *false* otherwise. The expression  $a != b$  in line 4 evaluates to *true* if the ontological individuals A and B are known to *not* refer to the same entities in the described world;  $a != b$  evaluates to *false* otherwise.

```
#A a = new #A("#A"); // a refers to A
#B b = new #B("#B"); // b refers to B
if (a == b) { Console.WriteLine("{0} and {1} are the same", a, b);}
if (a != b) { Console.WriteLine("{0} and {1} are not the same", a, b);}
```

The `==` and `!=`-operator do *not* evaluate as the the logical negations of each other since individuals can be unknown to be identical or different. In particular, the expressions  $a == b$  and  $a != b$  can simultaneously evaluate to *false* for two individuals A and B that are neither the same nor different.

**Assignments** The simple assignment operator (`=`) (see § 7.13.1) assigns a new value to an OWL variable. `Zhi#` admits covariant coercions for OWL classes. Assignments to OWL variables do not modify the ontological knowledge base. Assignments to properties of OWL variables assert the given rvalue as a value for the given property of the given host object. The following example adds the statement  $[A R B]$  to the ontology, where A and B are the individuals referred to by variables  $a$  and  $b$ , respectively.

```
a.R = b;
```

The semantics of assignments to OWL object and non-functional OWL datatype properties are *additive*. The specified statement is added to the knowledge base; no statements are removed. Assignments to functional OWL datatype properties update the knowledge base with the specified statement (i.e. all statements with the host object as the subject and the given property are removed from the knowledge base and the specified statement is added). A compile-time error occurs if the property is not declared in the ontology. A compile-time error occurs if the compile-time type of the rvalue (i.e. the asserted type of

an OWL class instance) is disjoint with the range declaration of an OWL object property.

Property assignments are not used for compile-time type inference (i.e. ontological reasoning is not used to infer compile-time types of OWL variables).

The type of an assignment to an OWL object and non-functional datatype property is an array type, where the element type is the property range declaration. The type of an assignment to a functional OWL datatype property is the property range declaration.

Assignments to OWL datatype properties are valid if the type of the rvalue is subsumed by the property range declaration or if an implicit conversion is defined from the type of the rvalue to the property range declaration. The type of the rvalue may have an appropriate value space that was inferred based on static flow analysis (see Subsection I.2.1). A compile-time error occurs if the declared or inferred type of the rvalue is not a subtype of the range declaration and there is no implicit conversion.

The *checked*-keyword (c.f. 7.5.12) supports a frame-view on OWL object properties. For assignments to OWL object properties within a *checked*-context the subject and object of the declared triple must be in the extension of the property domain and range restriction, respectively. An exception is thrown if either is not the case. The *checked*-operator does not influence the static type checks that apply, for example, to disjoint concept definitions. The *checked*-keyword can be used as an operator or a statement.

**Member lookup** The type of an OWL object and non-functional datatype property is an array type, where the element type is the property range declaration. The type of a functional OWL datatype property is the property range declaration.

OWL property access expressions are iterable by means of a *foreach*-loop. The values of the *foreach*-iteration variable are dynamically checked to be in the extension of the declared type (i.e. OWL class) of the iteration variable.

Auxiliary properties and methods are defined for OWL class instances, OWL properties, and static OWL class references as described in Appendix H.





## REFERENCES

- [ACP89] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. “Dynamic typing in a statically-typed language.” In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 213–227, New York, NY, USA, January 1989. ACM Press. 213, 298
- [ACP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. “Dynamic typing in a statically-typed language.” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **13**(2):237–268, April 1991. 213, 298
- [AH03] Grigoris Antoniou and Frank van Harmelen. *Handbook on Ontologies in Information Systems*, chapter Web Ontology Language: OWL, pp. 67–92. International Handbooks on Information Systems. Springer Verlag, 2003. 35
- [AHM07] D. Akehurst, G. Howells, and K. McDonald-Maier. “Implementing associations: UML 2.0 to Java 5.” *Software and Systems Modeling*, **6**(1):3–35, March 2007. 247
- [ANM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. “A framework for implementing pluggable type systems.” *ACM SIGPLAN Notices*, **41**(10):57–74, October 2006. 76
- [Asp08] “AspectJ.”, December 2008. <http://www.eclipse.org/aspectj/>. 285
- [Baa90] Franz Baader. “Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles.” Technical Report RR-90-13, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Erwin-Schrödinger Strasse, Postfach 2080, 67608 Kaiserslautern, Germany, 1990. 17
- [BBG63] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, Michael Woodger, and Peter Naur. “Revised report on the algorithm language ALGOL 60.” *Communications of the ACM (CACM)*, **6**(1):1–17, January 1963. 45
- [BCF07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. “XQuery 1.0: An XML Query Language.” Technical report, World Wide Web Consortium (W3C), January 2007. 170
- [BCM03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, Cambridge, United Kingdom, 2003. 1, 25, 41, 105, 175, 213
- [Bec02] Sean Bechhofer. “The DIG Description Logic Interface: DIG/1.0.” Technical report, University of Manchester, Oxford Road, Manchester M13 9PL, October 2002. 137

- [Ber05] Tim Berners-Lee. “Primer: Getting into RDF and Semantic Web using N3.” Technical report, World Wide Web Consortium (W3C), August 2005. 29
- [BFM98] Tim Berners-Lee, R. Fielding, and L. Masinter. “RFC 2396 – Uniform Resource Identifiers (URI): Generic Syntax.” Technical report, Network Working Group, August 1998. 25, 55, 145, 389
- [BG04a] Andrew Begel and Susan L. Graham. “Language analysis and tools for ambiguous input streams.” *Electronic Notes in Theoretical Computer Science*, **110**:75–96, December 2004. 77
- [BG04b] Dan Brickley and R.V. Guha. “RDF Vocabulary Description Language 1.0: RDF Schema.” Technical report, World Wide Web Consortium (W3C), February 2004. 1, 32, 105
- [BH91] Franz Baader and Philipp Hanschke. “A scheme for integrating concrete domains into concept languages.” In John Mylopoulos and Raymond Reiter, editors, *12th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 452–457. Morgan Kaufmann, August 1991. 17
- [BHP99] Sean Bechhofer, Ian Horrocks, Peter F. Patel-Schneider, and Sergio Tessaris. “A proposal for a description logic interface.” In Patrick Lambrix, Alexander Borgida, Maurizio Lenzerini, Ralf Möller, and Peter F. Patel-Schneider, editors, *12th International Workshop on Description Logics (DL)*, volume 22 of *CEUR Workshop Proceedings*. CEUR-WS.org, July 1999. 137
- [BL84] Ronald J. Brachman and Hector J. Levesque. “The tractability of subsumption in frame-based description languages.” In Ronald J. Brachman, editor, *4th National Conference on Artificial Intelligence (AAAI)*, pp. 34–37, August 1984. 17
- [BL85] Ronald J. Brachman and Hector J. Levesque. *Readings in Knowledge Representation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1985. 12
- [BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. “JTS: Tools for implementing domain-specific languages.” In *5th International Conference on Software Reuse (ICSR)*, pp. 143–153, Washington, DC, USA, June 1998. IEEE Computer Society. 77
- [BM04a] Dave Beckett and Brian McBride. “RDF/XML Syntax Specification (Revised).” Technical report, World Wide Web Consortium (W3C), February 2004. 30
- [BM04b] Paul V. Biron and Ashok Malhotra. “XML Schema Part 2: Datatypes Second Edition.” Technical report, World Wide Web Consortium (W3C), October 2004. 4, 26, 58, 81, 82, 83, 85, 176, 387

- [BMP91] Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, Lori Alperin Resnick, and Alexander Borgida. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, chapter Living with CLASSIC: When and how to use a KL-ONE-like language, pp. 401–456. Principles of Semantic Networks. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1991. 138
- [BMS02] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. “The <bigwig> project.” *ACM Transactions on Internet Technology (TOIT)*, **2**(2):79–114, 2002. 170
- [BPS06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. “Extensible Markup Language (XML) 1.0 (Fourth Edition).” Technical report, World Wide Web Consortium (W3C), August 2006. 25, 27, 141
- [Bra77] Ronald J. Brachman. “What’s in a concept: Structural foundations for semantic networks.” *International Journal of Man-Machine Studies*, **9**(2):127–152, 1977. 12
- [Bra79] Ronald J. Brachman. *Associative Networks*, chapter On the epistemological status of semantic networks, pp. 3–50. Academic Press, 1979. 12
- [Bra04] Gilad Bracha. “Pluggable type systems.” In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004. 76
- [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, Inc., Redwood City, CA, USA, 1999. 36
- [BS89] Ronald J. Brachman and James G. Schmolze. *Artificial Intelligence and Databases*, chapter An overview of the KL-ONE knowledge representation system, pp. 207–230. Kaufmann Publishers, Inc., San Mateo, CA, USA, 1989. 17
- [BST94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. “The GenVoca model of software-system generators.” *IEEE Software*, **11**(5):89–94, September 1994. 285
- [BV04] Martin Bravenboer and Eelco Visser. “Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions.” In John M. Vlissides and Douglas C. Schmidt, editors, *19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 365–383, New York, NY, USA, October 2004. ACM Press. 77
- [BW04] Fahiem Bacchus and Toby Walsh. “A constraint algebra.” In Marc van Dongen, editor, *1st International Workshop on Constraint Propagation and Implementation (CPAI)*, pp. 1–15, September 2004. 171

- [Cam09] Campwood Software. “SourceMonitor.”, 2009. <http://www.campwoodsw.com/sourcemonitor.html>. 219
- [CCD08] Jonathan Calladine, George Cowe, Paul Downey, and Yves Lafon. “Basic XML Schema Patterns for Databinding Version 1.0.” Technical report, World Wide Web Consortium (W3C), March 2008. 223
- [CD99] James Clark and Steve DeRose. “XML Path Language (XPath) Version 1.0.” Technical report, World Wide Web Consortium (W3C), November 1999. 170
- [CFF97] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James Rice. “The Generic Frame Protocol 2.0.” Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA, USA, July 1997. 137
- [CFF98] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. “Open Knowledge Base Connectivity 2.0.3.” Technical Report KSL-09-06, Stanford University Knowledge Systems Laboratory, January 1998. 137
- [Cha81] Eugene Charniak. “A common representation for problem solving and language comprehension information.” *Artificial Intelligence*, **16**(3):225–255, July 1981. 12
- [Chi98] Shigeru Chiba. “Macro processing in object-oriented languages.” In *28th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, p. 113, Washington, DC, USA, November 1998. IEEE Computer Society. 285
- [Chu36] Alonzo Church. “A note on the Entscheidungsproblem.” *The Journal of Symbolic Logic*, **1**(1):40–41, 1936. 16
- [Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941. 41
- [CMA94] Luca Cardelli, Florian Matthes, and Martín Abadi. “Extensible Syntax with Lexical Scoping.” Technical Report 121, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, February 1994. 77
- [CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. “Semantic type qualifiers.” In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 85–95, New York, NY, USA, June 2005. ACM Press. 101, 171
- [CMM06] Brian Chin, Shane Markstrum, Todd D. Millstein, and Jens Palsberg. “Inference of user-defined type qualifiers and qualifier rules.” In Peter Sestoft, editor, *15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pp. 264–278, Berlin / Heidelberg, Germany, March 2006. Springer Verlag. 101, 171

- [CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. “Extending Java for high-level web service construction.” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **25**(6):814–875, 2003. 170
- [CT04] John Cowan and Richard Tobin. “XML Information Set (Second Edition).” Technical report, World Wide Web Consortium (W3C), February 2004. 25
- [DAM04] DAML Ontologists. “DAML Ontology Library.” Technical report, DARPA’s Information Exploitation Office, 2004. 1
- [DG87] Linda G. DeMichiel and Richard P. Gabriel. “The Common Lisp object system: An overview.” In *1st European Conference on Object-Oriented Programming (ECOOP)*, pp. 151–170, London, UK, 1987. Springer-Verlag. 213
- [DLN95] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Werner Nutt. “The Complexity of Concept Languages.” Technical Report RR-95-07, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1995. 17
- [Dru93] Peter Ferdinand Drucker. *Concept of the Corporation*. Transaction Publishers, January 1993. 10
- [DS04] Mike Dean and Guus Schreiber. “OWL Web Ontology Language Reference.” Technical report, World Wide Web Consortium (W3C), February 2004. 34, 36
- [FGK02] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. “XL: An XML programming language for web service specification and composition.” In *11th International World Wide Web Conference (WWW)*, volume 42, pp. 65–76, New York, NY, USA, May 2002. ACM Press. 170
- [FHH01] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. “OIL: An ontology infrastructure for the Semantic Web.” *IEEE Intelligent Systems*, **16**(2):38–45, March/April 2001. 34
- [FKS01] Wenfei Fan, Gabriel M. Kuper, and Jérôme Siméon. “A unified constraint model for XML.” In *10th International World Wide Web Conference (WWW)*, pp. 179–190, New York, NY, USA, 2001. ACM Press. 171
- [FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. “Flow-sensitive type qualifiers.” In *24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *ACM SIGPLAN Notices*, pp. 1–12, New York, NY, USA, June 2002. ACM Press. 101, 171
- [FW04] David C. Fallside and Priscilla Walmsley. “XML Schema Part 0: Primer Second Edition.” Technical report, World Wide Web Consortium (W3C), October 2004. 26, 141

- [GF07] David Greenfieldboyce and Jeffrey S. Foster. “Type qualifier inference for Java.” In *22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 42 of *ACM SIGPLAN Notices*, pp. 321–336, New York, NY, USA, October 2007. ACM Press. 101, 172
- [GHJ94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., Reading, MA, USA, 1994. 110
- [GJS05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3 edition, June 2005. 26
- [GP03] Vladimir Gapeyev and Benjamin C. Pierce. “Regular object types.” In Luca Cardelli, editor, *17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pp. 151–175. Springer Verlag, July 2003. 170
- [Gra95] Paul Graham. *ANSI Common LISP*. Prentice Hall, November 1995. 137
- [Gre91] R. Mac Gregor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, chapter The evolving technology of classification-based knowledge representation systems, pp. 385–400. Kaufmann, San Mateo, 1991. 17
- [Han92] Philipp Hanschke. “Specifying role interaction in concept languages.” In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *3rd International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 318–329, San Francisco, CA, USA, October 1992. Morgan Kaufmann. 17
- [Hay79] Patrick J. Hayes. *Frame Conceptions and Text Understanding*, chapter The logic of frames, pp. 46–61. Walter de Gruyter and Co., 1979. 12
- [HB91] Bernhard Hollunder and Franz Baader. “Qualifying Number Restrictions in Concept Languages.” DFKI Research Report RR-91-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, February 1991. 17
- [HEP03] Masahiro Hori, Jérôme Euzenat, and Peter F. Patel-Schneider. “OWL Web Ontology Language XML Presentation Syntax.” Technical report, World Wide Web Consortium (W3C), June 2003. 36
- [HHP01] Ian Horrocks, Frank van Harmelen, and Peter F. Patel-Schneider. “DAML+OIL.” Technical report, DARPA’s Information Exploitation Office and European Union’s Information Society Technologies, March 2001. 1, 34, 105

- [HJW92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. “Report on the programming language Haskell: a non-strict, purely functional language version 1.2.” *ACM SIGPLAN Notices*, **27**(5):1–164, May 1992. 45
- [HM01] Volker Haarslev and Ralf Möller. “RACER system description.” In R. Goré, A. Leitsch, and T. Nipkow, editors, *1st International Joint Conference on Automated Reasoning (IJCAR)*, pp. 701–706, London, United Kingdom, June 2001. Springer-Verlag. 18, 36
- [HM04] Patrick Hayes and Brian McBride. “RDF Semantics.” Technical report, World Wide Web Consortium (W3C), February 2004. 32
- [Hoa69] C. A. R. Hoare. “An axiomatic basis for computer programming.” *Communications of the ACM (CACM)*, **12**(10):576–580, October 1969. 106, 121, 122
- [Hol90] Bernhard Hollunder. “Hybrid inferences in KL-ONE-based knowledge representation systems.” In *14th German Workshop on Artificial Intelligence (GWAI)*, volume 251 of *Informatik-Fachberichte*, pp. 38–47, London, United Kingdom, 1990. Springer Verlag. 17
- [Hor98] Ian Horrocks. “The FaCT system.” In Harrie C. M. de Swart, editor, *7th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 1397 of *Lecture Notes in Computer Science*, pp. 307–312. Springer Verlag, May 1998. 17, 18, 36, 137
- [Hor99] Ian Horrocks. “FaCT and iFaCT.” In Patrick Lambrix, Alexander Borgida, Maurizio Lenzerini, Ralf Möller, and Peter F. Patel-Schneider, editors, *12th International Workshop on Description Logics (DL)*, volume 22 of *CEUR Workshop Proceedings*, pp. 133–135. CEUR-WS.org, July 1999. 137
- [Hor04] Matthew Horridge. “The Manchester Pizza Finder Application.”, 2004. <http://www.co-ode.org/downloads/pizzafinder/>. 280
- [HP 04] HP Labs. “Jena Semantic Web Framework.”, 2004. <http://jena.sourceforge.net>. 1, 2, 105, 109, 131, 135, 138, 176, 264
- [HP03] Haruo Hosoya and Benjamin C. Pierce. “XDuce: A statically typed XML processing language.” *ACM Transactions on Internet Technology (TOIT)*, **3**(2):117–148, 2003. 170
- [HP04] Ian Horrocks and Peter F. Patel-Schneider. “Reducing OWL entailment to description logic satisfiability.” *Journal of Web Semantics*, **1**(4):345–357, October 2004. 25, 36, 41, 176

- [HWG02] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. “C# Language Specification Version 1.0.” Technical report, ECMA International, 2002. 5, 27, 54, 56, 387, 388, 391, 401
- [Inf04] Information Society Technology integrated project 506909. “Computers in the Human Interaction Loop (CHIL).”, 2004. <http://chil.server.de>. 20, 37, 106, 175, 211, 281
- [Int86] International Organization for Standardization. “Standard Generalized Markup Language (SGML) (ISO 8879).” Technical report, International Organization for Standardization, 1986. 25
- [Int96] International Organization for Standardization. “Language-independent datatypes (ISO 11404).” Technical report, International Organization for Standardization, 1996. 26
- [Int99] International Organization for Standardization. “SQL – Part 2: Foundation (SQL/Foundation) (ISO 9075-2).” Technical report, International Organization for Standardization, 1999. 26
- [Int03] International Organization for Standardization. “Document Schema Definition Languages (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG (ISO 19757).” Technical report, International Organization for Standardization, 2003. 27
- [Int04] International Organization for Standardization. “Representation of dates and times (ISO 8601).” Technical report, International Organization for Standardization, December 2004. 83
- [JLS09] JLSOft. “SourceCode Counter.”, 2009. <http://www.jlsoft.de/-sourcecodecounter.html>. 219
- [KLM97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-oriented programming.” In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer Verlag, June 1997. 285
- [KMS04] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. “Static analysis of XML transformations in Java.” *IEEE Transactions on Software Engineering (TSE)*, **30**(3):181–192, March 2004. 170
- [Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, USA, 3 edition, 1997. 96



- [KPB04] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. “Automatic mapping of OWL ontologies into Java.” In Frank Maurer and Günther Ruhe, editors, *16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 98–103, June 2004. 212
- [KRB91] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. 213
- [KT06] Seiji Koide and Hideaki Takeda. “OWL Full reasoning from an object-oriented perspective.” In Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors, *1st Asian Semantic Web Conference (ASWC)*, volume 4185 of *Lecture Notes in Computer Science*, pp. 263–277. Springer Verlag, September 2006. 213
- [Lea66] B. M. Leavenworth. “Syntax macros and extended translation.” *Communications of the ACM (CACM)*, **9**(11):790–793, November 1966. 77
- [Lev84] Hector J. Levesque. “Foundations of a functional approach to knowledge representation.” *Artificial Intelligence*, **23**(2):155–212, July 1984. 137
- [LGA03] Juhnyoung Lee, Richard Goodwin, Rama Akkiraju, Yiming Ye, and Prashant Doshi. “IBM Ontology Management System (Snobase).”, 2003. 136
- [Lip82] Thomas A. Lipkis. *Report No. 4842: Proceedings of the 1981 KL-One Workshop*, chapter A KL-ONE classifier, pp. 128–145. Bolt, Beranek and Newman, Boston, MA, USA, 1982. 17
- [LM07] Ralf Lämmel and Erik Meijer. “Revealing the X/O impedance mismatch (Changing lead into gold).” In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, p. 80. Springer-Verlag, June 2007. 222, 223
- [LO04] Sten Loecher and Stefan Ocke. “A metamodel-based OCL-compiler for UML and MOF.” *Electronic Notes in Theoretical Computer Science*, **102**:43–61, November 2004. 285, 286
- [McC04] Steve McConnell. *Code Complete*. Microsoft Press, 2 edition, June 2004. 219
- [MD04] Erik Meijer and Peter Drayton. “Static typing where possible, dynamic typing when needed.” In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. 4, 234
- [MDW91] Eric Mays, Robert Dionne, and Robert Weida. “K-Rep system overview.” *SIGART Bulletin*, **2**(3):93–97, 1991. 17
- [MH04a] Deborah L. McGuinness and Frank van Harmelen. “OWL Web Ontology Language Overview.” Technical report, World Wide Web Consortium (W3C), February 2004. 1, 34, 36, 105, 141, 175

- [MH04b] Ralf Möller and Volker Haarslev. “RACER: Renamed ABox and Concept Expression Reasoner.”, 2004. <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>. 1, 2, 105, 136, 176
- [Mic95] Microsoft Corporation. “Internet Explorer.”, August 1995. 228
- [Mic02] Microsoft Corporation. “Visual Basic .NET.”, 2002. 230
- [Mic06] Microsoft Corporation. “Common Language Infrastructure (CLI).” Technical report, ECMA International, June 2006. 147, 170
- [Mic07a] Microsoft Corporation. “LINQ to XML.”, November 2007. 230
- [Mic07b] Microsoft Corporation. “Microsoft SQL Server.”, 2007. 109
- [Mic07c] Microsoft Corporation. “Microsoft Visual Studio.”, 2007. 131
- [Mic07d] Microsoft Corporation. “MSBuild.”, 2007. 303
- [Min81] Marvin Minsky. *Mind Design*, chapter A framework for representing knowledge, pp. 95–128. The MIT Press, 1981. A longer version appeared in *The Psychology of Computer Vision* (1975). 12
- [Mit07] Nilo Mitra. “SOAP Version 1.2 Part 0: Primer (Second Edition).” Technical report, World Wide Web Consortium (W3C), April 2007. 135
- [MM04] Frank Manola and Eric Miller. “RDF Primer.” Technical report, World Wide Web Consortium (W3C), February 2004. 1, 28, 105, 141
- [Mot06] Boris Motik. “KAON2.”, 2006. <http://kaon2.semanticweb.org>. 2, 135
- [MRJ95] Deborah L. McGuinness, Lori Alperin Resnick, and Charles Lee Isbell Jr. “Description logic in practice: A CLASSIC application.” In *16th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2045–2046, August, 1995. Morgan Kaufmann. 138
- [MS03] Erik Meijer and Wolfram Schulte. “Unifying tables, objects and documents.” In *Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL)*. Forschungszentrum Jülich GmbH, 2003. 223
- [MSD09] MSDN Library. “How to: Create Task List Comments.”, 2009. 307
- [MyS07] MySQL AB. “MySQL.”, 2007. <http://www.mysql.com>. 109
- [Neb90] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Springer-Verlag, New York, NY, USA, 1990. 17
- [Net94] Netscape Communications Corporation. “Netscape Navigator.”, December 1994. 228

- [Obj05a] Object Management Group (OMG). “MetaObject Facility.”, August 2005. <http://www.omg.org/mof/>. 3, 300
- [Obj05b] Object Management Group (OMG). “Ontology Definition Metamodel.”, 2005. <http://www.omg.org/ontology/>. 2, 300
- [Obj06] Object Management Group. “Object Constraint Language, Version 2.0.” Technical report, Object Management Group, May 2006. 282, 283
- [Obj09a] Object Management Group. “Unified Modeling Language, Infrastructure.” Technical report, Object Management Group, February 2009. 282
- [Obj09b] Object Management Group. “Unified Modeling Language, Superstructure.” Technical report, Object Management Group, February 2009. 282
- [Obj09c] Object Management Group (OMG). “CORBA.”, 2009. <http://www.corba.org/>. 135
- [Ope06] Open RDF. “Sesame RDF Database.”, 2006. <http://www.openrdf.org>. 136
- [OWL09] “OWL2XMI.”, 2009. <https://www.ohloh.net/p/owl2xmi>. 281
- [Paa07] Alexander Paar. “Zhi# – Programming Language Inherent Support for Ontologies.” In Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter, editors, *ateM '07: Proceedings of the 4th International Workshop on Software Language Engineering*, number 4/2007 in Mainzer Informatik-Berichte, pp. 165–181, Mainz, Germany, October 2007. Johannes Gutenberg Universität Mainz. Nashville, TN, USA. 178
- [Par05] Terence Parr. “Another tool for language recognition (ANTLR).”, 2005. <http://www.antlr.org>. 56, 60, 96
- [Pat98] Peter F. Patel-Schneider. “DLP.” In Enrico Franconi, Giuseppe De Giacomo, Robert M. MacGregor, Werner Nutt, and Christopher A. Welty, editors, *11th International Workshop on Description Logics (DL)*, volume 11 of *CEUR Workshop Proceedings*. CEUR-WS.org, June 1998. 18
- [Pel91] Christof Peltason. “The BACK system – an overview.” *SIGART Bulletin*, 2(3):114–119, June 1991. 17
- [Pel06] “Pellet.”, 2006. <http://pellet.owldl.com>. 1, 2, 18, 105, 138, 202, 265
- [PH06] Jeff Z. Pan and Ian Horrocks. “OWL-Eu: Adding customised datatypes into OWL.” *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(1):29–39, January 2006. 209

- [PHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. “OWL Web Ontology Language Semantics and Abstract Syntax.” Technical report, World Wide Web Consortium (W3C), February 2004. 36, 58
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, USA, 2002. 41, 50, 91, 153, 168
- [Plo81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. PhD thesis, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981. 44
- [Pos07] PostgreSQL Global Development Group. “PostgreSQL.”, 2007. <http://www.postgresql.org>. 109
- [PPC08] Colin Puleston, Bijan Parsia, James Cunningham, and Alan L. Rector. “Integrating object-oriented and ontological representations: A case study in Java and OWL.” In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *7th International Semantic Web Conference (ISWC)*, volume 5318 of *Lecture Notes in Computer Science*, pp. 130–145. Springer Verlag, 2008. 264
- [PR09] Alexander Paar and Jürgen Reuter. *Computers in the Human Interaction Loop*, chapter Ontological Modeling and Reasoning, pp. 325–340. Human-Computer Interaction Series. Springer Verlag London, April 2009. 37
- [Pro07] Protégé Wiki. “Ontology Bean Generator.”, 2007. <http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator>. 211
- [Pro08] Protégé Wiki. “Protégé OWL Library.” Technical report, Stanford University School of Medicine, 2008. [http://protegewiki.stanford.edu/index.php/Protege\\_Ontology\\_Library](http://protegewiki.stanford.edu/index.php/Protege_Ontology_Library). 1
- [PRS06] Alexander Paar, Jürgen Reuter, John Soldatos, Kostas Stamatis, and Lazaros Polymenakos. “A Formally Specified Ontology Management API as a Registry for Ubiquitous Computing Systems.” In Ilias Maglogiannis, Kostas Karpouzis, and Max Bramer, editors, *AIAI '06: Proceedings of the 3rd IFIP Conference on Artificial Intelligence Applications and Innovations*, volume 204/2006, pp. 137–146, Boston, MA, USA, June 2006. IFIP International Federation for Information Processing, Springer-Verlag. Athens, Greece. 106, 121
- [PRS09] Alexander Paar, Jürgen Reuter, John Soldatos, Kostas Stamatis, and Lazaros Polymenakos. “A Formally Specified Ontology Management API as a Registry for Ubiquitous Computing Systems.” *Applied Intelligence*, **30**(1):37–46, February 2009. 96, 139

- [PS93] Peter F. Patel-Schneider and Bill Swartout. “Description-Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort.” Technical report, AI Principles Research Department, AT&T Bell Laboratories, November 1993. 137
- [Qui67] M. Ross Quillian. “Word concepts: A theory and simulation of some basic capabilities.” *Behavioral Science*, **12**:410–430, 1967. 12
- [Rou07] Vlad Roubtsov. “EMMA: a free Java code coverage tool.”, 2007. <http://emma.sourceforge.net>. 131, 139
- [Rum87] James Rumbaugh. “Relations as semantic constructs in an object-oriented language.” In *2nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 466–481, New York, NY, USA, 1987. ACM Press. 234
- [Sch24] Moses Ilyich Schönfinkel. “Über die Bausteine der mathematischen Logik.” *Mathematische Annalen*, **92**(3 – 4):305–316, 1924. 46
- [Sch91] Klaus Schild. “A correspondence theory for terminological logics: Preliminary report.” In John Mylopoulos and Raymond Reiter, editors, *12th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 466–471. Morgan Kaufmann, August 1991. 18
- [Sch02] Guus Schreiber. “A UML Presentation Syntax for OWL Lite.” Technical report, Vrije Universiteit Amsterdam, April 2002. 36
- [SL05] Henrike Schuhart and Volker Linnemann. “Valid updates for persistent XML objects.” In Gottfried Vossen, Frank Leymann, Peter C. Lockemann, and Wolfried Stucky, editors, *11te GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, volume 65 of *Lecture Notes in Informatics*, pp. 245–264, Bonn, Germany, March 2005. Gesellschaft für Informatik. 170
- [Sow91] John F. Sowa, editor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, May 1991. 12
- [SPG07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. “Pellet: A practical OWL-DL reasoner.” *Web Semantics: Science, Services and Agents on the World Wide Web*, **5**(2):51–53, June 2007. 18
- [Spi01] J. Michael Spivey. “The Z Notation: A Reference Manual (Second Edition).” Technical report, University of Oxford, Programming Research Group, 2001. 121

- [SR05] Kurt Stirewalt and Spencer Rugaber. “Automated invariant maintenance via OCL compilation.” In Lionel C. Briand and Clay Williams, editors, *8th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 3713 of *Lecture Notes in Computer Science*, pp. 616–632, Berlin / Heidelberg, October 2005. Springer Verlag. 285
- [SS91] Manfred Schmidt-Schauß and Gert Smolka. “Attributive concept descriptions with complements.” *Artificial Intelligence*, **48**(1):1–26, February 1991. 17, 19, 243, 250
- [ST06] Jeremy G. Siek and Walid Taha. “Gradual typing for functional languages.” In Mark W. Bailey, editor, *7th Workshop on Scheme and Functional Programming (Scheme)*, volume 41 of *ACM SIGPLAN Notices*, New York, NY, USA, September 2006. ACM Press. 214, 298
- [Sta06] Stanford University School of Medicine. “Protégé knowledge acquisition system.”, 2006. <http://protege.stanford.edu>. 1, 105, 136, 176, 264
- [Sun06a] Sun Microsystems. “Java Remote Method Invocation (Java RMI).”, 2006. 135
- [Sun06b] Sun Microsystems. “JDBC technology.”, 2006. 136
- [TBM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. “XML Schema Part 1: Structures Second Edition.” Technical report, World Wide Web Consortium (W3C), October 2004. 26
- [Tel07] Telecom Italia. “Java Agent Development Framework (JADE).”, 2007. <http://jade.cselt.it/>. 211
- [TF08] Sam Tobin-Hochstadt and Matthias Felleisen. “The design and implementation of typed scheme.” *ACM SIGPLAN Notices*, **43**(1):395–406, January 2008. 214, 298
- [Tha90] Satish R. Thatte. “Quasi-static typing.” In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 367–381, New York, NY, USA, January 1990. ACM Press. 214, 298
- [The98] The Document Object Model Working Group. “Document Object Model (DOM) Level 1 Specification.” Technical report, World Wide Web Consortium (W3C), October 1998. 222, 228
- [The00] The Defense Advanced Research Projects Agency. “The DARPA Agent Markup Language.”, 2000. 34
- [Tho03] Dave Thomas. “The impedance imperative: Tuples + objects + infosets = too much stuff!” *Journal of Object Technology*, **2**(5):7–12, September-October 2003. 223

- [V06] Max Völkel. “RDFReactor – from ontologies to programmatic data access.” In *1st Jena User Conference (JUC)*. HP Bristol, May 2006. 212
- [WC93] Daniel Weise and Roger F. Crew. “Programmable syntax macros.” In *15th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 156–165, 1993. 77
- [Woo95] Derick Wood. “Standard Generalized Markup Language: Mathematical and philosophical issues.” In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pp. 344–365. Springer Verlag, 1995. 25
- [Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005. 267
- [ZW97] Amy Moormann Zaremski and Jeannette M. Wing. “Specification matching of software components.” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **6**(4):333–369, October 1997. 91