A STATICALLY TYPED LOGIC CONTEXT QUERY LANGUAGE WITH PARAMETRIC POLYMORPHISM AND SUBTYPING

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Tobias Rho

aus Geldern

Bonn, 2011

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

ERSTGUTACHTER: Prof. Dr. Armin B. Cremers, Bonn
ZWEITGUTACHTER: Prof. Dr. Robert Hirschfeld, Potsdam

TAG DER PROMOTION: 30. April 2012
ERSCHEINUNGSJAHR: 2012

ABSTRACT

The objective of this thesis is programming language support for context-sensitive program adaptations. Driven by the requirements for context-aware adaptation languages, a statically typed Object-oriented logic Context Query Language (OCQL) was developed, which is suitable for integration with adaptation languages based on the Java type system.

The ambient information considered in context-aware applications often originates from several, potentially distributed sources. OCQL employs the Semantic Web-language RDF Schema to structure and combine distributed context information.

OCQL offers parametric polymorphism, subtyping, and a fixed set of meta-predicates. Its type system is based on mode analysis and a subset of Java Generics. For this reason a mode-inference approach for normal logic programs that considers variable aliasing and sharing was extended to cover all-solution predicates.

OCQL is complemented by a service-oriented context-management infrastructure that supports the integration of OCQL with runtime adaptation approaches. The applicability of the language and its infrastructure were demonstrated with the context-aware aspect language CSLogicAJ. CSLogicAJ aspects encapsulate context-aware behavior and define in which contextual situation and program execution state the behavior is woven into the running program.

The thesis concludes with a case study analyzing how runtime adaptation of mobile applications can be supported by pure object-, service- and context-aware aspect-orientation. Our study has shown that CSLogicAJ can improve the modularization of context-aware applications and reduce anticipation of runtime adaptations when compared to other approaches.

ACKNOWLEDGMENTS

First of all, I would like to thank my adviser Armin Cremers for supervising this thesis and guiding me over the last years.

I want to thank Mark von Zeschau for putting so much effort into the Ditrios framework, which has been an important building block for the implementation of this thesis. My colleagues Holger Mügge, Daniel Speicher, and Pascal Bihler had an important impact on the direction of my work in the course of joint work in the project *context-sensitive intelligence*.

Special thanks go to Malte Appeltauer, who accompanied my work from generic aspect languages to context-sensitive modularization approaches. It has always been productive and enjoyable to work with him. Robert Hirschfeld and his group at the Hasso Plattner Institute have given me the impulse to generalize my approach from aspect-orientation to general adaptation languages.

Stephan Lerche implemented the initial version of the context-management system and influenced the context query language significantly with his ideas.

I am also indebted to a number of people who have helped proof-reading my thesis. Lunjin Lu commented on an early draft of chapter 3, Jan Wielemaker gave in-depth comments concerning the semantics and standardization of Prolog. I want to thank Jan-Paul Imhoff, Günter Kniesel, Daniel Morales, Jan Nonnen, Daniel Speicher, and Josh Ward for their helpful feedback in the finalization phase of this work.

Finally I want to thank my family and friends for encouraging and supporting me during this period of life.

CONTENTS

1 INTRODUCTION 1
1.1 Motivation 1
1.2 Definition and Representation of Context 3
1.3 Running Example 4
1.3.1 Realization Considerations 5
1.4 Requirements for Context-based Adaptations 6
1.4.1 Query Language Properties 9
1.5 Contributions 9
1.6 Thesis Outline 10

I STATE OF THE ART 11
2 BASIC CONCEPTS 13
2.1 OSGi 13
2.1.1 OSGi Shortcomings 13
2.1.2 LDAP Attributes and Filters 15
2.2 Semantic Web 16
2.2.1 RDF Schema 17
2.2.2 RDFS Entailment 17
2.2.3 Ontology Web Language 18
2.2.4 Open World vs. Closed World Reasoning 18
2.3 Prolog 19
2.3.1 Operational Semantics 20
2.3.2 Module Concept 21
2.3.3 Meta Predicates 21
2.3.4 Prolog Syntax 22
2.4 Typed Logic Programming 22
2.4.1 Instantiation Modes 24
2.4.2 Type System based on Restricted Modes 25
2.4.3 Order-sorted Unification 25
2.5 Mode Analysis 26
2.5.1 Abstract Interpretation 26
2.5.2 Sound Approximation 28
2.5.3 Chosen Mode Inference Approach 28
2.6 Aspect-Oriented Programming 30
2.6.1 Join Points and Pointcuts 30
2.6.2 Advice 32
2.6.3 Inter-type Declarations 33
2.6.4 Aspects 33

II APPROACH 35
3 FINDALL-EXTENDED MODE ANALYSIS 37
3.1 Mode Analysis Approach by Lu 37
3.1.1 Program Graph 40
3.1.2 Concrete Semantics 41
3.1.3 Collecting Semantics 42
3.1.4 Abstract Semantics 43
3.2 Findall-extended Mode Analysis 43
3.2.1 Concrete Semantics 45
3.2.2 Collecting Semantics 47
3.2.3 Abstract Semantics 47

3.2.4 Example 50
3.2.5 Computational Complexity 51
3.2.6 Application to Further All-Solutions Predicates 52
3.3 Summary 53
4 OBJECT-ORIENTED LOGIC CONTEXT QUERY LANGUAGE 55
4.1 RDF Schema - Java Mapping 55
4.1.1 Namespace Binding 57
4.1.2 Class and Property Mapping 57
4.1.3 RDF Reification 63
4.2 Object-oriented logic Context Query Language 64
4.2.1 Arrays and Tuples 66
4.2.2 Predicates 67
4.2.3 Built-in Predicates 67
4.2.4 Generic Types & Mapping Predicates 67
4.2.5 Context History 70
4.2.6 Querying Context Sources 70
4.2.7 Type Checks & Casts 71
4.3 Type Checking 71
4.3.1 Overview 71
4.3.2 Pre-Mode Analysis Type Checking 73
4.3.3 Type Inference for Generic Predicates 79
4.3.4 Translation To Prolog 81
4.3.5 Mode Analysis and Final Type Checking 88
4.3.6 Mode Checking for Other Unsafe Expressions 90
4.4 Summary 90
5 CONTEXT MANAGEMENT INFRASTRUCTURE AND SERVICE ASPECTS 91
5.1 Context Management Infrastructure 91
5.1.1 Context Listeners 93
5.1.2 OCQL Compilation 93
5.1.3 Requesting Context 95
5.1.4 Queries and Language Integration 96
5.2 Query Context Sources 99
5.3 Service Discovery and Interception 99
5.3.1 Proxy Indirection 100
5.3.2 Service Adaptation 101
5.3.3 Transaction-awareness 102
5.4 Context-Sensitive Service Aspects 103
5.4.1 Context Pointcut Language 103
5.4.2 Service Pointcut 105
5.4.3 Asynchronous Onchange Advice 105
5.4.4 First-Class Join Point 106
5.4.5 Referring to Context Sources 106
5.5 Music Player Example Revisited 107
5.6 Reconsidering Requirements 110
5.7 Summary 110

III IMPLEMENTATION AND EVALUATION 113
6 IMPLEMENTATION 115
6.1 Mode Analysis 115
6.2 Context Management System 115
6.2.1 OCQL Parsing Framework 116
6.3 CSLogicAJ 118
6.3.1 Compiler 118

6.3.2 Extensible OCQL/Pointcut Parser 120
6.3.3 Static Analysis 121
6.3.4 Mapping Advice Constructs to Java Source Code 122
6.3.5 Integrated Development Environment 123
6.3.6 Realization of Query Context Sources 124
6.4 Summary 124
7 EVALUATION 125
7.1 Programming for Context-based Adaptability - A Case Study 125
7.1.1 Requirements Elicitation 126
7.1.2 General Requirements for Context-Sensitive Adaptivity 127
7.1.3 Pure Object-Orientation - Patterns for Adaptivity 127
7.1.4 SOA and Object-Orientation - Patterns for Adaptivity 128
7.1.5 SOA and Aspect-Orientation - Patterns for Adaptivity 130
7.1.6 Summary 133
7.2 JCop Query Library 133
7.2.1 Overview 133
7.2.2 Example 134
7.2.3 Summary 135
7.3 Intercepted Service Call Benchmark 135
7.4 Summary 136

IV RELATED WORK, CONCLUSIONS, AND FUTURE WORK 139
8 RELATED WORK 141
8.1 Context Query languages 141
8.1.1 SPARQL 141
8.1.2 SWRL 142
8.1.3 MUSIC CQL 143
8.1.4 F-Logic / Flora-2 143
8.1.5 Prova 2 143
8.2 Context-management Systems 144
8.2.1 Context-aware Aspect-oriented Programming 144
8.2.2 Dynamic Component-based Aspect-oriented Programming 145
9 CONCLUSIONS AND FUTURE WORK 147
9.1 Conclusions 147
9.2 Future Work 148
9.2.1 Weaving Optimizations 148
9.2.2 OWL Support 148
9.2.3 Typed Java Library 149

V APPENDIX 151
A APPENDIX 153
A.1 XML Primitive Type Mapping 153
A.2 XSD Built-in Type Hierarchy 154
A.2.1 Lunjin Lu - Abstract Unification Algorithm 155
A.2.2 Implementation of Array Index Access 157
A.2.3 Predefined Predicates and Arithmetic Expressions 157

BIBLIOGRAPHY 161

LIST OF FIGURES

Figure 1.1 Context-sensitive adaptation of mobile applications. Adapted bookmarks, notes and email applications provide prompt access to currently relevant documents. 1
Figure 1.2 Comparison of a monolithic and modularized implementation. The modularized variant on the right is based on a component framework, a context management system, and a context-sensitive adaptation. 2
Figure 1.3 Basic music player and contact application 5
Figure 1.4 Context-sensitive music player 6
Figure 2.1 OSGi bundle lifecycle 14
Figure 2.2 OSGi service tracker 14
Figure 2.3 The RDF triple 16
Figure 2.4 Steps to abstract interpretation 27
Figure 2.5 Illustration of aspect weaving. An aspect selects a set of join points in the base classes of a program and weaves additional code at these points. 31
Figure 2.6 JoinPoint interface 33
Figure 3.1 ∇ operator, over-estimates the mode of two unified terms. Adapted from [130, Figure 5.1]. 39
Figure 3.2 Call graph example 41
Figure 3.3 Extended call graph example 45
Figure 4.1 RDFS classes considered subsets 57
Figure 4.2 Hierarchy of mapped Java objects 58
Figure 4.3 Exemplary mapping of the rdfs class `Contact` 59
Figure 4.4 Cycle in case of naive powerset mapping. The dotted lines are subtype relationships introduced by $E_{\wp}$. 60
Figure 4.5 Illustration of RDF instance to JI_e mapping 62
Figure 4.6 RDF reification `Statement` interface 63
Figure 4.7 EBNF of OCQL expressions 64
Figure 4.8 Syntax of the `findall` predicate 67
Figure 4.9 EBNF of generic predicates 68
Figure 4.10 Type conformance 75
Figure 4.11 Typing rules - core 77
Figure 4.12 Typing rules for `select`, `one`, and `findall` 78
Figure 4.13 Type checking rules - first pass 78
Figure 4.14 Our simplification of the Java type parameter inference algorithm [88, 15.12.2.7] 80
Figure 4.15 Final typing rules for delayed checking 89
Figure 4.16 Final typing rules for findall calls with a single variable template 90
Figure 5.1 Overview of the context management system 91
Figure 5.2 The IContextProvider interface 92
Figure 5.3 Context data snapshots 92
Figure 5.4 Requesting context - dynamic approach 93
Figure 5.5 The query and listener interfaces 94
Figure 5.6 The compilation interface 94
Figure 5.7 Mapping from concrete to abstract syntax and to a term-serialized form 95
Figure 5.8 Requesting context - dynamic approach 96

Figure 5.9 Three different alternatives of integrating OCQL with a host language 97
Figure 5.10 Generic context class IContext 99
Figure 5.11 Lastfm context source 100
Figure 5.12 The client service interface, Ditrios' entry point for service searching and tracking 100
Figure 5.13 Ditrios service lookup workflow 101
Figure 5.14 Ditrios Invocation Handler 102
Figure 5.15 Services are transaction-aware 102
Figure 5.16 Syntax of CSLogicAJ's generalized advice construct. 104
Figure 5.17 EBNF of Primitive Pointcuts 105
Figure 5.18 Service level logging call pointcut. 105
Figure 5.19 Requesting Context - Static Approach 106
Figure 5.20 Context-Sensitive Music Player, modularized by an aspect 107
Figure 5.21 Binding different pointcuts 107
Figure 5.22 Context Sensitive Media Player Adaptation 108
Figure 5.23 The sharedArtist pointcut binds artist by collecting all artis from nearby contacts with the findall meta-call. 108
Figure 5.24 Last.fm onchange advice 109
Figure 5.25 Intercepting service events 109
Figure 5.26 Filtered Address Book Entries 109
Figure 5.27 Around Advice Example with Proceed Statement 110
Figure 6.1 Compilation of a Java language extension with embedded OCQL 116
Figure 6.2 Prolog clause generation overview 117
Figure 6.3 Ditrios service runtime model 119
Figure 6.4 JTransformer Prolog AST 119
Figure 6.5 Ditrios interceptor interface 122
Figure 6.6 CSLogicAJ's integrated development environment 123
Figure 7.1 Simple document management tools for mail, minutes and bookmarks 125
Figure 7.2 Combining services leads to a beneficial adaptation 126
Figure 7.3 Adapted tools provide prompt access to currently relevant documents. 127
Figure 7.4 Strategy Pattern in a service-oriented architecture 129
Figure 7.5 Usage of the Strategy Pattern in PimPro for the view categorization 129
Figure 7.6 The anticipated decorator pattern for the bookmark tree decoration 130
Figure 7.7 Aspect-oriented Strategy Pattern based on user preferences 131
Figure 7.8 Dynamic strategy change 132
Figure 7.9 Aspect-oriented decorator pattern for the bookmark tree decoration 132
Figure 7.10 Bookmarks Flattening Aspect 133
Figure 7.11 Cop modularization approach; Figure adapted from [15] 134
Figure 7.12 Implementation of the ToDo application using JCop's query library. 137
Figure 7.13 Cached service call compared with normal calls 138

INTRODUCTION

1.1 MOTIVATION

Context-awareness is essential for a large number of today's mobile applications. Location-awareness, in particular, has become an integral feature of many smartphone applications. An example for that is a bus schedule application that determines the current location of the user and shows the arrival times at nearby bus stops.

Still, context is more than the location of the device. By combining the data from different context sources, the situation can be analyzed and facilitated to adapt the application to the user's current needs. For example, an email client can filter emails relevant to the current situation such as a meeting, or a music player can arrange playlists according to the musical taste of the people in a car.

In order to analyze the user's situation and adapt themselves to the environment, applications need to access the sensors of the mobile device[1], the state of other applications, user preferences and even external data-sources, such as social networks (e.g., Google Latitude, Facebook, last.fm, Twitter, etc).

In most applications context retrieval, analysis, and modifications are typically repetitively implemented in every application leading to a lot of small, specialized "context management systems" embedded within each application. This has several drawbacks. First, a lot of development time goes into implementing and maintaining all these context-management systems. *Derived context*, such as the history of visited locations, is calculated and stored by several applications, thereby repeatedly consuming computing time and memory, both of which are limited resources on mobile devices. Next, there might be generic adaptations, spanning several applications that need to be repetitively implemented in each application, which again leads to more development and maintenance effort. Moreover, the tangling of context-aware adaptation and core features leads to complex designs, which can result in badly modularized code that is prone to errors and high testing complexity.

Figure 1.1 exemplary shows three context-aware mobile applications in which the repetition becomes apparent. A mobile phone with a notes, a bookmarks, and an email application is brought to a trade fair. The trade fair offers a service which highlights and re-orders documents on mobile applications relevant to fair stands nearby. All three

1 orientation, light, proximity, accelerometer, temperature, etc.

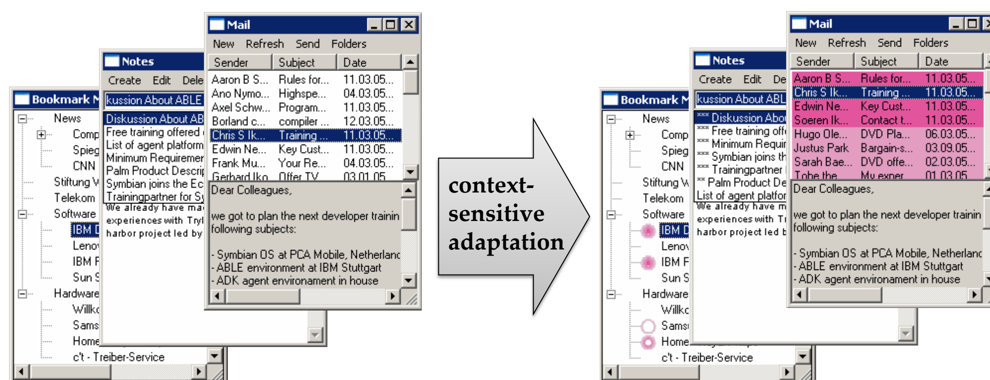


Figure 1.1: Context-sensitive adaptation of mobile applications. Adapted bookmarks, notes and email applications provide prompt access to currently relevant documents.

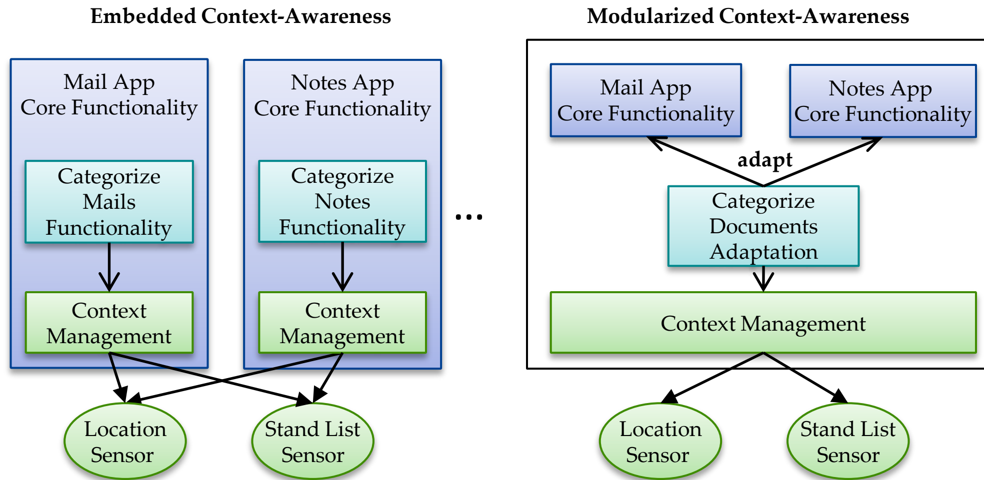


Figure 1.2: Comparison of a monolithic and modularized implementation. The modularized variant on the right is based on a component framework, a context management system, and a context-sensitive adaptation.

applications need the same context-information (nearby stands, keywords relevant to the stands) and similar, additional functionality (highlight entries, re-order the application's documents)[2].

Figure 1.2 illustrates the problem from an architectural point of view and gives a glimpse of our solution. The dilemma as described is depicted on the left-hand side. Every application contains core functionality and additional categorization functionality, which reorders and highlights documents based on context information. All applications use their own context-management the categorization functionality is an integral part of the whole application.

On the right, we see a modularized solution in which the document categorization is extracted from the applications and encapsulated in an adaptation module. Only one context-management system is deployed, which can also be used by other adaptation modules. And the adaptation module can be added and removed while the applications are running, which is especially useful in the given scenario, where the functionality is only useful while the user stays in the area of the trade fair.

This leads to the question of how context-sensitive program adaptation and the core functionality of an application can be separated. Software architecture research has approached this concern with *dynamism* or *self-adaptivity architectures* [154]. These are software architectures that allow component reconfiguration at runtime. Bradley et al. [42] and Medvidovic et al. [138] have surveyed a number of architectural approaches that apply graph rewriting rules, first-order logic, π-calculus or architecture modification languages to realize the runtime adaptation. Most systems follow a four step approach of component graph reconfiguration [42]:

1. An internal component triggers a change based on a state change, events, etc.
2. An appropriate change is selected, whether pre-defined or calculated at runtime.
3. An implementation realizes the component graph modification.
4. The modification is assessed, e.g., analyzed for consistency.

What all these approaches have in common is that they are applied on a set of components and connectors, and that dynamic reconfiguration is not an intrinsic part of the

2 The example is fully elaborated on in section 7.1.

architecture and needs the mentioned extensions to support dynamism. Components are typically not prepared for dynamic adaptation and often use stateful calls to other components, which makes replacing of components a complex task since the state has to be transferred to the other, newly introduced component.

Lately, service-oriented architectures (SOA) [70] have proven to be a good conceptual base for dynamic reconfiguration. In a SOA, software components are intrinsically loosely coupled. *Service consumers* query a *registry* for *providers* offering a service of certain properties. A fixed binding between a consumer and a provider is not intended. Services storing consumer specific state (as in classic component architectures) are possible, but they are typically avoided. SOA architectural patterns such as *Stateful Services* [71], decouple state of service usage from a service performing a concrete task. This leads to low coupling between consumers and producers as well as effortless switching between different service providers at runtime. Tsai et al. [200] state that

> "SOA has provided a new direction for software architecture study, where the architecture is determined at runtime and architecture can be dynamically changed at runtime to meet the new software requirements."

To implement context-aware applications based on a SOA, they need to be decomposed into services. Applications are then built by composing these services and configuring them at runtime based on context-information. Truong and Dustdar [199] surveyed a range of systems following this approach and determined that adaptation based on context information is applied in a range of use cases:

> "*Service selection and task adaptation*: context information is mostly used to select the most suitable service and task to perform actions, given a situation. One example of this purpose is how to use context for service provisioning." Further it is used in *"Security and privacy control"*, *"Communication adaptation"*, and *"Content adaptation*: context information is used to adapt content resulting from a request and to return the content in a form suitable to the context of the requester. One example is to adapt content in mobile Web services."

The adaptation/reconfiguration in the considered systems are carried out by *actuators* that configure service properties or orchestrate the service configuration.

Numerous researchers argue that there needs to be additional programming language support for context-aware program adaptation. Hirschfeld et al. [107] have proposed *Context-Oriented Programming* (COP), a programming technique realizing context-dependent adaptation based on a layer concept. Layers enrich existing objects with additional functionality and state and they are context-sensitively (de-)activated. The context of a running program is the whole perceivable program state, which may make use of further technologies or frameworks to describe this context.

Frei [80], Fuentes and Jimenez [82], Prezerakos et al. [80, 82, 174, 165] and we [174] have shown that *aspect-orientated programming* is a good means for dynamic service reconfiguration. The main stream aspect-oriented programming (AOP) languages [76] offer constructs to describe where additional functionality should be added to an existing program in a declarative manner[3]. The existing program is typically unaware (*oblivious)* [75] of the concern implemented by the aspect, meaning it does not prepare for modifications carried out by the aspect.

1.2 DEFINITION AND REPRESENTATION OF CONTEXT

The term *context* has been widely used in *ubiquitous computing* literature to describe the physical and virtual environment of a person or device, starting with Weiser's influential "The computer for the 21st century" [207]. For the *Context Toolkit,* Dey [65] formulated a

3 Section 2.6 covers aspect-orientation in detail.

definition based on previous descriptions by Schilit [179] and Pascoe [159] with the aim of enhancing practical applicability:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [65]

Hirschfeld et. al [107] have given an even more technical definition of context in the field of COP:

> "Any information which is computationally accessible may form part of the context upon which behavioral variations depend."

We consider the combination of these context definitions suitable for our approach. Context is perceived through a range of *context sources* [169] describing the situation of context entities. The information from different context sources is aggregated so as to form the basis for further analysis. The context sources can represent program state (e.g., contacts stored in the local address book), local sensors (e.g., a gyroscope) or remote services offering access to a broad range of information, such as the current weather or data from social (media) networks.

The context information needs to be modeled, aggregated, and analyzed to trigger adaptations. A range of context management systems have been proposed to manage context sources and information, such as *Semantic Space* [204], *WildCAT* [61], and *MUSIC* context-management [169]. Baldauf et al. [24] surveyed eight further systems. A majority of them are based on the *Semantic Web* stack [34] (also see Section 2.2), which in turn is build on the resource description framework (RDF) [121], the W3C standard data interchange format for the Web. Semantic Web-based context models are defined in (distributed) schemas, reasoning deduces implicit knowledge from given context data, and query languages (e.g., SPARQL) allow for further context analysis. Once a certain context situation is detected, *actuators* are activated and execute an action. We follow this approach and build context modeling on *RDF Schema*[45], a means to model simple ontologies in RDF.

1.3 RUNNING EXAMPLE

The following example is used throughout this thesis to illustrate the requirements of context-aware adaptation of mobile applications. Consider the user of a smartphone. We assume that he and most of his contacts are registered to the social networks *last.fm* and *latitude* and that the account names are integrated into the phone's contact application.

The last.fm [4] service is a music stream service, which creates music playlists based on the user's preferred artists and songs. Furthermore it is able to analyze similarities between two users interests and provide lists of equally liked songs and artists. Latitude [5], on the other hand, is a localization service, which shows the current position of contacts, which have agreed to make their current position visible to this user. The sentences in the following scenario are marked with enumeration annotations for later reference:

The user is driving a car and is taking a number of friends with him. He connects the smartphone to the car's audio system[4] (b) and starts a music application. The user has installed an extension to his music player, which allows context-sensitive selection of songs based on his last.fm profile. The extension detects the startup of the player (a) and asks if the music should be selected based on the group of people nearby (b). Based on the profile of the user and his friends a new playlist is created and played (d) and updated once the group of nearby people changes (e). In case some of the last.fm

4 For example, by using it as an UPnP media render [113]

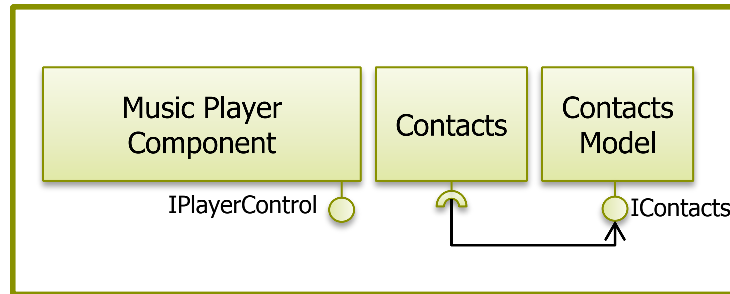


Figure 1.3: Basic music player and contact application

accounts are unknown, the user is offered to edit those contacts in the phone's address book application (c).

The last.fm extension is independent from the music player. It can be installed from an application store after the music player was installed (f).

To realize such a context-aware adaptation of an existing music player, a number of requirements have to be fulfilled:

(a) The startup of the player must be *observed* by the adaptation. It must intercept the startup to ask the user if the extension should be activated.

(b) The extension must *access* and *aggregate* different kinds of context information:
- audio system connection status (docking station)
- user's contacts (contact application)
- nearby contacts (latitude Web service)
- profile comparison (last.fm Web service)

(c) The model of the contact application must be *filtered* to show only nearby contacts without last.fm accounts.

(d) There must be a means (e.g., a service interface) to configure the player's playlist.

(e) It must *react to* context *changes* once the set of nearby contacts change.

(f) It should enable the user to (de)activate the last.fm extension at runtime.

1.3.1 *Realization Considerations*

Such a functionality could be hard-coded into a music player and offered as a stand-alone mobile application. As discussed above, we are striving for a modularized solution - an existing music player is extended by a context-sensitive adaptation, which encapsulates the last.fm-based playlist generation.

We assume that the device's application environment is based on a component-oriented architecture and the music player manages and plays music files stored on the device. A simple contact application is deployed on the device, which is modularized into a contact GUI[5] component and a model component.

Both applications are illustrated in Figure 1.3. The components have the following responsibilities:

5 Graphical User Interface

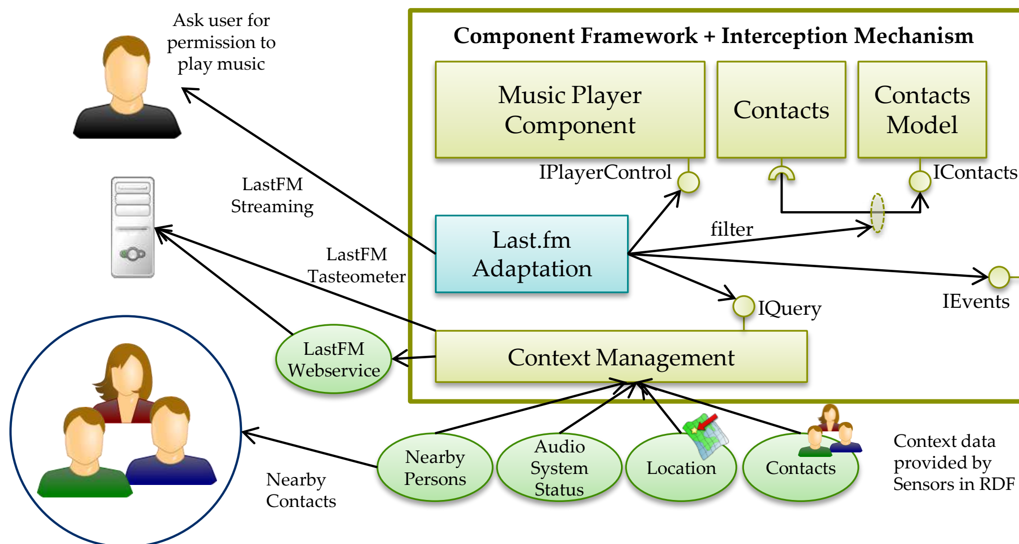


Figure 1.4: Context-sensitive music player

The Music Player Component encapsulates the music player. It offers the interface `IPlayerControl` for creating playlists from local or remote (URL) files as well as starting, pausing, and stopping the player.

The Contacts Component is the GUI component of a contacts application. It shows a list of contact entries, containing the name, address fields and custom fields for social network (login) names.

The Contacts Model Component stores the contacts in a database and offers access to the model via an interface. The GUI component uses it to retrieve the list of all contacts and the contact entry details.

Figure 1.4 illustrates a realization of such an adaptation based on a given component architecture and the given components. Again we annotate sentences with the related requirements from the last section. The *Last.fm Adaptation* is realized as a component, which can be installed, started, stopped and deleted while the system is running (f). To filter message calls (a, c) we need to intercept the communication between components, so we assume an interception mechanism is in place.

Framework events (a) can be monitored via listeners, represented here as the interface `IEvents`. The aggregation and querying (b) of context information is encapsulated in a generic context management component that gathers data from local and remote sources and offers means to attach listeners for context changes (e). The interfaces to components are modeled as services (d) to make them accessible for other components.

1.4 REQUIREMENTS FOR CONTEXT-BASED ADAPTATIONS

Although the previous section highlighted a number of important requirements for context-based adaptation, it is not a thorough analysis of the technical requirements for context-awareness development support in general. Programming language and framework support for context-aware applications has been studied by a large number of researchers. The surveys by Baldauf et al. [24] and Truong et al. [199] give an overview of context-awareness frameworks, [107, 177, 197, 82, 64, 87, 169] have argued for the need of explicit language constructs to simplify the development of context-awareness features. The EU project MUSIC[6][169] has gathered a number of common requirements

6 An acronym for "Self-adapting applications for Mobile USers In ubiquitous Computing environments"

for context-query languages (CQL) and context-oriented frameworks. We will take these requirements as a basis for our approach. Reichle et al. [169] state that context-query languages should provide means to

1. retrieve *sets of elements*, such as the persons in a room;
2. specify *filters and conditions* (queries) on contexts so as to select contexts by their properties and qualities;
3. query context *meta data*, such as the accuracy of a context source;
4. combine elementary conditions with *logic operators* in order to analyze complex context situations;
5. allow *subscriptions on asynchronous* context change events so as to react to context changes;
6. aggregate context, for example to find the best location sensor or the average of a network bandwidth;
7. access the context history, for example to calculate the average network bandwidth from the last ten bandwidth measurements;
8. have loosely coupled context queries and sources;
9. have context sources residing on the device or be remotely connected.

The requirements 1 - 6 are supported by the requirements (a) - (f) elicited from the music player example. Requirement 8 and 9 go back to Perich et al. [161], who characterize context management systems as mobile distributed databases [68]. Mobile databases are not necessarily located on a single device and the availability of context sources is not guaranteed. A context source may therefore be replaced by another context source providing similar data, transparent to the using clients. For this reason a context query should generally not be linked to a fixed context source, but rather only specify the requested context information.

In addition to meeting these common context-query language requirements, the music player example contributes to the following requirements. The component framework should

10. support the interception of service communication, to allow filtering and replacing of services;
11. support interception of infrastructure events and access to the component management to start and stop existing components; and
12. allow dynamic activation and deactivation of adaptations.

Runtime adaptation comes in different levels of granularity. Here we will divide them into two different kinds: *controlled adaptation* and *unrestricted adaptation.* The first level assumes an abstraction layer that provides well-defined points for adaptation. An example is the well-known software plug-in concept [196], which offers a fixed API for customizations. Plug-in concepts are typically specialized for a concrete application. These applications explicitly open parts of their functionality for extensions.

A number of AOP approaches operating on a similar level of abstraction have been proposed. Aspect-oriented software development (AOSD) [77][7] focuses on the modularization of *crosscutting concern*s. Crosscutting concerns are aspects of a program that are scattered throughout the program. The basis for most AOP approaches is a *join point model* [118], which defines points in a program flow (called "join points") can be modified

7 see also Section 2.6

by inserting additional code before, after or around the code. Typical join points are method calls or field accesses. In general, these join points are not further restricted. AOP approaches for component frameworks (e.g., Jadabs [80] and to some extent Jasco [194]) restrict the join point to the component interface level. Only method calls on component interfaces can be adapted. Gudmundson and Kiczales [90] have proposed a similar approach on the programming-language level. So called *pointcut interfaces*[8] provide an abstraction layer between aspects and program base code, ensuring that only join points that are meant to be adapted are visible to aspects. Furthermore Fuentes and Jimenez [82] have presented an approach on architecture-level that encapsulates context-aware functionality with aspect-orientation and semantic web technology. They use a library/framework approach instead of a programming language-level integration.

In contrast *unrestricted adaptation* allows the adaptation of arbitrary program code. Dynamic AOP approaches such as JAC [160] and PROSE [163], allow runtime weaving of aspects at arbitrary join points in the program. Dynamic software update approaches such as Gilgul [56] or JavAdaptor [168], go even further and allow for arbitrary runtime patches of running applications.

Also AOP approaches for dynamically-typed languages, such as AspectL (Lisp) [57] and AspectS (Smalltalk) [104], and reflective languages such as Newspeak [41] fall into this category.

We decided to use a *controlled adaptation* approach and have applied AOP on service interface level. We apply this restriction for several reasons. First, in a SOA environment, the concrete implementation of services are not necessarily on the local system, and its core logic might be written in a different language running on a different infrastructure. Likewise even if the service implementation is available there may still be different uses of the same service from the local or a remote system which are in a different contexts. This is further supported by the Parnas' *information hiding principle* [156] which postulates that an evolvable design should provide functionality through a controlled interface and hide the concrete implementation details. These observations result in the last set of requirements:

11. considering components as black-boxes, according to which the implementation of the components may be distributed or written in a different language;

12. making the query language statically type safe. In other words, in a scenario of dynamic reconfigurations and changing contexts, the aspect language should check for potential ill-use of types already at compile-time. We will see later on (Section 2.4.3) that this further reduces the number of runtime-type tests necessary compared to an untyped query language.

13. including support for distributed context schemas. For example since the last.fm username of a contact is unlikely to be predefined in a common contact application, other context sources should be able to add properties to existing types.

Although we have good reasons to use the blackbox approach, it also has its drawbacks. Should components be coarse-grained, a great part the program's implementation is not accessible for adaptations. In [142], we discussed how components could be further opened up for controlled adaptation by exposing parts of the internal structure as explicit adaptation points. By annotating essential parts of a component's structure, such as filter and sorting hooks of GUI widgets, they are exposed to adaptation code via an interface of the enclosing component. This approach is orthogonal to the concepts presented in this thesis and can therefore be easily combined, thereby reducing this limitation of black box components.

8 Pointcuts is an AOP term for an expression which selects events in the program flow.

1.4.1 *Query Language Properties*

AspectJ and most derived aspect languages use logic expressions (called *pointcuts*) to select the join points effected by an aspect. We decided to develop a query language based on logic programming so as to allow a natural integration with pointcut languages. The type system of the language must support *subtyping*, since the Semantic Web languages are inherently based on concept taxonomies.

Queries typically make use of generic operations on lists such as sorting, aggregation, selection and projection. Modern static type systems typically employ *parametric polymorphism* to define generically typed operations. Parametric polymorphism steams from ML-inspired functional languages [95], and has been applied to generic types in object-oriented languages such as Java [40] and C# [101] and to a range of logic programming languages such as Mercury [189], Gödel [102], Typical [139], and many others [162].

As an example, consider the generic append function for lists, with the type signature [T] x [T] → [T]. Here, the brackets stand for a list and T is an unrestricted type parameter, representing the type of all elements. Once the function is called with concrete parameters the type T is instantiated to the least upper bound in the type hierarchy of the two arguments. For example assume the type hierarchy *nat* <: *int*, where *nat* are the natural numbers, *int* are all integer values, and <: represents the subtype relationship. When append is called with the concrete types [*nat*] x [*int*], T is bound to *int*, and the return type of the expression is bound to the most general type [*int*].

We cannot apply this type checking procedure directly to logic programming. The "dataflow" in the evaluation of logic predicates is not unique as in method calls. For example the list concatenation predicate *append* can be used to append the list `[c,d]` to `[a,b]` and bind the resulting list `[a,b,c,d]` to the variable `Var`:

```
append([a,b],[c,d],Var)
```

Alternatively it can be used to bind the postfix `[c,d]` of the list `[a,b,c,d]` to `PostfixVar`:

```
append([a,b],PostfixVar,[a,b,c,d]).
```

The different options to handle different dataflows in the type system and the chosen solution are discussed in Section 2.4.

1.5 contributions

In this thesis we developed a *Context Management Infrastructure* (CMI) that simplifies the development of context-sensitive (adaptation) languages based on Java. Next to the infrastructure itself, the following main contributions have been made:

Object-oriented logic Context Query Language (OCQL). We developed a statically typed logic query language that supports subtyping and parametric polymorphism. In the type system we developed

- a mapping from RDF Schema to Java interfaces with statically typed properties that covers multiple *range* definitions, and
- an extension of Lu's mode analysis algorithm for normal logic programs [130] and its formal semantics to support the higher-order predicate findall.

Context- and Service-oriented Aspect-Language (CSLogicAJ). We have shown the applicability of the CMI with the integration of OCQL in an aspect-language. The resulting language CSLogicAJ fulfills all requirements of a context-sensitive adaptation language elicited above.

- For this purpose CSLogicAJ introduces a generalized advice construct allowing synchronous and asynchronous program adaptation.
- In a case study we demonstrate the practicability of CSLogicAJ and show that the modularization is improved over traditional modularization approaches.

1.6 THESIS OUTLINE

Chapter 2 introduces necessary fundamental concepts used in this thesis. It shortly summarizes Java Generics, the Semantic Web, the OSGi component framework, logic programming, and mode analysis; it discusses approaches for typed logic programming and, lastly introduces AOP to the unfamiliar reader.

Chapter 3 develops an extension of an existing mode analysis for logic programs. The higher-order predicate findall is added to a mode analysis with high precision [130].

Chapter 4 presents OCQL, a statically typed logic language with meta predicates based on a polymorphic type system with subtyping. It starts with a mapping from RDF schema classes to Java types which ensures type-safe property definitions. The syntax and semantics of the language are then depicted and a type checking method based on Java Generics, the mode analysis from Chapter 3, and the RDF Schema - Java mapping is presented.

Chapter 5 gives an overview of a CMI, a framework for the definition of context-aware adaptation languages. Based on the CMI we develop the aspect language CSLogicAJ and show that is fulfills all the requirements discussed above.

Chapter 6 illustrates the realization of the presented concepts, while Chapter 7 evaluates them. A case study comparing a purely object-oriented solution with a solution based on context-aware aspects is then conducted. Afterwards, the CMI is evaluated by extending the COP-approach JCop with context-analysis capabilities. The chapter closes with thoughts on the platform's performance.

Chapter 8 summarizes related work from AOP and context-awareness frameworks. Chapter 9 provides a conclusion and discussion of future work.

Part I

STATE OF THE ART

2 BASIC CONCEPTS

2.1 OSGI

The *OSGi Service Platform* is a specification introduced by the *OSGi Alliance* in March 1999 [12]. It specifies a Service-Oriented Architecture (SOA) framework for Java based upon a component concept, the so called *bundles*. Bundles are a unit of Java classes, libraries and resources. They define static and optional package level dependencies to other bundles. They can be independently deployed, as long as static dependencies are fulfilled, and have a framework-controlled life cycle. Figure 2.1 shows the states a bundle goes through. OSGi was build with runtime installations and updates in mind. A bundle can be installed at the startup of the runtime environment or at any time after that and can be loaded from both a file as well as a URL. Once a new version of the bundle becomes available it can be stopped, updated and restarted.

Bundles communicate via services. Bundles provide their services via a central service registry and at which consumers may request specific services by using service attribute filters. Section 2.1.2 presents the LDAP filters used in service queries in detail. The dynamic nature of service compositions makes tracking of registered services necessary. A consumer needs to be notified once a referenced service is unregistered or alternative services become available.

OSGi introduced the ServiceTracker ([11, 701]) class to support tracking with a common concept. A bundle does not use a direct reference to a service but a reference controlled by the ServiceTracker class. The ServiceTracker is thereby notified about service events and can update the reference as necessary. Bundles react to service events programmatically and define for instance how a service is rebound. Figure 2.2 illustrates the indirection. Every time the bundle refers to a service it calls the method getService()[1] that may rebind the service reference depending on previous calls to addingService(..) and removedService(..) that are called on OSGi service events.

2.1.1 *OSGi Shortcomings*

OSGi is a SOA framework that has to fit many different requirements. It has to stay technically low-level while providing all further needed support as an API to accomplish the requisites. The following subsections enumerate the limitations of OSGi concerning an ambient intelligence setting and its particular requirements.

Stale References

The abrupt departure of services often implies the appearance of so-called *stale references* ([12, 5.4]), which are a very common problem to handle. On the one hand OSGi provides tools and techniques such as listeners [12, 6.1.23] and the described service trackers to help dealing with this. On the other hand this indirection must be used consistently throughout the code. Especially when building ontop of frameworks that expect a fixed reference to a service, e.g., a content provider for a GUI framework, the ServiceTracker is not applicable.

Furthermore, the service selection strategies have to be implemented in the bundle itself, even if the selection is based upon architectural considerations made on the deployment level.

1 Alternatively getServiceReference() can be called. A ServiceReference contains further meta data about a service and holds a reference to service.

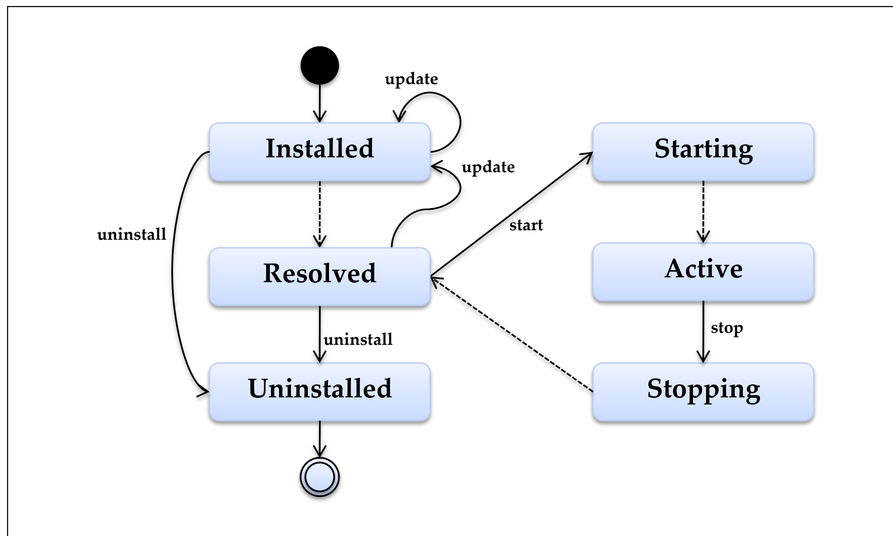


Figure 2.1: OSGi bundle lifecycle

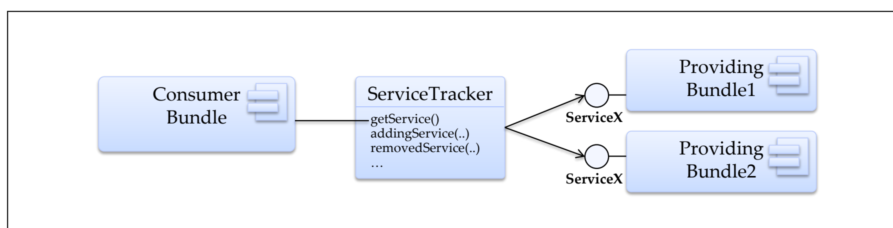


Figure 2.2: OSGi service tracker

Service management and composition

OSGi offers different means for service binding. By default clients handle service bindings themselves. They have to manage the requested services after the binding succeeded. This implies they also have to deal with the departure and the (re-)arrival of services and, of course, with stale references. With respect to OSGi as a platform for highly dynamic services this problem even increases.

OSGi R3 does not provide composition support for services. The standard procedure for service interaction is to find them in the service registry, track and use them. This approach does not consider relationships between services and therefore there is no way to express the composition of services. In particular the dynamic (re)composition of services according to external changes is not supported.

Eclipse OSGi R4 introduced *Declarative Services* to solve this problem to some extend. A bundle can statically define *service references*, which are configured and bound. This concept was introduced to simplify the automatic composition of services and the configuration of fallbacks. Declarative services allow for dynamic reconfiguration to a small extend. Service references marked as *dynamic* can be reconfigured at runtime once a bound service becomes unavailable. Equivalent services are automatically retrieved, and if necessary started and bound. But OSGi does not offer the means to influence this behavior further by providing it with a custom implemented logic.

In this work we concentrate on the standard tracking approach and show how it can be extended to support dynamic reconfiguration, which can be configured by adaptation code.

2.1.2 *LDAP Attributes and Filters*

The LDAP (Lightweight Directory Access Protocol) [18] was originally developed as a directory service to manage information about users, hardware and services in company networks. The LDAP standard defines a filter language to search for (filter) resources on the network based on attributes attached to the resource. Each attribute stores a single type of information. The predefined attribute *objectclass* represents all class (interface) names under which a service was registered to the system. OSGi services are optionally registered with a set of attributes attached to the service. For example, the location sensor below is registered with a `quality` property with value `10`:

```
Hashtable props = new Hashtable();
props.put("quality", "10");
LocationSensor sensor = ...;
bc.registerService(LocationSensor.class.getName(),
                   sensor, props);
```

LDAP filters define conditions on these attributes, based on the following operators:

Equality: (attribute=value) , e.g.,
(objectclass=org.cs3.AddressBook), the service was registered under the class name org.cs3.Addressbook

Negation: (!(attribute=value)) , e.g.,
(!objectClass=group)

Presence: (attribute=*) , e.g.,
(quality=*), the argument quality is present

Absence: (!(attribute=*)) , e.g.,
(!quality=*), the argument quality not is present

Greater than: (attribute>value) , e.g.,
(precision> 10), the precision property is greater than 10

Less than: (attribute<value) , e.g.,
((precision<90), the precision property is less than 90

Logic And: (&(attribute_1=value_1),...) , e.g.,
(&(objectclass=org.cs3.Location),(precision< 100))

Logic Or: (|(attribute_1=value_1),...) , e.g.,
(|(objectclass=org.cs3.Location),(objectclass=org.cs3.GPS))

Wildcards: (mail=*@uni-bonn.de)

The full syntax of LDAP filters in OSGi is defined in OSGi Specification [12, 3.2.7]. The attributes can be updated at runtime by altering the properties on the service reference:

```
ServiceReference ref =
    bc.registerService(LocationSensor.class.getName(),
            sensor, props);
...
ref.setProperties(props);
```

In case the properties change, a service event is fired and all registered listeners are notified. This will be relevant for the definition and implementation of context source services in Section 5.1.3.

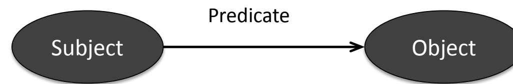


Figure 2.3: The RDF triple

2.2 SEMANTIC WEB

The term *Semantic Web* was coined by Tim Berners-Lee [36]:

> It "is a Web of action-able information - information derived from data through a semantic theory for interpreting the symbols. The semantic theory provides an account of "meaning" in which the logical connection of terms establishes interoperability between systems." [182]

The semantic web is based on the *Resource Description Framework* (RDF) [121]. RDF is a specification for the representation and exchange of resources and their metadata. An RDF resource is any resource that can be referred to by a URI, such as Web pages, servers, or other resource descriptions. Resources are associated with properties, consisting of a property type, describing the property's relationship to the resource, and a value. RDF uses triples to describe resources, which are structured into

- a *Subject*
- an *Object*, and
- a *Predicate*, called property.

A property is, like a resource, described by a globally unique *Unified Resource Identifier* (URI). Each subject is associated either with a URI or a *blank node*. Blank nodes are distinct in their defined context (e.g., a file), but have no globally defined URI reference. They are mostly used for constructing auxiliary structures such as ordered lists, which are otherwise not expressible in RDF. Objects can be URIs, blank nodes or *Literals*. Literals are either typed or untyped (plain) and may have an additional language tag. Literals store arbitrary strings. Typed literals have an datatype URI attached, which typically refers to XML Schema[2] built-in datatypes [203]. For example, to refer to the Integer type the URI http://www.w3.org/2001/XMLSchema#int is used. The triples can be viewed as a directed graph, see Figure 2.3.

Triples are usually serialized as XML or in the N3 notation [35], a more compact and human readable representation of triples. A subset of N3 was specialized for RDF and named *Turtle* (Terse RDF Triple Language). The following two code examples illustrate the serialization of the triple "Distance to Stephan is 10 Meters" in XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:cms="http://www.cs.bonn.edu/cslaj/cms#">
 <rdf:Description
   rdf:about="http://www.cs.bonn.edu/cslaj/cms#Stephan">
  <cms:distanceTo rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
   10
  </cms:distanceTo>
 </rdf:Description>
</rdf:RDF>
```

and in the Turtle triple syntax:

```
@prefix cms: <http://www.cs.bonn.edu/cslaj/cms#>.
cms:Stephan cms:distanceTo "10"^^<http://www.w3.org/2001/XMLSchema#int>.
```

Throughout this thesis we will use the Turtle syntax for brevity.

2 Fallside and Walmsley [74]

2.2.1 *RDF Schema*

Except for typed literals triples are untyped in RDF. *RDF Schema* (RDFS) [45] was introduced to enable the modeling of simple ontologies. RDF schema allows to describe classes and their properties as well as the subtype relationships between classes and properties. The members of a class are called *instances*. The following constructs and their explanation are an excerpt of the RDF schema specification [45, Section 2]. The constructs will be used in Section 4.1 that describes a mapping from RDF schema to a Java type taxonomy.

Classes

Classes describe sets of resources. The following concepts are instances of rdfs:Class - even rdfs:Class itself:

rdfs:Resource is the class of everything. All other classes are subclasses of this class.

rdfs:Class is the class of resources which are RDF classes.

rdfs:Literal is the class of plain and typed literals. Typed literals are instances of rdfs:Datatype. rdfs:Literal is an instance of rdfs:Class and a subclass of rdfs:Resource.

rdfs:Datatype is the class of data types, a subclass of rdfs:Literal.

rdf:XMLLiteral is the class of XML literal values, an instance of rdfs:Datatype

rdf:Property is the class of RDF properties.

Properties

RDF Properties describe the relationship between subject and object resources. The following constructs are all instances of rdf:Property:

rdfs:range states that the value of a property is an instance of a certain class. In case of several range definitions the value is an instance of all declared classes.

rdfs:domain states that subject of a property is an instance of a certain class or a set of classes.

rdf:type states a resource is an instance of a class.

rdfs:subClassOf describes the subtype relationship between two classes.

rdfs:subPropertyOf describes the subtype relationship between two properties.

rdfs:label states a human-readable version of a resources's name.

Additionally RDFS defines the container classes rdf:Bag, rdf:Seq, rdf:Alt, for sets, lists and enumerations and the collections class rdf:List. Since they will all be represented as lists in our approach we will not further investigate their semantics.

2.2.2 *RDFS Entailment*

The *RDFS entailment* is the complete deductive closure of a given set of triples. RDF and RDF Schema define a number of axiomatic triples [98, Section 3.1] and entailment rules [98, Section 7] like the following:

```
uuu rdfs:subClassOf xxx.        ⟹        vvv rdf:type xxx.
vvv rdf:type uuu.
```

It states that if there is a subclass relationship between uuu and xxx and a resource vvv is of type uuu, then vvv is also of type xxx. Applying the rules on the triple set is a monotonic function, once a fixpoint is reached the entailment is complete. Approaches computing the entailment by a fixpoint computation are called *forward chaining*. They typically apply the RETE algorithm [78], which optimizes the recalculation of the entailment on modifications of the triple base. Semantic web reasoners, such as Pellet and Jena, follow this approach [157, 48].

Alternatively the rules are applied by *backward chaining* when executing a query as realized in the SWI-Prologs semantic web library [208]. Wielemaker optimizes the query evaluation by reordering literals. Other approaches [176, 211] avoid redundant evaluation of rules by applying *tabling*, a caching technique that stores previously evaluated (sub)queries.

2.2.3 *Ontology Web Language*

RDFS was the first attempt to describe resources in the Semantic Web. As an ontology language it shows a number of drawbacks. Among other things it does not offer constructs to define cardinalities, describe complement relationships, declare classes to be equivalent or instances to be the same[3].

To tackle these limitations the W3C developed the Ontology Web language family (OWL) [124]. Initially it contained three languages with different levels of expressiveness and reasoning complexity. They are built around the description logic [22] inspired language OWL DL and contains a decidable subset of first-order predicate logic. The ontologies describe sets of *individuals*[4] and their relationships. The rather restricted language OWL lite, is intended to simplify tool support and reduce worst-case reasoning complexity.

OWL Full has different semantics than the other two and was designed to preserve compatibility with RDF Schema, so the separation between individuals and classes is not enforced. In general OWL Full is not decidable. Since OWL Full uses an RDF-style model theory a serialization and combination with RDFS graphs is possible. Under certain restrictions[5], OWL DL can also be serialized into RDFS.

2.2.4 *Open World vs. Closed World Reasoning*

The semantic web is based on the *open world assumption* (OWA). Reasoning with OWA assumes that we have only partial knowledge about the world, meaning the absence of a fact does not imply this fact is false. Here, negative knowledge like "Peter is not a computer scientist." must explicitly asserted. The absence of the fact "Peter is a computer scientist." does not imply it. In other words the query "Is Peter not a computer scientist?" will result in the answer "unknown".

In turn logic programming approaches like Prolog are based on the *closed world assumption*(CWA) [170]. Prolog assumes a fact is false if it is not asserted or inferable. This is called *Negation as Failure* (NaF). Under NaF the query "Is Peter not a computer scientist?" results in the answer "no".

---

3 Declared class equivalence can be an important means for merging independent ontologies.
4 instead of instances in RDFS
5 Amongst other things the graph must not contain cyclic constructions, for a detailed discussion see [22, Chapter 14].

2.3 PROLOG

An in-depth introduction to Prolog is given in [52]. Here we will only summarize the core principles and terms relevant to this work. The name Prolog stands for PROgramming in LOGic. Prolog is a declarative language based on horn clauses [109], a subset of first-order logic (FOL). A horn clause is a set of atomic literals with at most one positive literal. Let's assume p_i and u are positive literals. Then

$\neg p_1 \vee ... \vee \neg p_n \vee u$

is a definite horn clause. Rewritten as an equivalent implication

$p_1 \wedge ... \wedge p_n \rightarrow u$

it represents a *definite* Prolog clause, meaning there is no negative literal in the body of a clause. The Prolog syntax of the corresponding clause is

u:-p_1,...,p_n.

A clause without a head is called a *goal*, a clause without a body is called a *fact*. The disjunction of all clauses with the same name and argument arity assemble a *predicate*. A predicate is fully determined by its name and argument arity, written as *<Predicate-Name>/<Arity>* in short. A set of predicates forms a logic *program*.

A logic program is called a *normal logic program* [128] in case the clause bodies are conjunctions of optionally negated literals. *Queries* are goals posted by a user to the Prolog system for evaluation. In the following example the first two lines are facts forming the predicate person with arity 1. Line 1-8 represent a Prolog program, the last line a query, querying for the father of Frank.

```
person('Peter').
person('Frank').
male('Peter').
parent('Peter','Frank').
father(Father,Child) :-
  parent(Father,Child),
  male(Father).
?- father(Father,'Frank').
```

Prolog's only data type is a *term*. Terms are either variables, atoms, numbers and compound terms.

Variables are denoted as identifiers starting with an upper-case character. A special case are anonymous variables denoted with an underscore "_". Every occurrence of an anonymous variable stands for a new, unique variable.

Atoms are atomic character arrays, which are either arbitrary characters enclosed by single quotes, e.g., 'atom 1' or identifiers starting with a lower character.

Numbers can be float or integer values.

Compound Term is an atom, called *functor* in this case, which has a number of arguments that are enclosed by parenthesis and separated by commas: functorname(term$_1$,...,term$_n$). Terms are *strict*, meaning they are of finite depth[6].

In the following we do not distinguish between atoms and numbers, because the distinction is not relevant to this work. Since both are atomic we consider both to be constant and therefore in the set *Atom*.

6 Most of the current Prolog implementations support *rational trees*[54], also known as cyclic trees. They result from recursion in the unification process, for example in X = t(X), which leads to the infinite tree t(t(t(...))). We will not consider rational trees any further here and only assume that strict terms.

2.3.1 *Operational Semantics*

The operational semantics of Prolog is essential for this work, since we base the semantics and static type analysis of the context-query language on Prolog semantics.

A *substitution* $\theta = \{V_1 \leftarrow t_1, ..., V_n \leftarrow t_n\}$ with $V_i \in LVar$, $t_i \in Term$ represents the assignment of terms to variables. The application of a substitution to a term t is written as $t\theta$. All occurrences of variables in t, which occur on the left-hand-side (lhs) of θ, are replaced by the terms on the right-hand side. In case all t_i are variables and all variables are unique the substitution is called a renaming substitution.

At the core of Prolog's resolution process is the concept of unification [128]. Unification is a symmetric and transitive operation that makes two terms equal. Two terms t_1 and t_2 are unifiable if there is a substitution θ with $t_1\theta = t_2$. In this case θ is called a *unifier*. θ is the *most general unifier* (mgu) of two terms t_1 and t_2 if for all other substitutions θ' there is a substitution η with $t_1\theta' = t_1\theta\eta$. Unification can be implemented efficiently [135], with linear time complexity.

From a logical perspective, Prolog evaluates a goal by finding the resolution refutation of the negated goal. In case the negated goal can be refuted, the goal is a logical consequence of the program. The evaluation of Prolog is based on (SLDNF) [16] with the *left-to-right selection rule* [128]. *Linear* here means that literals in a clause body are processed in order of their definition. The left-to-right selection rule results in depth-first search on the resolution tree, accordingly the variable bindings occur from left to right.

The operational semantics of Prolog are based on a depth-first left-to-right search with *backtracking*. A literal is unified with the head of the called clauses starting with the first clause and backtracks over all defined clauses to find bindings for given variables.

Every time multiple clause heads can be matched by a literal a *choice point* is added by the Prolog system. If a (sub-)goal fails in the further execution all variable bindings that have been made since the last choice point, are undone and the evaluation continues with the next alternative at this choice point.

Consider the following example:

C_1 :`a :- c,b.`
C_2 :`b.`
C_3 :`c :- d.`
C_4 :`c :- e.`
C_5 :`d.`
C_6 :`e.`

Let's assume the goal ?- a is evaluated. Below we list the resolution steps and the evaluated clauses:

Evaluated Term	**Clause**
a	
c,b	C_1
d,b	C_3
true,b	C_5
true	C_2

The literal a is unified with clause C_1 and a is replaced by c, b. Next, c is unified with C_3 and d is unified with the fact C_5. Finally b is unified with C_2, resulting in "true".

SLDNF adds *negation as failure* to the SLD resolution. A sub-goal $\neg Goal$ succeeds if the sub-proof of *Goal* fails and vise versa.

2.3.2 *Module Concept*

Most Prolog systems modularize predicates with a module system, where most of the current approaches[7] adopted the module system of *Quintus Prolog [6].* Via modules, predicates can be defined in a namespace. Modules can export predicates and import predicates from other modules. For details we refer to the Quintus Prolog documentation *[6].*

2.3.3 *Meta Predicates*

Prolog offers a set of meta predicate to, e.g., map predicates to lists, convert lists to terms, terms to goals, or backtrack over all solutions of a goal. Further Prolog allows the definitions of new meta predicates. The *univ* predicate "=.." converts between a term and a list and allows the constructions of terms at runtime. The predicate call(*Term*) calls *Term* as a goal. The following meta-predicate constructs the term `Goal` from the name `PredName` and a list of arguments `ArgList` and calls the constructed goal:

```
call_pred(PredName,ArgList):-
  Goal =.. [PredName|ArgList],
  call(Goal).
```

In this thesis we only consider a small subset of the meta predicate facilities, the three - so called - *all-solution* predicates `findall/3`, `bagof/3` and `setof/3`. They will later be used in the specification and implementation of the logic query language OCQL.

The predicate findall(*Template*, *Goal*, *Bag*) backtracks over all solutions of *Goal* and aggregates the results as a list of the instantiations of the term *Template* and unifies the list with *Bag*. In case *Goal* fails, *Bag* is an empty list. For example, the following goal binds the variable `Bag` to the cross product of the two lists `[a,b]` and `[c,d]`:

```
?- findall((A,B),(member(A,[a,b]),member(B,[c,d])),Bag).

  Bag = [ (a, c), (a, d), (b, c), (b, d)].
```

The meta-predicates bagof/3 and setof/3 are similar to findall/3, but allow backtracking over variables not part of the template term. For example:

```
?- bagof(A,(member(A,[a,b]),member(B,[c,d])),Bag).
  B = c, Bag = [a, b] ;
  B = d, Bag = [a, b].
```

The variable `B` is not part of the template and the evaluations backtracks over each binding of `B`, collecting all corresponding bindings of `A` in `Bag` for each binding of `B`. The setof/3 predicate binds the last argument to a *set* instead of a *bag* of bindings. The two predicates make use of so called *existential* variables. The semantics is similar to template variables. They are also not bound in the evaluation of bagof, even if they are not contained in the template term. The syntax for existential variables is *Var^Goal.* Applied to the example above, the result is:

```
?- bagof(A,B^(member(A,[a,b]),member(B,[c,d])),Bag).

  Bag = [a, a, b, b].
```

Existential variables are syntactic sugar. They can be realized by forwarding predicates, which omit existential variables [153]. For the given example the code can be rewritten into:

---

7 SICStus, Ciao, YAP and SWI, to some extend Eclipse

```
forward(A) :-
   member(A,[a,b]),
   member(B,[c,d]).
?- bagof(A,forward(A),Bag).
```

2.3.4 *Prolog Syntax*

We will use only a subset of Prolog in this work, except for the findall predicate we will not use any meta predicates. We will now define the syntax that will be used throughout this thesis to describe Prolog code. Let *LVars* be the set of logic variables, *Term* the set of terms, *Literal* the set of literal, the *PosLiteral* the set of positive literals, *Atom* the set of atoms, *Formula* the set of formulas, *Clause* the set of clauses, *Program* the set of Prolog programs and *Goal* the set of goals. To refer to single elements of these lists we use the characters: $V \in LVars, t \in Term, L \in Literal, B \in PosLiteral, A \in Atom, \phi \in Formula, Cl \in Clause, \mathcal{P} \in Program, \mathcal{G} \in Goal$. The single elements have the following syntax:

$$
\begin{aligned}
t &::= VT | f(t_1, ..., t_n) | A \\
VT &::= V | tuple(t_1, ..., t_n) \\
B &::= A(t_1, ..., t_n) \\
L &::= B | \backslash + B \\
A &::= string | number \\
\phi &::= \epsilon | L | \phi_1, \phi_2 | t_1 = t_2 | \phi_1 - > \phi_2 ; \phi_2 | findall(t, L, V) | V \; is \; f(t_1, ..., t_n) \\
Cl &::= A : - \phi \\
\mathcal{P} &::= Cl_1, ..., Cl_n \\
\mathcal{G} &::= ? - \phi
\end{aligned}
$$

2.4 TYPED LOGIC PROGRAMMING

Logic programming languages have not been developed with type systems in mind. Flexibility and conciseness have been the main design goals. But, this has lead to problems in debugging and code comprehensibility. A range of approaches have been proposed to integrate type systems with logic programming [162]. They follow two different perspectives, *prescriptive typing* and *descriptive typing*. Prescriptive type systems require type annotations for every predicate. Based on the typed predicate declarations the types of untyped variables are inferred and finally type checkers verify the well-typedness of a program, partially guided by *mode* annotations[8].

Descriptive typing approaches approximate types from a program without type annotations. They describe semantic properties of a program mostly based on regular tree sets [100, 120]. The approximated types are not necessarily intuitive or reflect the intention of the user. Nevertheless they can help to understand the programs semantics, detect errors in the program or guide evaluation optimization. Here we will focus on a static type system that can be aligned with the Java type system and therefore concentrate on prescriptive type systems. We will now illustrate the general considerations that led to the chosen approach.

The first type systems for Prolog containing parametric polymorphism were based on type terms [144, 125, 102, 189], which declare type constructors and predicate type signatures. For example, the following declarations define two type constructors:

```
:- type int := {...,-1,-2,0,1,2,...}.
```

8 Modes will be discussed later in this section

```
:- type list(T) := {[], [T|list(T)]}.
```

The first represents the set of integer as a list of negative, 0 and positive numbers. The parametric type list with type variable T is recursively defined as an empty list or a list, where the first element hast the same type as the type parameter of the rest of the list. Based on the parametric types generic predicates can be declared:

```
:- pred append(list(T), list(T), list(T)).
:- pred member(T, list(T)).
```

The `append` predicate appends the two lists in the first and second argument to form the list in the third argument. All lists share the same component type. The `member` predicate tests membership of the first argument in the list given in the second argument. The type of the element and the component type of the list are the same.

The goals and predicates are type checked with regards to the type declarations. Typically a constraint solving algorithm based on Mycroft O'Keefe's type checking algorithm [144] is applied, which is an adaptation of Milner's constraint-based type checking algorithm for the functional language ML [141].

For example consider combining two lists of integers and comparing their contents with 1:

```
append([1,2,3],[1,2],List),member(A,List),A==1.
```

The goal is well-typed and type variable `T` is unified with the type `int`. Later approaches considered also subtyping [188, 103, 162], but Meyer [139] has shown deficiencies in their combination of polymorphism and subtyping. He gave a number of examples for typing problems, one is the generic list concatenation predicate append/3 with type signature

```
append(list(T),list(T),list(T))
```

where T is a type variable. The type system contains two primitive types, `nat`, `negint`, and `int`, where `nat` and `negint` are subtypes of `int`. Further a predicate `p/1` is defined with a type signature `p(list(nat))`. Now we consider the following goal:

```
?- p(L), append([1], [-1], L).
```

The goal is obviously ill-typed, since L is a list over natural numbers, but the list to append contains negative values. The afore-mentioned approaches consider the goal well-typed, since their type checking algorithms do not consider the subtype relationships between the type parameters of the arguments. The constraint solving results in type `int` for `T`:

```
append(list(int),list(int),list(int))
```

Meyers solution in the type system of *Typical*, was the introduction of subtype relationships between type arguments. Here is the revised version for the append predicate with subtype constraints:

```
:- pred appendnew(list(@T1), list(@T2), list(@T3))   |> T1 =< T3, T2 =< T3.
```

When we reconsider the append example from above, the type checking fails, because the constraints for the type parameters are not solvable. T3 must be of type `nat` and a supertype of `negnat`. Meyer considered static typing as a *type approximation*, which detects ill-typed goals, but does not give any runtime guarantees. In addition, the approach does not consider any operational semantics, but is applicable to arbitrary evaluation schemes, for instance to SLD resolution or bottom-up approaches as in deductive databases [69].

The drawback is, that type checking does not detect obviously ill-typed goals that contain non-ground arguments. Consider the following type definition of the predicate nat_list and the unification operator:

```
pred nat_list := list(nat).
pred '=' := @T x @T.
nat_list([1,_]).
```

The nat_list predicate declares a list of natural numbers as its only argument type, unification is a polymorphic predicate over arbitrary arguments. The definition of nat_list contains an important detail, the argument is unified with a non-ground list with one argument of type nat. The type checker accepts this unification, since the list is only partially instantiated, but the instantiated part has the correct type.

Now we evaluate a goal on a predicate that binds LNat to the list [1,-1,1], and is obviously not of type list(nat):

```
?- nat_list(LNat),LNat=[1,_],append(LNat, [1], L), append(L, [-1], X),L=[_,-1 ]
```

Meyers approach considers the program as well typed. The main issue here is that the type checking does not consider the *dataflow* between variables occurrences. This was not his intention, since type approximation, was the main aim of his work. The same considerations apply for TCLP [73]. TCLP is type checker for Prolog and *Constraint Logic Programming* [114]. TCLP extends the typing rules by Mycroft O'Keefe with subtyping rules and solves them with by constrain logic programming. Besides subtyping and parametric polymorphism TCLP considers predicate overloading[9]. Subtype hierarchies are restricted to type lattices, meaning a least upper bound and greatest lower bound must be defined for every pair of type. The constraint system resulting from rule application is solved via *Constraint Handling Rules* [81].

Already Hill and Todor [103, Example 1.4.14] have shown that subtyping can go wrong in typed logic programming and needs further restrictions. To solve this problem the dataflow between variables can be taken into account, by making use of a *mode system.*

2.4.1 *Instantiation Modes*

One of the unique features of logic programming is that predicates can be used in multiple directions, meaning arguments of predicates might be bound or unbound when calling a predicate. For example consider the predicate

```
atom_concat(Atom1,Atom2,Atom3)
```

It can be used on the one hand to concatenate to atoms `Atom1` and `Atom2` and bind a unbound third argument `Atom3` to result:

```
?- atom_concat(a,b,Atom3).
 Atom3 = ab
```

Alternatively `Atom3` could be bound and `Atom1` and Atom2 are bound to all combinations of prefixes and postfixes of `Atom3`.

```
?- atom_concat(Atom1,Atom2,ab).
 Atom1 = ", Atom2 = ab ;
 Atom1 = a, Atom2 = b ;
 Atom1 = ab, Atom2 = ".
```

The different instantiations are called *modes*. Legal modes of a predicate can be annotated for documentation or optimization purposes [189, 206]. The sign + means the argument must be bound, - must not be bound and ? that it can be used in both ways. For atom_concat/3 this means that (+,+,+),(+,-,+),(-,+,+),(+,+,-) and (-,-,+) are legal modes for the predicate. The logic language presented in this work is based on Prolog semantics

9 This can avoid subtyping relationships, e.g., between float and integer, that are typically not found in ISO Prolog implementations

and needs to deal with different directions of variable unification in the presence of a static type system. Let's assume that int is a subtype of float in the following example. The type of a variable is postfixed with a colon to the variable name:

```
p(A:int, B:float):-
  A=B.
```

If the predicate is called with mode (+,-) B is bound to an int value. This is type safe since int is a subtype of float. When p is called with the mode (-,+) an illegal unification of A with a float value happens. To overcome this problem different solutions have been proposed. One solution is the extension of the unification process to order-sorted unification [29, 30], the other is to restrict predicates to certain modes [189] allowing to restrict only type-correct unification.

2.4.2 *Type System based on Restricted Modes*

Several logic programming languages are based on strong mode and type systems with parametric polymorphism, e.g., Mercury [189] and Gödel [102], but do not consider subtype relationships[10].

Dietrich and Hagl [66] included subtyping in their type system. They facilitate the modes to describe a dataflow relation between variable occurrences to rule out illegal dataflows between a subtype and a supertype variable bindings, as shown in Section 2.4.

Since the subtype relationship between polymorphic types depends on the concrete values of type variables Dietrich and Hagl introduced the notion of *conditional subtype relations* (CSR). CSRs lead to a constraint system that has to be solved to type check a program, which is not decidable in general [66, 212].

Another approach based on mode annotations was proposed by Smaus et al. [185]. They have shown *subject reduction*[11] for typed logic programs with subtyping and parametric polymorphism, but further restricted unification to *moded unification* [185]. In short, moded unification demands a fixed data-flow for each argument in a literal unification. This is a severe restriction, as also Smaus stated in his PhD thesis [186]. Even simple clause testing the equality of two arguments, cannot be type checked:

```
eq(A,A).
```

The two arguments are naturally input arguments, but the dataflow depends on the concrete usage.

Although the approaches by Dietrich and Smaus show the mentioned deficiencies, both have presented possible solutions for statically typed logic programming in Prolog, which support subtyping and parametric polymorphism facilitating a mode analysis. This will be the starting point for our own approach for a typed logic language, based on the Java type system.

2.4.3 *Order-sorted Unification*

An alternative approach to type checking, is order-sorted unification [162], which extends term unification in typed logic languages. Predicates and variables are typed based on an *ordered type language* $\mathcal{T}$ based on an *ordered type alphabet* of partially ordered term constructors L, K. The type constructors are either monotypes, representing a non-parameterized set of values, e.g., integer or float values, or parameterized constructors $K(\alpha_1, ...)$, where α_i are type parameters. A partial order $<:$ on the set of type constructors

10 Mercury's theoretically supports *undiscriminated* unions, which have not been implemented, because type checking would take exponential time.

11 Subject reduction is a consistence property, that ensures that in a well-typed program, all derivations starting a well-typed goal are again well-typed.

represents the subtype relationship between the constructors $L <: K$. A type declaration is constructed based on (nested) type constructors.

Order-sorted unification on typed variables unifies two unified terms, as well as the declared types of contained variables. Assume that the variable V is a typed integer and W is typed float, and that these are ordered in the following way: integer <: float. Now an order-sorted unification V=W with the *most-general type unifier (mgtu)* is calculated, which represents the lower bound in the type hierarchy of the two types, in this case integer. The new type for both variables in the rest of the resolution process is integer. On every unification of a typed variable with a non-variable term a type-check is applied with respect to the variable's domain. Statically we can type check a unification by calculating the *mgtu* for each effected type pair. Still we need to test at runtime the unification of constants with a typed variables.

Let's assume we defined the following typed predicate p with typed arguments integer and float. We use Mercury's [189] syntax for typing predicates for this example:

```
:- pred p(integer, float).
p(V, W):-
  V=W.
```

The goal will fail, since Z's type is integer, which is the lower bound of integer and float, resulting from the unification in the body of p. Then at runtime the unification of Z and 1.0 will fail. Either as an error or silently, depending of the chosen semantics.

```
?- p(_,Z),Z=1.0.
```

Order-sorted unification can be integrated with Prolog in different ways [162, 1.4.3]. Either by transforming the language to an untyped language with embedded type checking code, or by extending the SLDNF resolution to directly support order-sorted unification. In both cases this goes along with an evaluation overhead and in the latter case the Prolog system needs to be modified. Paschke [158] applied the concept of order-sorted unification to *description logic-typed unification* to ensure type safety in the earlier versions of the knowledge reasoning system Prova [123]. He extended the normal unification algorithm with the description logic reasoner Pellet [184], resulting in an EXPTIME worst case unification complexity in case of the description logic OWL lite.

2.5 MODE ANALYSIS

This thesis makes use of a *mode analysis* to analyze possible dataflows in logic programs. It is used to detect potentially illegal dataflows between two typed variables and illegal instantiation patterns of mode-restricted predicates. In the following we summaries the *abstract interpretation*[59] concept, the basis of most mode analysis approaches. Afterwards we introduce the chosen mode analysis approach and its distinguishing properties.

2.5.1 *Abstract Interpretation*

Cousot [59] presented the mathematical framework of *abstract interpretation* in 1977. Abstract interpretation statically approximates dynamic properties of a language and has become a common basis for data-flow analysis algorithms. For instance, it has been applied to type inference, code optimizations, garbage collection, and program transformations such as partial evaluation [60].

The initial step in the application of the abstract interpretation framework is the definition a program's formal semantics, the so called *concrete semantics*, for instance, formalized by a transition system [60].

Based on the concrete semantics the *collecting semantics* [58] is defined that associates every program point with the set of possible states of the program when the execution

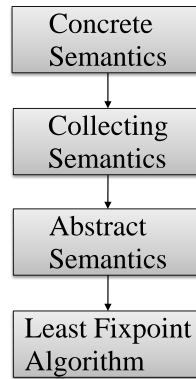


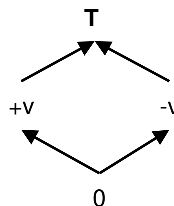
Figure 2.4: Steps to abstract interpretation

(evaluation) reaches that point. The described state at a program point only considers properties the analysis is focused on, ignoring information irrelevant for the later defined abstractions, e.g., the execution traces. Based on the collection semantics an *abstract semantics* is defined that approximates the program state with regards to the properties in focus. Figure 2.4 summarizes the process. In every step the soundness approximation of the more concrete representation must be proven.

A simple example for abstract interpretation is the *rule of sign* analysis taken from [60]. Integer values are categorized into the sets $+v$ (positive or 0), 0, $-v$ (negative values or null) and the symbol $\top$ that denotes arbitrary integer values. These abstract values are called an *abstract domain*. Operations on the concrete domain, such as addition and multiplication, are reinterpreted on the abstract domain in the following way:

$$
\begin{array}{cccc}
+v + +v = +v & +v \times +v = +v & -v \times +v = -v & \\
-v + -v = -v & +v \times -v = -v & -v \times -v = +v & \\
+v + -v = \top & +v + \top = \top & \top \times +v = \top & -v \times \top = \top \\
-v + v = \top & -v + \top = \top & \top \times -v = \top & -v \times \top = \top \\
\top + v = \top & \top + \top = \top & +v \times \top = \top & \\
\top + -v = \top & & & \\
0 + +v = +v & +v + 0 = +v & 0 \times +v = 0 & +v \times 0 = 0 \\
0 + -v = -v & -v + 0 = -v & 0 \times -v = 0 & -v \times 0 = 0 \\
0 + \top = \top & \top + 0 = \top & 0 \times \top = 0 & \top \times 0 = 0 \\
0 + 0 = 0 & & 0 \times 0 = 0 &
\end{array}
$$

Applied to the formula $(3 \times 3) + (4 \times 4)$ the result is $+v$. The different abstract values are not distinct. A partial order can be introduced to compare the precision of the values, here represented as a complete lattice:



It means that $\top$ is the most general abstraction, in which $+v$ and $-v$ are subsets of $\top$, and 0 represents a subset of both $+v$ and $-v$.

The definition of an abstract domain is usually a tradeoff between precision and costs. For example, it might consider the context of a call at a certain point in the program,

resulting in more precise approximations, but at increased space costs compared to a semantics only considering program points in isolation.

2.5.2 *Sound Approximation*

Abstract interpretation is a sound approximation of program's concrete semantics. Ideally, *Galois connections* are used to formalize the approximation process. The abstract domain and the concrete domain are described as *complete lattices* in this case. A complete lattice $L(\sqsubseteq, \bot, \top, \sqcap, \sqcup)$ is a partial ordered set in which any two elements have a least upper bound $\sqcup$ and a greatest lower bound $\sqcap$. For the concrete domain, typically the power set $\wp(C)$ over the set of concrete values C is chosen, which forms a lattice via an order induced by the subset relationship: $(\wp(C), \subseteq, \emptyset, C, \cup, \cap)$. Between the two lattices $D^\flat(\sqsubseteq^\flat, \bot^\flat, \top^\flat, \sqcap^\flat, \sqcup^\flat)$ for the abstract domain and $D^\#(\sqsubseteq^\#, \bot^\#, \top^\#, \sqcap^\#, \sqcup^\#)$ for the concrete domain a Galois connection represents the concretion function γ and abstraction function α with the following properties:

$\gamma : D^\flat \to D^\#$

$\alpha : D^\# \to D^\flat$

$\forall a \in D^\flat : \forall c \in \wp(C), \alpha(c) \sqsubseteq^\flat a \Leftrightarrow c \sqsubseteq^\# \lambda(a)$

These properties guarantee the soundness of the abstraction in both directions. By the Knaster-Tarski fixpoint theorem [198] the fixpoints of a monotone mapping on a complete lattice $L(\sqsubseteq, \bot, \top, \sqcap, \sqcup)$ form a complete lattice. The collecting semantics is described as the least fixpoint (*lfp*) on $lfp\,F^\#$, with $F^\#$ being a monotone function on the concrete domain $D^\#$. The abstract semantics $lfp\,F^\flat$, with the monotone function $F^\flat$ on the abstract domain $D^\flat$, is a sound approximation of the collecting semantics ($lfp\,F^\# \sqsubseteq \gamma(lfp\,F^\flat)$) if

$\forall d^\flat \in D^\flat.(F^\# \circ \gamma(d^\flat) \sqsubseteq \gamma \circ F^\flat(d^\flat))$

For the proof in see [134].

2.5.3 *Chosen Mode Inference Approach*

A large number of mode inference approaches for logic languages based on abstract interpretation have been proposed [62, 46, 55, 192, 53, 131]. Lu [130] gives an extensive overview, we only give a short summary here and argue why we have chosen Lu's approach over others. Besides the mode information, mode inference approaches consider variable relationships in their analysis. Most approaches observe sharing between variables to ensure correctness of the analysis. Sharing means that two variables are bound to two terms, which may contain each other. So in case the mode of one variable changes, the mode of the other variable is influenced. For example, consider the two variables A and B. In case A is bound to t(B) or to B itself the variables are in the sharing relationship, or A=a(B,C) and B=b(A).

We base our analysis on the mode analysis by Lu [131], since his approach has a high approximation accuracy in contrast to former approaches, because his chosen abstract product domain [25] considers - next to variable sharing - also variable aliasing.

The abstract domain is an *abstract substitution* consisting of mode abstractions, variable aliasing and variable sharing. Here, variable sharing ensures the soundness of the approximation, while aliasing increases the accuracy. Modes are described by the set $\Delta = \{f, g, o\}$, in which

f stands for unbound (free) variables

g for ground terms, which do not contain any variables

o for nested terms, which contain unbound variables, e.g., the term t(X)

Example for Aliasing

Already simple examples show the improved precision by a product domain including variable aliasing. Consider the following predicates c/2, p/1 and b/1[12]:

```
p(A):-
  b(A),
  c(A,B),
  B=val.
c(A,B):-
  A=B.
b(a(_A)).
b(1).
```

The goal p(A) results in the substitution ([A=val]), so we expect the inferred mode $\{g\}$. But, when we apply a mode analysis, which only considers variable sharing, the inferred mode for A is $\{g, o\}$. Below we see the goal and the two predicates annotated with the inferred modes for each program point and the corresponding sharing information. The annotation is written as a comment beginning with a list of variable/mode pairs followed by a list of variable sharing tuples. The goal is represented by the common Prolog command prompt "?-".

At the first program point before the call of p(A) the variable A has the mode $\{f\}$ and the sharing list is empty. In the predicate p the predicate b is called, which results in a mode $\{g, f\}$ for A, since the first clause does not bind it and the second binds it to an atom. Now predicate c is called, which results in a sharing relationship between A and B. Now B is unified with val and the mode of B changes to $\{g\}$. Since B shares with A, the mode of A also changes. But the analysis does not know that the variables have been unified and updates the mode on the grounds of the sharing relationship only. Since B might have been just one part of a partially bound term, like A=t(B,_), the sharing information is not sufficient to infer the correct mode $\{g\}$ for A in this case. So the resulting mode for A is $\{f, g, o\}$.

```
?-
      %[A/{f}],[]
    p(A),
      %[A/{f,g,o}],[]
p(A) :-
      %[A/{f},B/{f}],[]
     b(A),
      %[A/{f,g},B/{f}],[]
   c(A,B),
      %[A/{f,g},B/{f,g,o}],[ (A,B), (B,A)]
   B=val,
      %[A/{f,g,o},B/{g}],[]
c(U,U). %[U/{f,g}],[]
b(A).   %[A/f],[]
b(1).   %[],[]
```

Now we consider the product domain with aliasing. A list with the aliasing relationship is postfixed to the sharing information. We ignore the reflexive alias case here for brevity:

```
?-
      %[A/{f}],[],[]
    p(A),
      %[A/{g}],[],[]
```

12 The example was adopted from [131]

```
p(A) :-
      %[A/{f},B/{f}],[],[]
    b(A),
      %[A/{f,g},B/{f}],[],[]
   c(A,B),
      %[A/{f,g},B/{f,g}],[(A,B), (B,A)],[(A,B), (B,A)]
   B=val,
      %[A/{g},B/{g}],[],[(A,B), (B,A)]
c(U,U). %[U/{f,g}],[],[]
b(A).   %[A/f],[],[]
b(1).   %[],[],[]
```

The essential difference here is when `B` is unified with `val`. Now aliasing information is available and the mode $\{g\}$ can be propagated correctly to `A`, resulting in the expected mode for `A`.

2.6 ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Software Development (AOSD) [77] is a paradigm to enhance the separation of concerns. Functionality that would otherwise be scattered throughout a program, because it cannot be modularized in the dominant decomposition [155] of a particular language, is encapsulated into an aspect. This functionality is called a *crosscutting concern*. Typical crosscutting concerns are tracing, transactions and exception handling. Filman and Friedman [75] characterize AOSD as a modularization concept with two desirable properties: *quantification*, the ability to declare actions to be applied consistently to many places of a program, and obliviousness, which means that the modified program entities do not need to be aware of being subject to the execution of aspect code and do not need to provide special hooks or interfaces for enabling aspect application, also known as *weaving*.

A large number of approaches have been proposed to modularize crosscutting concerns [77]. We can divide them into two groups. The first are composition approaches, like the Hyper/J language [155] inspired by subject-oriented programming [96]. Hyper/J is based on the concept of *multi-dimensional separation of concerns* (MDSoC) [155], in which features are independent modules. A program is defined via composition descriptions that form a concrete application from the feature modules.

The second group [127, 33, 118, 17] distinguishes between a base program and aspects. Aspects encapsulate a crosscutting concern and define where in the base program the concern should be woven. For the later group of programming languages the term *Aspect-Oriented Programming* (AOP) has been coined [117].

Most AOP approaches are based on a *join point model*, inspired by AspectJ [118], the first AOP language for Java. The join point model defines which points in the execution of a program can be modified by an aspect. Typical join points are method calls, thrown exceptions and field accesses. Aspects select join points in the so called *base classes* of a program and *weave* the crosscutting concern into the application. Figure 2.5 illustrates the weaving process.

The following sections introduce the core concepts of AspectJ, which has been a role model for our and many other aspect-oriented language designs [105, 91, 32, 47, 191].

2.6.1 *Join Points and Pointcuts*

Join Points are well-defined points in the execution of a program. A *pointcut* selects sets of *join points*. AspectJ distinguishes between primitive pointcuts, an atomic construct, which selects join points of a certain kind, such as method calls, and the pointcut construct, which represents a logic expression over a set of primitive pointcuts.

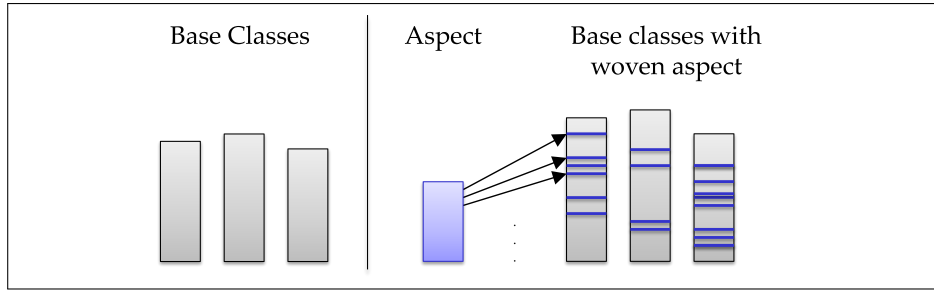


Figure 2.5: Illustration of aspect weaving. An aspect selects a set of join points in the base classes of a program and weaves additional code at these points.

call(*Signature*)	selects all calls to methods whose signature matches the pattern *Signature: e.g.* public * Contact.*(..) selects all method calls on the class Contact with arbitrary number of parameters.
target(*Type* | *Var*)	binds all join points whose runtime receiver object is of type *type*. In case the argument is the name of an advice[14] parameter the type comparison is applied on the parameters type and the advice parameter is bound to the receiver object.
this(*Type* | *Var*)	similar to target, but here the enclosing runtime instance of the join point's call site is checked and bound.
args(*Type* | *Var,...)*	binds all join points, whose arguments are compatible with the list of type names or advice variables. Arguments are either method call arguments or the values assigned to a field. For method calls the wildcard '..' can be used for an arbitrary number of parameters.
further	AspectJ offers a number of additional pointcuts for selecting method executions and field read and write accesses that are not further considered in this thesis, see also [118].

Table 2.1: AspectJ's primitive pointcuts

In AspectJ join points can be field accesses, method calls and thrown exceptions. A pointcut not only selects sets of join points, it may additionally refer to their context. This can be the enclosing class or method, the object the join point is executed on, the enclosing object of the join point itself or the call's arguments. Table 2.1 summarizes the pointcuts with focus on the pointcuts used in this thesis. The *target*, *this* and *args* pointcuts are also called *state-based* pointcuts, because they refer to the runtime state of the program. In contrast to the other primitive pointcuts, runtime checks have to be woven at the selected join points[13], the so-called *dynamic residue*.

Primitive pointcuts use either explicit class or construct names or use patterns with wildcards. The wildcard '*' stands for arbitrary legal Java identifier characters. In type name patterns the pattern '**' can be used to match also the dots in fully qualified type names, so that several sub-packages can be matched at once. To include subtypes in the pattern, too, a '+' sign can be attached to a type name. The following example illustrates the type pattern in a call pointcut:

```
call(void com.**.Test+.set*(Object, ..))
```

13 Attempts have been made to partially remove the runtime tests with static analysis [20].

This poincut selects all calls to methods of the types named `Test` and it's subtypes. `Test` must be defined in a sub-packages of `com`. The method names must start with `set` and the first parameter must be of type `Object`. The primitive pointcuts can be composed by logic operators. These terms can be reused by defining a *named pointcut* containing the term:

```
pointcut allTestMethods() : call(void Test.set*(Object, ..));
```

The pointcut `allTestMethods` selects all methods of the class Test, which must have been resolved in the context of the enclosing aspect.

2.6.2 *Advice*

Advice constructs encapsulate the functionality, which should be executed at a join point, and define where the functionality should be executed. Four different advice kinds are defined in AspectJ:

before(): The *before* advice is executed before the join points.

after(): The *after* is always executed after the join point, regardless if it returns regularly or with a thrown exception. This advice kind is further specialized into two variants, which consider the two mentioned cases:

- **after() throwing():** The advice is only executed if the join point throws an exception. The thrown exception is bound in the throwing clause to an advice parameter.
- **after() returning():** The advice is executed on successful execution of the join point. The return value is bound in the returning clause to an advice parameter.

around(): The *around* advice is the most generic advice and subsumes all others. It is executed around a join point. Optionally the advice body can execute the original join point.

The following advice adds tracing messages before all calls of the method `m` of the class `Test`:

```
before() : call(void Test.m()) {
    System.err.println("called method Test.m()");
}
```

The *around* advice offers a special call statement: *proceed*. A call of *proceed* either calls the original join point or the next advice woven at this join point. The following example shows an advice, which caches the values returned by a method `m` of the class `Test`:

```
private Hashtable cached = new Hashtable();
String around() : call(String Test.m(..)) {
    if (cached.get("Test.m") != null)
        return (String)cached.get("Test.m");
    String ret = proceed();
    cached.put("Test.m", ret);
    return ret;
}
```

Advices can access the runtime context of a join point via the field `thisJoinPoint`. Figure 2.6 shows the field's interface. The method `getThis()` returns the enclosing instance, `getTarget()` the target objects and `getArgs()` the arguments of the join point. Further the advice can access the method signature elements.

State-based pointcuts offer means to bind the join point's context. The target object, the current instance, the return value, and the arguments of a join point can be bound to advice parameters, which can further be processed in the advice body. In around advices

```
interface JoinPoint {
  Object getThis();
  Object getTarget();
  Object[] getArgs();
  Signature getSignature();
  DitriosFacade getDitriosFacade();
}
```

Figure 2.6: JoinPoint interface

parameters can be passed to the proceed call, allowing the arguments and also the target object of a join point to be altered.

We can make use of advice parameters in our caching example. Let's assume the method has a parameter of type Object. We include the hash code of this argument in our cache, to cache results separately for each unique argument:

```
Hashtable cached = new Hashtable();
String around(Object arg) : call(String Test.m(..))
                            && args(arg) {
  String key = "Test.m." + arg.hashCode();
  if (cached.get(key) != null)
      return (String)cached.get(key);
  String ret = proceed(arg);
  cached.put("Test.m." + arg.hashCode(), ret);
  return ret;
}
```

2.6.3 *Inter-type Declarations*

AspectJ offers inter-type declarations, a means to alter the class hierarchy or add new fields and methods to existing classes. Since this thesis is only concerned with runtime adaptations and these features is not applicable at runtime (in Java), we will not consider them any further here.

2.6.4 *Aspects*

Aspects encapsulate pointcut, advice and inter-type declarations in a class-like construct. They are mainly a syntactic extension of a Java class, in which the `class` keyword is replaced by the keyword `aspect`. Next to aspect constructs, they can contain the definition of fields and methods which can be referenced in the body of the advices. Aspects are by default singleton instances. Below we define an exemplary tracing aspect. It puts out a tracing message before every method call that prints the number of calls since the start of the application:

```
public aspect TraceAspect {
  int counter = 0;
  before(Object target) : call(* *.*(..)) &&
                          target(Object target){
    counter++;
    System.err.println("current class:" +
              o.getClass().getName() + " +
              "number of method calls: "+counter);
  }
}
```

Part II

APPROACH

3 FINDALL-EXTENDED MODE ANALYSIS

This chapter introduces an essential ingredient for a later presented statically-typed logic query language. As we discussed in Section 2.4.2, typed logic programming with parametric polymorphism and subtyping needs to consider the dataflow in the resolution of a goal, to avoid type errors. We will facilitate the mode analysis by Lu [131] to resolve the dataflow. His approach has a high approximation accuracy, resulting from an abstract product domain [25] that considers variable aliasing in contrast to former approaches. In Section 2.5.3 we argued why aliasing support is important for the precision of the analysis.

Our query language makes use of the higher-order predicate findall/3 and other all-solution predicates, but higher-order predicates are not considered by Lu's abstract semantic. In this chapter we will first introduce the formal underpinning of Lu's approach and then we extend the semantics and the algorithm to support findall/3 and similar meta-predicates.

In his Phd thesis [131] Lu presented a generic abstract interpretation approach for *normal logic programs* [128][1], which can be tailored to static analysis problems. Next to type inference he presented a mode analysis as an application of the framework. Section 3.1 summarizes the syntax and relevant definitions from [131], but we only consider the abstract mode domain and ignore the generic parts. The definitions and syntax used in Section 3.1 are precisely taken over from Lu's thesis Lu [131, Chapter 4 and 5], which simplifies references to his work. For readability reasons we will not cite each single definition, since all originate from the same publication.

Section 3.2 presents our extension to Lu's approach.

3.1 MODE ANALYSIS APPROACH BY LU

Lu's analysis is based on the abstract interpretation framework, which we summarized in Section 2.5.1. The abstract domain considered in Lu's semantics is an *abstract substitution* $Sub^{\flat}$ consisting of mode abstractions, variable aliasing and variable sharing. Two *aliased* variables have be unified directly or indirectly in previous resolution step. Two s*haring* variables v_1 and v_2 are not unified directly, but one of the variables is unified with a term containing the other variable, for example, $v_1 = t(v_2)$.

The modes of a variable are described by the set $\Delta = \{f, g, o\}$, where

f stands for unbound (free) variables

g for ground terms, which do not contain any variables

o for nested terms, which contain unbound variables, e.g., the term t(X)

The abstract substitution correlates every variable with a subset $m \in \Delta$. The abstract domain is build up from the power set $\wp(\Delta)$ forming the complete lattice $< \wp(\Delta), \subseteq >$. An abstract substitution θ^b is $< \mathcal{I}, \mathcal{S}, \mathcal{A} >$, with the instantiation state function $\mathcal{I} : \mathcal{V} \to \wp(\Delta)$, the sharing relation $\mathcal{S}$ and aliasing relation $\mathcal{A}$. The sharing between two variables X and Y under substitution θ means that $X\theta$ and $Y\theta$ contain a common variable. Aliasing means that two variables are unified by a substitution.

A *program point* p refers to a location before or after a literal in a program. We refer to the enclosing clause or goal of p with $p[1]$. Only the variables used in the context of the enclosing clause of a program point are relevant for the mode of p. Section 3.1.1 will

1 *In a normal logic program all clause bodies are conjunctions of optionally negated literals.*

explain the program graph representation in more detail, but for now we introduce its syntax without further explanation. The set of variables used in a clause i is referred to as $\mathcal{V}_i$. So $\mathcal{V}_{p[1]}$ represents the set of variables used in the enclosing clause of p. $Sub^\flat[\![\mathcal{V}]\!]$ denotes the domain of abstract substitutions for describing values of variables in $\mathcal{V}$. $Sub^\flat$ denotes the set of all abstract substitutions and $< Sub^\flat[\![\mathcal{V}]\!], \sqsubseteq [\![\mathcal{V}]\!] >$ forms a complete lattice with $\sqsubseteq [\![\mathcal{V}]\!]$ defined as

$< \mathcal{I}_1, \mathcal{S}_1, \mathcal{A}_1 > \sqsubseteq [\![\mathcal{V}]\!] < \mathcal{I}_2, \mathcal{S}_2, \mathcal{A}_2 > \stackrel{def}{=} \forall X \in \mathcal{V}.\mathcal{I}_1(X) \subseteq \mathcal{I}_2(X) \wedge (\mathcal{S}_1 \subseteq \mathcal{S}_2) \wedge (\mathcal{A}_1 \subseteq \mathcal{A}_2)$

The concretion function on the domain is defined on the abstract substitution as

$\gamma[\![\mathcal{V}]\!](\theta^\flat) \stackrel{def}{=} \gamma_i[\![\mathcal{V}]\!](\theta^\flat \downarrow Inst) \cap \gamma_s[\![\mathcal{V}]\!](\theta^\flat \downarrow Share) \cap \gamma_a[\![\mathcal{V}]\!](\theta^\flat \downarrow Alias)$

where

$$\begin{aligned}
\gamma_i[\![\mathcal{V}]\!](\mathcal{I}) &\stackrel{def}{=} \{\theta \in Sub \mid \forall X \in \mathcal{V}.(X\theta \in \check{\gamma}(\mathcal{I}(X)))\} \\
\gamma_s[\![\mathcal{V}]\!](\mathcal{S}) &\stackrel{def}{=} \left\{ \theta \in Sub \;\middle|\; \begin{array}{c} \forall X, Y \in \mathcal{V}.(X \not\equiv Y) \notin \mathcal{S} \rightarrow vars(X\theta) \cap vars(Y\theta) = \varnothing) \\ \wedge \\ \forall X \in V.((X, X) \notin \mathcal{S} \rightarrow \chi(X\theta) \neq 2) \end{array} \right\} \\
\gamma_a[\![\mathcal{V}]\!](\mathcal{A}) &\stackrel{def}{=} \{\theta \in Sub \mid \forall X, Y \in \mathcal{V}.((X, Y) \in \mathcal{A} \rightarrow X\theta \equiv Y\theta)\} \\
\check{\gamma}(M) &\stackrel{def}{=} \{t \in Term \mid mo(t) \in M\} \\
\chi(t) &\stackrel{def}{=} \begin{cases} 0 & if\ vars(t) = \varnothing \\ 2 & if\ \exists X \in vars(t).repeat(X, t) \\ 1 & otherwise \end{cases}
\end{aligned}$$

The $\downarrow$ operator extracts the corresponding component from $Sub^\flat$. $\gamma_i[\![\mathcal{V}]\!](\mathcal{I})$ represents the set of substitutions binding all variables $X \in \mathcal{V}$ to terms having the same mode as given by the instantiation function $\mathcal{I}(X)$. The set $\gamma_s[\![\mathcal{V}]\!](\mathcal{S})$ contains all substitutions that contain the sharing property given by the relation $\mathcal{S}$ and finally $\gamma_a[\![\mathcal{V}]\!](\mathcal{A})$ covers all substitution resulting in the aliasing defined in $\mathcal{A}$.

For example, for the abstract substitution[2]

$< \{A/\{f\}, B/\{f\}, C/\{f\}, D/\{o\}\}, \{\{A, B\}, \{C, D\}\}, \{\{A, B\}\} >$

some of the corresponding concrete substitutions are:

$< \{A = B, D = t(a, C)\} >$

$< \{A = B, D = t(C)\} >$

$\vdots$

The abstract unification operation $unify^\flat(B_1, \theta^\flat, B_2, \sigma^\flat)$ for the mode analysis is build around the ∇ operator $\nabla : \wp(\Delta) \times \wp(\Delta) \rightarrow \wp(\Delta)$, shown in Figure 3.1. It unifies two terms based on their abstract modes resulting in the most general common abstract value of the terms. For example, $\{f, g\} \nabla \{o\} = \{g, o\}$, since the unification of a free term with a partially instantiated term results in a partially ground term, and the unification of a ground term with a partially ground term is a ground term. Both combinations are summed up in the resulting set.

The abstract unification algorithm solves a set of equations E in solved form[3] resulting from the most-general unifier (mgu) of B_1 and B_2. The instantiation state resulting from one equation in E is defined by $est^m(X = t, \mathcal{I})$, where $\mathcal{I}$ is an abstraction instantiation state on $V, X \in V$ a variable, and t a term with $vars(t) \subseteq V$. Let $V_e = \{X\} \cup vars(t)$.

2 we omit the reflexive aliasing and the symmetric sharing and aliasing tuples for brevity

3 the left-hand side (lhs) of all equations in E is variable and occurs only once on the lhs in the equation set and the lhs does not occur in the rhs of any equation in E.

∇	$\varnothing$	$\{f\}$	$\{g\}$	$\{o\}$	$\{f,g\}$	$\{f,o\}$	$\{g,o\}$	Δ
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\{f\}$	$\varnothing$	$\{f\}$	$\{g\}$	$\{o\}$	$\{f,g\}$	$\{f,o\}$	$\{g,o\}$	Δ
$\{g\}$	$\varnothing$	$\{g\}$	$\{g\}$	$\{g\}$	$\{g\}$	$\{g\}$	$\{g\}$	$\{g\}$
$\{o\}$	$\varnothing$	$\{o\}$	$\{g\}$	$\{g,o\}$	$\{g,o\}$	$\{g,o\}$	$\{g,o\}$	$\{g,o\}$
$\{f,g\}$	$\varnothing$	$\{f,g\}$	$\{g\}$	$\{g,o\}$	$\{f,g\}$	Δ	$\{g,o\}$	Δ
$\{f,o\}$	$\varnothing$	$\{f,o\}$	$\{g\}$	$\{g,o\}$	Δ	Δ	$\{g,o\}$	Δ
$\{g,o\}$	$\varnothing$	$\{g,o\}$	$\{g\}$	$\{g,o\}$	$\{g,o\}$	$\{g,o\}$	$\{g,o\}$	$\{g,o\}$
Δ	$\varnothing$	Δ	$\{g\}$	$\{g,o\}$	Δ	Δ	$\{g,o\}$	Δ

Figure 3.1: ∇ operator, over-estimates the mode of two unified terms. Adapted from [130, Figure 5.1].

$$est^m(X = t, \mathcal{I}) \stackrel{def}{=} \lambda U \in V_e. \begin{cases} \mathcal{I}(X) \nabla mo^\flat(t, \mathcal{I}) & if\, t \equiv U \vee U \equiv X \\ \varnothing & if\, \mathcal{I}(X) = \varnothing \vee mo^\flat(t, \mathcal{I}) = \varnothing \\ \{g\} & if\, \mathcal{I}(X) = \{g\} \vee mo^\flat(t, \mathcal{I}) = \{g\} \\ \Delta \nabla \mathcal{I}(U) & otherwise \end{cases}$$

where $mo^\flat$ evaluates the mode of the term t:

$$mo^\flat(t, \mathcal{I}) \stackrel{def}{=} \begin{cases} \mathcal{I}(t) & if\, t \in \mathcal{V} \\ \varnothing & if\, \exists X \in vars(t).\mathcal{I}(X) = \varnothing \\ \{g\} & if\, \forall X \in vars(t).\mathcal{I}(X) = \{g\} \\ \{g, o\} & if\, \forall X \in vars(t).(g \in \mathcal{I}(X)) \\ \{o\} & otherwise \end{cases}$$

$est^m(X = t, \mathcal{I})$ evaluates the mode of X after unification with t, based on the previous known modes for X and t. For example, for $\mathcal{I}(X) = \{f\}, \mathcal{I}(Y) = \{f\}$ and $t = a(Y)$ the evaluation of $est^m(X = t, \mathcal{I})$ results in $\{g\}$.

The full $unify^\flat$ algorithm with equation solving is quoted in appendix A.2.1. The mode analysis is build up from a collecting semantics abstracting from a transition system that approximates a variant of SLDNF resolution [130, 4.1.2], abbreviated as VSLDNF. The evaluation of VSLDNF is semantically equivalent to SLDNF, but instead of preserving the variable names of a goal it focuses on preserving variable names at a program point. We mainly illustrate the SLD and transition system here. For a detailed description and the SLDNF VSLDNF equivalence proof see [130, 4.1]. A clause C_i is represented as head H_i with a list of literals $L_{(i,j)}$, where $m : N \to N$ denotes the number literals in C_i.

$C_i \equiv H_i \leftarrow L_{(i,1)}, ..., L_{(i,m[i])}$

In SLDNF resolution a sub-refutation step of a positive literal $B_{i,j}$, which is unified with a clause head H_k with a current substitution θ, is processed by the calculating the most general unifier:

$mgu(H_k\rho, B_{i,j}\rho'\theta) = \eta$

Here, ρ' and is a renaming substitution from previous evaluation steps and ρ a renaming substitution ensuring that the right-hand side does not share variable names with H_k.

Afterwards, the literals $L_{k,l}$ are processed under the substitution $\theta \circ \eta$. So in case a goal g(A) is unified with clause g(B) the variable name A is preserved through the refutation steps in the clauses of g/1. VSLDNF differs in that the renaming of variables takes place on the right-hand side, preserving the name of the clause C_i instead of the goal's variables:

$mgu(H_k, B_{i,j}\rho'\rho\theta) = \eta$

This makes a new unification step necessary once a sub-refutation of a clause is completed, since otherwise the variables do not match the variables in further literals in the context anymore. Assume a goal $\leftarrow L_{(i,1)}, ..., L_{(i,m[k])}\theta$, and the refutation of $L_{(i,1)}$ on the clause C_j was completed with θ', then in exit step a renaming substitution ρ'' with $vars(H_j, \theta') \cap vars(C_i\theta) = \varnothing$ is created and the substitution of this step becomes $\theta'' = mgu(B_{(i,1)}\theta, H_j\theta'\rho'')$, resulting in the new goal

$\leftarrow L_{(i,2)}, ..., L_{(i,m[k])}\theta''$.

3.1.1 *Program Graph*

Lu abstracts VSLDNF with a transition system based on the program graph$E_{P,G}$ defined as follows[4]:

$$\mathcal{E}_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \bigcup_{0 \leqslant j \leqslant 5} \mathcal{E}^j_{\mathcal{P},\mathcal{G}}(X) \tag{3.1}$$

$$\mathcal{E}^0_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ entry(k) \leftarrow\!\!\bullet (0,0) \;\; | k \in \aleph_G \right\} \tag{3.2}$$

$$\mathcal{E}^1_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ entry(i) \leftarrow\!\!\bullet q \;\middle|\; \begin{array}{ll} & q[2] \leqslant m[q[1]] \\ \wedge & i \in \aleph_C \\ \wedge & \exists \rho. \left(\begin{array}{ll} & \rho \text{ is a renaming substitution} \\ \wedge & vars(B_q\rho) \cap vars(H_i) = \varnothing \\ \wedge & mgu(B_q\rho, H_i) \neq fail \end{array} \right) \end{array} \right\} \tag{3.3}$$

$$\mathcal{E}^2_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ p \leftarrow\!\!\bullet exit(i) \;\middle|\; entry(i) \leftarrow\!\!\bullet p^- \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \wedge L_{p^-} \equiv B_{p^-} \right\} \tag{3.4}$$

$$\mathcal{E}^3_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ p \leftarrow\!\!\bullet p^- \;\middle|\; L_{p^-} \equiv \neg B_{p^-} \right\} \tag{3.5}$$

where $\aleph_G$ represents a set of natural numbers enumerating goals, $\aleph_C$ the set of clause numbers, with $\aleph_G \cap \aleph_C = \varnothing$. The set of initial goals $\mathcal{G} \stackrel{def}{=} \{< G_k, \Theta_k > | k \in \aleph_G\}$, where G_k stands for goal and Θ_k is a set of substitutions θ_k. Each $G_k\theta_k$ is an initial goal. *Program points* are represented as tuples (i, j), where the first argument is the clause number and the second the position in a clause body or goal, starting with 1. The set of program points is named PP. The $entry : N \to PP$ refers the first program point in goal $(i, 1)$ or a clause, $exit : N \to PP$ the last one $(i, m[i] + 1)$. The functions p^- and p^+ are used to refer to point before resp. after a program point in a clause or goal. The functions are only defined if there is such a point. An edge in the graph is represented as $p \leftarrow\!\!\bullet q$, meaning the edge starts at q and ends at p. A literal indexed by a program point L_p is the literal directly following the program point p. Literal kinds can be tested with the equivalence relation $\equiv$. They can be either positive $L_p \equiv B_p$ or negative $L_p \equiv \neg B_p$.

${\mathcal{E}_{\mathcal{P},\mathcal{G}}}^0$ is a set of edges from the dummy program point (0,0) to the entry points of initial goals. ${\mathcal{E}_{\mathcal{P},\mathcal{G}}}^1$ contains the edges to clause-entry points, ${\mathcal{E}_{\mathcal{P},\mathcal{G}}}^2$ from exit points to program points after the calls from positive literals. ${\mathcal{E}_{\mathcal{P},\mathcal{G}}}^3$ considers negation as failure rules. In this case the called clause is entered, but no bindings are returned to the calling literal.

Figure 3.2 gives an example of a program graph. The goal `g(A,B)` is referenced from from program point (0,0). Program point (1,1) is located before the first literal in the goal `g(A,B)`. $L_{(1,1)}$ refers to this literal. The program graph contains call edges to the program point (2,1), after the head of the fact `g(a2,E)`, referred to as H_2. Since the clause has no body the call immediately exits to (1,2). The rest of the graph follows the same scheme.

4 the definition is a copy from [130]

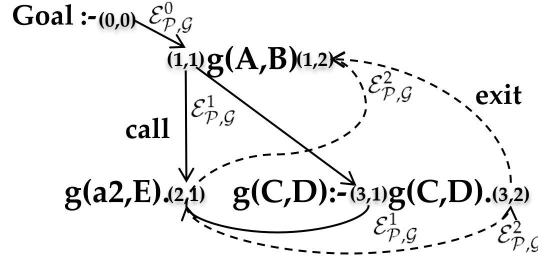


Figure 3.2: Call graph example

3.1.2 *Concrete Semantics*

We will now summarize the transition system approximating VSLDNF, which is then approximated by the collecting semantics as the least fixed-point of the function $F^{\#}_{\mathcal{P},\mathcal{G}}$, defined below. A state in the transition system is a stack of stack items. A stack item has the form $\| p \leftarrow\!\bullet q, \theta \|$, where $p \leftarrow\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}}$ and $\theta \in Sub$. The set of possible stack items is
$\mathbf{S}^{\#} = \{\| p \leftarrow\!\bullet q, \theta \| \,| p \leftarrow\!\bullet q \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \wedge \theta \in Sub\}$

The empty stack is denoted as \$, the set **S** of stack items is inductively defined:

- $\$ \in S$;and
- $\| p \leftarrow\!\bullet q, \theta \| \cdot S \in \mathrm{S}$ if $\| p \leftarrow\!\bullet q, \theta \| \in \mathbf{S}^{\#} \wedge S \in \mathbf{S}$.

Instead of $x_1 \cdot ... \cdot x_n \cdot S$ we write

$$\frac{x_1}{\quad}\ \vdots\ \frac{x_n}{\quad}\ \frac{\quad}{S}$$

The set of $S_0 \subseteq \mathbf{S}$ represents the set of initial states, determined by the set of initial goals in SLDNF:

$\mathbf{S}_0 \stackrel{def}{=} \{\| p \leftarrow\!\bullet q, \theta \| \cdot \$ | \ \| p \leftarrow\!\bullet q, \theta \| \in \mathbf{S}^{\#} \wedge p \leftarrow\!\bullet q \cdot \in \mathcal{E}^{o}_{\mathcal{P},\mathcal{G}} \wedge \theta \in \Theta_{p[1]}\}$

where $\Theta_{p[1]}$ are the initial substitutions at the initial goals. The set of final states is

$\mathbf{S}_\infty \stackrel{def}{=} \{\| exit(k) \leftarrow\!\bullet q, \theta \| \cdot \$ | k \in \aleph_G \wedge \theta \in Sub \wedge exit(k) \leftarrow\!\bullet q \in \mathcal{E}^{o}_{\mathcal{P},\mathcal{G}}\}$

The set of descendent stats of the initial states in S_0 is the least fixed-point of the function $F_{\mathcal{P},\mathcal{G}}$ as follows:

$$F_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \bigcup_{0\leqslant j\leqslant 3} F^j_{\mathcal{P},\mathcal{G}}(X) \tag{3.6}$$

$$F^0_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \{\| \, p \leftarrow\!\bullet q, \theta \, \| \cdot \$ | p \leftarrow\!\bullet q \cdot \in \mathcal{E}^0_{\mathcal{P},\mathcal{G}} \wedge \theta \in \Theta_{p[1]}\} \tag{3.7}$$

$$F^1_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \frac{\| \, p \leftarrow\!\bullet q, \theta \, \|}{\dfrac{\| \, q \leftarrow\!\bullet u, \sigma \, \|}{S}} \;\middle|\; \begin{array}{c} p \leftarrow\!\bullet q \cdot \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \wedge L_q \equiv B_q \\ \wedge \quad \| \, q \leftarrow\!\bullet u, \sigma \, \| \cdot S \in X \\ \wedge \quad \theta = unify(B_q, \sigma, H_{p[1]}, \epsilon) \neq fail \end{array} \right\} \tag{3.8}$$

$$\cup \left\{ \| \, p \leftarrow\!\bullet q, \theta \, \| \cdot S \;\middle|\; \begin{array}{c} p \leftarrow\!\bullet q \cdot \in \mathcal{E}^1_{\mathcal{P},\mathcal{G}} \wedge L_q \equiv \neg B_q \\ \wedge \quad \| \, p \leftarrow\!\bullet q, \sigma \, \| \cdot S \in X \\ \wedge \quad \theta = unify(B_q, \sigma, H_{p[1]}, \epsilon) \neq fail \end{array} \right\} \tag{3.9}$$

$$F^2_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \| \, p \leftarrow\!\bullet q, \theta \, \| \cdot S \;\middle|\; \begin{array}{c} p \leftarrow\!\bullet q \cdot \in \mathcal{E}^2_{\mathcal{P},\mathcal{G}} \\ \wedge \quad \dfrac{\| \, q \leftarrow\!\bullet u, \sigma \, \|}{\dfrac{\| \, p^- \leftarrow\!\bullet v, \eta \, \|}{S}} \in X \\ \wedge \quad \theta = unify(H_{q[1]}, \sigma, B_{p^-}, \eta) \neq fail \end{array} \right\} \tag{3.10}$$

$$F^3_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \{\| \, p \leftarrow\!\bullet q, \theta \, \| \cdot S | p \leftarrow\!\bullet q \in \mathcal{E}^3_{\mathcal{P},\mathcal{G}} \wedge \| \, q \leftarrow\!\bullet u, \theta \, \| \cdot S \in X\} \tag{3.11}$$

The $unify$ operation reflects VSDLNF semantics, ensuring to preserve variable names at the call-side:

$$unify(A, \theta, B, \omega) \stackrel{def}{=} \left\{ \begin{array}{c} let\ \rho\ be\ a\ renaming\ substitution\ that \\ vars(A\theta\rho) \cap vars(B\omega) = \emptyset\ in \\ \begin{cases} \omega \circ mgu(A\theta\rho, B\omega) & if\ mgu(A\theta\rho, B\omega) \neq fail \\ fail & otherwise \end{cases} \end{array} \right. \tag{3.12}$$

The domain $\mathbf{D}$ of $F_{\mathcal{P},\mathcal{G}}$ is $\wp(\mathbf{S})$, where $< D, \subseteq, \emptyset, \mathbf{S}, \cap, \cup >$ forms a complete lattice, $F_{\mathcal{P},\mathcal{G}}$ is monotone on $\mathbf{D}$. $F^0_{\mathcal{P},\mathcal{G}}(X)$ represents all program starts, with all edges from (0,0) to the graph's goals combined with all initial substitutions Θ_k for a goal G_k. $F^1_{\mathcal{P},\mathcal{G}}(X)$ represents the stacks with calls on top, either on positive (equation 3.8) or negative literals (equation 3.10).

$F^2_{\mathcal{P},\mathcal{G}}(X)$ represents the exit edges from positive literal calls. $F^3_{\mathcal{P},\mathcal{G}}(X)$ the exits from negative literal calls. In this case the substitution in the call of $\neg B_q$ are ignored, since they do not contribute to the substitution in the clause $q[1]$. So the substitution θ of the edge $q \leftarrow\!\bullet u$, with $q = p^-$, is taken over for $p \leftarrow\!\bullet q$.

3.1.3 *Collecting Semantics*

The fixed-point collecting semantics abstracts from the traces by collecting sets of substitutions at a program point and classifies the stack items according to the edges $p \leftarrow\!\bullet q$. Let A, B be atoms, and Θ, Ω be sets of substitutions. The unification in the collecting semantics is defined as:

$$unify^{\#}(A, \Theta, B, \Omega) \stackrel{def}{=} \{unify(A, \theta, B, \omega) \neq fail | \theta \in \Theta \wedge \omega \in \Omega\}$$

The fix-point collection semantics is defined as follows:

$$
\begin{aligned}
&[F^{\#}_{P,G}(X^{\#})]_{p\leftarrow\bullet q} \stackrel{def}{=} \\
&\Theta_{p[1]} && if\ p\leftarrow\bullet q \in \mathcal{E}^0_{P,G}\\
&\bigcup\{unify^{\#}(B_q, X^{\#}_{q\leftarrow\bullet u}, H_{p[1]}, \{\epsilon\}) | q\leftarrow\bullet u \in \mathcal{E}_{P,G}\} && if\ p\leftarrow\bullet q \in \mathcal{E}^1_{P,G}\\
&\bigcup\left\{unify^{\#}(H_{q[1]}, X^{\#}_{q\leftarrow\bullet u}, B_{p^-}, X^{\#}_{p^-\leftarrow\bullet v}) \middle| \begin{array}{l} p^-\leftarrow\bullet v\cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge\ q\leftarrow\bullet u\cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\end{array}\right\} && if\ p\leftarrow\bullet q \in \mathcal{E}^2_{P,G}\\
&\bigcup\{X^{\#}_{q\leftarrow\bullet u} | q\leftarrow\bullet u \in \mathcal{E}_{P,G}\} && if\ p\leftarrow\bullet q \in \mathcal{E}^3_{P,G}
\end{aligned}
$$

The proof that $lfpF_{P,G}(X)$ is approximated by $lfpF^{\#}_{P,G}(X^{\#})$ is given by Lu [130] based on transfinite induction.

3.1.4 *Abstract Semantics*

The abstract semantics associates abstract substitutions with each edge $p\leftarrow\bullet q$, approximating the sets of substitutions described by the $lfpF^{\#}_{P,G}(X^{\#})$. Let $\theta^{\flat}_{p[1]} \in Sub^{\flat}[\![\mathcal{V}_k]\!]$ be the least abstract substitution such that $\Theta_k \subseteq \gamma[\![\mathcal{V}]\!](\theta^{\flat})$ for each $k \in \aleph_G$. The least upper-bound $\sqcup[\![\mathcal{V}]\!]$ approximates the most specific substitution from the elements of the complete lattice $< Sub^{\flat}[\![\mathcal{V}]\!], \sqsubseteq [\![\mathcal{V}]\!] >$:

$$[F^{\flat}_{P,G}(X^{\flat})]_{p\leftarrow\bullet q} \stackrel{def}{=} \tag{3.13}$$

$$\{\theta^{\flat}_{p[1]} | p\leftarrow\bullet q \in \mathcal{E}^0_{P,G}\} \tag{3.14}$$

$$\sqcup [\![\mathcal{V}_{p[1]}]\!] \left\{unify^{\flat}(B_q, X^{\flat}_{q\leftarrow\bullet u}, H_{p[1]}, \epsilon^{\flat}[\![\mathcal{V}_{p[1]}]\!]) \middle| \begin{array}{l} p\leftarrow\bullet q \in \mathcal{E}^1_{P,G} \\ \wedge\ q\leftarrow\bullet u \in \mathcal{E}_{P,G}\end{array}\right\} \tag{3.15}$$

$$\sqcup [\![\mathcal{V}_{p[1]}]\!] \left\{unify^{\flat}(H_{q[1]}, X^{\flat}_{q\leftarrow\bullet u}, B_{p^-}, X^{\flat}_{p^-\leftarrow\bullet v}) \middle| \begin{array}{l} p\leftarrow\bullet q \in \mathcal{E}^2_{P,G} \\ \wedge\ p^-\leftarrow\bullet v \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge\ q\leftarrow\bullet u \in \mathcal{E}_{\mathcal{P},\mathcal{G}}\end{array}\right\} \tag{3.16}$$

$$\sqcup [\![\mathcal{V}_{p[1]}]\!] \left\{X^{\flat}_{q\leftarrow\bullet u} \middle| \begin{array}{l} p\leftarrow\bullet q \in \mathcal{E}^3_{P,G} \\ \wedge\ q\leftarrow\bullet u \in \mathcal{E}_{P,G}\end{array}\right\} \tag{3.17}$$

Where the abstract identity substitution

$$\epsilon^{\flat}[\![\mathcal{V}]\!] \stackrel{def}{=} < \{X/\{f\} | X \in \mathcal{V}\}, \emptyset, \{(X,X) | X \in \mathcal{V}\} >$$

abstracts from the identity substitution ϵ such that $\epsilon \in \gamma[\![\mathcal{V}_i]\!](\epsilon^{\flat}[\![\mathcal{V}_i]\!])$ for each $i \in \aleph_C$.

3.2 FINDALL-EXTENDED MODE ANALYSIS

Lu's mode analysis does not provide support for meta predicates. The program graph, concrete semantics, collecting semantics, and abstract semantics only consider calls of positive or negative literals. We extend the program graph and the abstractions with edges from the meta predicate findall literals (see Section 2.3.3) to the called predicate and back.

Instead of the regular findall predicate we use a more general definition $findall^B$ that binds the permutation of all subsets of answer substitutions of the regular findall/3 predicate (see Section 2.3.3):

Definition 3.2.1. Let `permutation`(*List*, *PermList*) be a permutation predicate binding *PermList* to all permutations of *List*. And `subset`(*Sub*, *List*) be predicate binding *Sub* to all subsets including the empty list of the list *List*. Then

```
findall^B(Template,Literal,Bag):-
  findall(Template,Literal,BagTmp),
  subset(Sub,BagTmp),
  permutation(Bag,Sub).
```

The bindings of $findall^B/3$ are obviously a superset of bindings of the regular `findall/3` predicate. But this will not influence the precision of the mode analysis. We will not give a formal proof for this assertion, but give a short explanation. The modes of variables are only considered locally for each program point. The order of list elements or the bindings of list members are not relevant for the abstraction. Since we cannot statically detect if the goal in the second argument of findall will fail or not, the collecting semantics for the Bag argument must include the binding "[]". So the subsets and permutations will not influence the later abstraction and allow for the set representation of the possible bindings for the template.

We further restrict $findall^B$ literals to simplify the later abstractions, and name its arguments and program points:

Definition 3.2.2. Let B_q be a positive literal, $Tl_{p \leftarrow\bullet q}$ a template term and Bag_p be a variable, then $findall^B(Tl_{p \leftarrow\bullet q}, B_q, Bag_p)$ is a literal with the following three program points:

q — is the program point before the literal.

p — represents the point after meta-call B_q and before the unification of the list of bindings $[Tl_{p \leftarrow\bullet q}\theta_1, \ldots, Tl_{p \leftarrow\bullet q}\theta_n] = Bag_q$, where the θ_i represent answer substitutions from the call B_q, whose variables are renamed such that $\forall i, j : i \neq j \Rightarrow vars(Tl_{p \leftarrow\bullet q}\theta_i) \cap vars(Tl_{p \leftarrow\bullet q}\theta_j) = \emptyset$.

p^+ — represents the program point after the $findall^B$ literal.

We restrict Bag_p to a variable to allow for a later simplification in the abstraction semantics. We only consider the case of one positive literal in the second argument of a positive findall literal. Other formulas can be normalized into this case by extracting the formulas in the second argument or negative calls into a separate clause.

The program graph $\mathcal{E}_{\mathcal{P},\mathcal{G}}(X)$ (see Section 3.1.1) is extended to the graph $\mathcal{E}^f{}_{\mathcal{P},\mathcal{G}}(X)$ by three additional edge sets. $\mathcal{E}^4_{\mathcal{P},\mathcal{G}}(X)$ represents the meta calls starting at the second argument of the $findall^B$ predicate. It is aligned with positive literal calls in $\mathcal{E}^1_{\mathcal{P},\mathcal{G}}(X)$ (3.3). $\mathcal{E}^5_{\mathcal{P},\mathcal{G}}(X)$ represents the exit edges from the meta call, and $\mathcal{E}^6_{\mathcal{P},\mathcal{G}}(X)$ is a set of edges representing the unification of the answer substitution list $[Tl_{p \leftarrow\bullet q}\theta_1, \ldots, Tl_{p \leftarrow\bullet q}\theta_n]$ with the Bag_p variable.

$$\mathcal{E}^f{}_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \bigcup_{0 \leqslant j \leqslant 5} \mathcal{E}^j_{\mathcal{P},\mathcal{G}}(X) \tag{3.18}$$

$$\vdots \tag{3.19}$$

$$\mathcal{E}^4_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ entry(i) \leftarrow\!\bullet q \;\middle|\; \begin{array}{l} \quad q[2] \leqslant m[q[1]] \\ \wedge \quad i \in \aleph_C \\ \wedge \quad L_q = findall^B(Tl_{p\leftarrow\bullet q}, B_q, Bag_p) \\ \wedge \quad \exists \rho. \left(\begin{array}{l} \quad \rho \text{ is a renaming substitution} \\ \wedge \quad vars(B_q\rho) \cap vars(H_i) = \varnothing \\ \wedge \quad mgu(B_q\rho, H_i) \neq fail \end{array} \right) \end{array} \right\} \tag{3.20}$$

$$\mathcal{E}^5_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ p \leftarrow\!\bullet exit(i) \;\middle|\; \begin{array}{l} \quad entry(i) \leftarrow\!\bullet q \in \mathcal{E}^4_{\mathcal{P},\mathcal{G}} \\ \wedge \quad L_q \equiv findall^B(Tl_{p\leftarrow\bullet q}, B_q, Bag_p) \end{array} \right\} \tag{3.21}$$

$$\mathcal{E}^6_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ p^+ \leftarrow\!\bullet p \;\middle|\; \begin{array}{l} \quad p \leftarrow\!\bullet v \in \mathcal{E}^5_{\mathcal{P},\mathcal{G}} \\ \wedge \quad L_q \equiv findall^B(Tl_{p\leftarrow\bullet q}, B_q, Bag_p) \end{array} \right\} \tag{3.22}$$

Figure 3.3 illustrates the new edges from the sets $\mathcal{E}^4_{\mathcal{P},\mathcal{G}}(X), \mathcal{E}^5_{\mathcal{P},\mathcal{G}}(X)$, and $\mathcal{E}^6_{\mathcal{P},\mathcal{G}}(X)$ and shows which program points are linked by which kind of edges. We use a $findall^B$ call here, which backtracks over all solution of the predicate g/2. The literal is partitioned in the to program points (1,2) (before the meta-call), (1,3) (after the meta-call) and (1,4) (after the unification of answer set and the variable Bag. The substitutions resulting from edge set ((1,2)$\leftarrow\!\bullet$(4,1)) are applied on Bag and unified with the list of renamed answer substitutions of the goal g(A,B), which are associated with exit edges ((1,3)$\leftarrow\!\bullet$(2,1)) and ((1,3)$\leftarrow\!\bullet$(3,2)).

Figure 3.3: Extended call graph example

3.2.1 *Concrete Semantics*

We now extend the concrete semantics $F_{\mathcal{P},\mathcal{G}}(X)$ by $F^f_{\mathcal{P},\mathcal{G}}(X)$. $F^4_{\mathcal{P},\mathcal{G}}$ is similar to $F^1_{\mathcal{P},\mathcal{G}}$, but instead of positive calls edges from $\mathcal{E}^1_{\mathcal{P},\mathcal{G}}$ it describes the meta-call of a positive literal in the second argument of $findall^B$ literals. $F^5_{\mathcal{P},\mathcal{G}}$ handles the exits to the goals and is based on the definition of $F^2_{\mathcal{P},\mathcal{G}}$, but additionally it renames the variables in the substitution resulting from unification of $H_{q[1]}$ and B_{p-}. The renaming emulates the $findall^B$ semantics, which ensures distinct variable sets in the set of answer substitutions of the goal B_{p-}. And the renaming ensures that no variable binding in the clause $C_{q[1]}$ is affected.

$F^6_{\mathcal{P},\mathcal{G}}$ collects all renamed substitutions σ_i from $F^5_{\mathcal{P},\mathcal{G}}$, which are the top elements on a common stack $\parallel q^- \leftarrow\!\bullet v, \eta \parallel \cdot S$. $F^6_{\mathcal{P},\mathcal{G}}$ applies them all to the template term $Tl_{p\leftarrow\bullet q}$ and unifies the resulting list with Bag_p. $F^6_{\mathcal{P},\mathcal{G}}$ collects all permutations of subsets of the lists of lists of stacks$\parallel p \leftarrow\!\bullet q, \sigma_i \parallel \cdot \parallel q^- \leftarrow\!\bullet v, \eta \parallel \cdot S$, with $i \in [1, n]$, and $n = |\{\parallel p \leftarrow\!\bullet q, \sigma_1 \parallel \cdot \parallel q^- \leftarrow\!\bullet v, \eta \parallel \cdot S, \ldots\}|$, the number of stacks ending in the two given top elements. This emulates the semantics of $findall^B$, except for the empty list case. This special case is defined in the set $F^7_{\mathcal{P},\mathcal{G}}(X)$.

$F^6_{\mathcal{P},\mathcal{G}}$ ignores the order of substitutions σ_i, **S** contains all lists with all permutations of $[Tl_{p\leftarrow\bullet q}\sigma_1, \ldots]$. The transition system could be extended to preserve the order, but this precision would be removed in the abstracted semantics anyway. Therefore we kept this imprecision already in the concrete semantics, although it is, strictly speaking, no approximation of the $findall^B$ backtracking semantics, but a superset of possible bindings.

$$F^f_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \bigcup_{0 \leqslant j \leqslant 7} F^j_{\mathcal{P},\mathcal{G}}(X) \tag{3.23}$$

$$F^4_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \frac{\dfrac{\parallel p \leftarrow\!\bullet q, \theta \parallel}{\parallel q \leftarrow\!\bullet u, \sigma \parallel}}{S} \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \cdot \in \mathcal{E}^4_{\mathcal{P},\mathcal{G}} \\ \wedge & L_q \equiv findall(Tl_{q^+\leftarrow\bullet q}, B_q, Bag_{q^+}) \\ \wedge & \parallel q \leftarrow\!\bullet u, \sigma \parallel \cdot S \in X \\ \wedge & \theta = unify(B_q, \sigma, H_{p[1]}, \epsilon) \neq fail \end{array} \right\} \tag{3.24}$$

$$F^5_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \frac{\dfrac{\parallel p \leftarrow\!\bullet q, \theta \parallel}{\parallel q \leftarrow\!\bullet v, \eta \parallel}}{S} \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \cdot \in \mathcal{E}^5_{\mathcal{P},\mathcal{G}} \\ \wedge & \dfrac{\dfrac{\parallel p \leftarrow\!\bullet u, \sigma \parallel}{\parallel q \leftarrow\!\bullet v, \eta \parallel}}{S} \in X \\ \wedge & L_q = findall(Tl_{p\leftarrow\bullet q}, B_q, Bag_p) \\ \wedge & \theta = unify(H_{q[1]}, \sigma, B_{p^-}, \eta) \circ \rho \neq fail \\ \wedge & \rho\, renaming\, substitution \end{array} \right\} \tag{3.25}$$

$$F^6_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \parallel p^+ \leftarrow\!\bullet p, \theta \parallel \cdot S \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \cdot \in \mathcal{E}^5_{\mathcal{P},\mathcal{G}} \\ \wedge & L_q = findall(Tl_{p\leftarrow\bullet q}, B_q, Bag_p) \\ \wedge & \dfrac{\dfrac{\parallel p \leftarrow\!\bullet q, \sigma_1 \parallel}{\parallel q^- \leftarrow\!\bullet v, \eta \parallel}}{S} \in X \\ & \vdots \\ \wedge & \dfrac{\dfrac{\parallel p \leftarrow\!\bullet q, \sigma_k \parallel}{\parallel q^- \leftarrow\!\bullet v, \eta \parallel}}{S} \in X \\ \wedge & \theta = unify([Tl_{p\leftarrow\bullet q}\sigma_1\rho_1, \ldots], \epsilon, Bag_p, \eta) \\ \wedge & \theta \neq fail \\ \wedge & \rho_i\, renaming\, substitutions \end{array} \right\} \tag{3.26}$$

$$F^7_{\mathcal{P},\mathcal{G}}(X) \stackrel{def}{=} \left\{ \parallel p^+ \leftarrow\!\bullet p, \theta \parallel \cdot S \;\middle|\; \begin{array}{ll} & q^- \leftarrow\!\bullet v \cdot \in \mathcal{E}^4_{\mathcal{P},\mathcal{G}} \\ \wedge & \dfrac{\parallel q^- \leftarrow\!\bullet v, \eta \parallel}{S} \in X \\ \wedge & L_q = findall(Tl_{p\leftarrow\bullet q}, B_q, Bag_p) \\ \vee & \theta = unify([], \epsilon, Bag_p, \eta) \end{array} \right\} \tag{3.27}$$

3.2.2 *Collecting Semantics*

The $findall^B$ abstract semantics $F^f_{\mathcal{P},\mathcal{G}}(X)$ is extended into the $findall^B$ collection semantics $[F^{\#f}_{P,G}]_{p\leftarrow\!\bullet q}$:

$$[F^{\#f}_{P,G}(X^\#)]_{p\leftarrow\!\bullet q} \stackrel{def}{=} \tag{3.28}$$

$$[F^{\#}_{P,G}(X^\#)]_{p\leftarrow\!\bullet q} \tag{3.29}$$

$$\cup \left\{ unify^\#(B_q, X^\#_{q\leftarrow\!\bullet u}, H_{p[1]}, \{\epsilon\}) \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \in \mathcal{E}^4_{P,G} \\ \wedge & q \leftarrow\!\bullet u \in \mathcal{E}_{P,G} \\ \wedge & L_q \equiv findall(Tl_{q^+\leftarrow\!\bullet q}, B_q, Bag_{q^+}) \end{array} \right\} \tag{3.30}$$

$$\cup \left\{ \Theta \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \in \mathcal{E}^5_{P,G} \\ \wedge & p^- \leftarrow\!\bullet v\cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & q \leftarrow\!\bullet u\cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & L_{p^-} \equiv findall^B(Tl_{p\leftarrow\!\bullet p^-}, B_{p^-}, Bag_p) \\ \wedge & \Theta = unify^\#(H_{q[1]}, X^\#_{q\leftarrow\!\bullet u}, B_{p^-}, X^\#_{p^-\leftarrow\!\bullet v}) \end{array} \right\} \tag{3.31}$$

$$\cup \left\{ \theta^\# \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \in \mathcal{E}^6_{P,G} \\ \wedge & q \leftarrow\!\bullet u \in \mathcal{E}^5_{P,G} \\ \wedge & p^- \leftarrow\!\bullet v \in \mathcal{E}_{P,G} \\ \wedge & \{\sigma_1, \ldots, \sigma_k\} \subseteq X^\#_{q\leftarrow\!\bullet u} \\ \wedge & L_{p^-} \equiv findall^B(Tl_{p\leftarrow\!\bullet p^-}, B_{p^-}, Bag_p) \\ \wedge & \theta^\# = unify^\#([Tl_{p\leftarrow\!\bullet q}\sigma_1\rho_1, \ldots], \{\epsilon\}, Bag_p, X^\#_{p^-\leftarrow\!\bullet v}) \\ \wedge & \rho_i \; renaming \; substitutions \end{array} \right\} \tag{3.32}$$

The subformulas 3.30 to 3.31 are adaptions of the positive call and exit subformulas in the definition of $F^\#_{P,G}(X^\#)$. The subformula 3.32 abstracts from the sets $F^6_{\mathcal{P},\mathcal{G}}(X)$ and edges, by selecting all subsets, including the empty set (and is therefore subsuming $F^7_{\mathcal{P},\mathcal{G}}(X)$), of the possible bindings σ_i from $X^\#_{q\leftarrow\!\bullet u}$, defined by the subformula 3.31. The variables in σ_i are renamed by disjoint renaming substitutions ρ_i to reflect the renaming substitution from 3.32.

Lemma 3.2.3. $lfpF^f_{\mathcal{P},\mathcal{G}}(X) \subseteq F^{\#f}_{\mathcal{P},\mathcal{G}}(X)$

The proof of $lfpF_{\mathcal{P},\mathcal{G}}(X) \subseteq F^\#_{\mathcal{P},\mathcal{G}}(X)$ in [130], Lemma 4.2.3, can be directly applied to $lfpF^f_{\mathcal{P},\mathcal{G}}(X) \subseteq F^{\#f}_{\mathcal{P},\mathcal{G}}(X)$.

3.2.3 *Abstract Semantics*

To use the existing $unify^\flat$ algorithm for the $findall^B$ call we need to abstract from the list of bindings, since $unify^\flat$ expects two fixed terms. We need to find an abstraction for the list $[Tl_{p\leftarrow\!\bullet q}\sigma_1, \ldots]$ with a fixed term that abstracts from
$unify^\#([Tl_{p\leftarrow\!\bullet q}\sigma_1, \ldots], \{\epsilon\}, Bag_p, X^\#_{p^-\leftarrow\!\bullet v})$.

We will exploit a beneficial property of $unify^\flat$ for the mode abstraction domain in the following lemma:

Lemma 3.2.4. *Let* $t \in T, Bag \in LVars$ *and* $t_i = t\rho_i, \theta^\flat_i = \theta^\flat\rho_i$, *with* $i \in [1,..,n], n > 0$ *and* ρ_i *renaming substitutions with* $\forall_{i,j} i \neq j \Rightarrow vars(\theta^\flat_i) \cap vars(\theta^\flat_j) = \varnothing$, then $unify^\flat([t], \theta^\flat, Bag, \sigma^\flat) = unify^\flat([t_1, ..., t_n], \theta^\flat_1 \circ ... \circ \theta^\flat_n, Bag, \sigma^\flat)$.

The call of $unify^\flat(t', \theta^\flat, Bag, \epsilon)$ calculates the mgu of Bag and t' and normalizes the result to equations in solved form. Since Bag is a variable, this set is $Eo = \{(Bag = t')\}$. In the following we can ignore aliasing and sharing, since the resulting substitution is restricted to the variable list $\{Bag\}$. So we only need to consider abstract unification of the abstract instantiations, reducing the problem to the equality of $est^m(Bag = [tl], \theta^\flat \downarrow Inst) \stackrel{?}{=} est^m(Bag, [tl_1, ..., tl_n], \theta^\flat_1 \downarrow Inst \circ \ldots \circ \theta^\flat_n \downarrow Inst)$. From its definition this is equivalent to $mo^\flat([tl], \theta^\flat \downarrow Inst) \stackrel{?}{=} mo^\flat([tl_1, ..., tl_n], \theta^\flat_1 \downarrow Inst \circ ... \circ \theta^\flat_n \downarrow Inst)$, since $\mathcal{I}(Bag)$ is the same in both cases. We now consider all four conditions of $mo^\flat(t, \mathcal{I})$. Its definition is repeated below for easier reference:

$$mo^\flat(t, \mathcal{I}) \stackrel{def}{=} \begin{cases} \mathcal{I}(t) & if\ t \in \mathcal{V} \\ \varnothing & if\ \exists X \in vars(t).\mathcal{I}(X) = \varnothing \\ \{g\} & if\ \forall X \in vars(t).\mathcal{I}(X) = \{g\} \\ \{g, o\} & if\ \forall X \in vars(t).(g \in \mathcal{I}(X)) \\ \{o\} & otherwise \end{cases}$$

The first conditions, $t \in \mathcal{V}$, is never true for both lists. The second condition is equivalent for both lists, since if there is a variable with an impossible mode ($\varnothing$) in $vars(t)$, there are at least n variables in $[tl_1, ..., tl_n]$ with this abstract state. The same consideration can be made for the third and the fourth condition. The universal quantifier applied on duplicated terms has the same semantics.

□

But, the second and fourth condition need the prerequisite on the list size $n > 0$. In case that $t = []$, the second condition can never be true and the fourth condition is always true, which is apparently not true in general for arbitrary $[tl]$ terms. We will now define the $findall^B$-extended abstract semantics $F^{\flat f}_{P,G}$ ignoring this restriction at first:

$$[F^{\flat f}_{P,G}(X^\flat)]_{p \leftarrow\bullet q} \stackrel{def}{=}$$

$$[F^{\flat}_{P,G}(X^\flat)]_{p \leftarrow\bullet q} \tag{3.33}$$

$$\sqcup [\![\mathcal{V}_{p[1]}]\!] \left\{ \theta^\flat \;\middle|\; \begin{array}{ll} & p \leftarrow\bullet q \in \mathcal{E}^4_{P,G} \\ \wedge & q \leftarrow\bullet u \in \mathcal{E}_{P,G} \\ \wedge & L_q \equiv findall^B(Tl_{q^+ \leftarrow\bullet q}, B_q, Bag_{q^+}) \\ \wedge & \theta^\flat = unify^\flat(B_q, X^\flat_{q \leftarrow\bullet u}, H_{p[1]}, \epsilon^\flat [\![\mathcal{V}_{p[1]}]\!]) \end{array} \right\} \tag{3.34}$$

$$\sqcup [\![\mathcal{V}_{p[1]}]\!] \left\{ \theta^\flat \;\middle|\; \begin{array}{ll} & p \leftarrow\bullet q \in \mathcal{E}^5_{P,G} \\ \wedge & p^- \leftarrow\bullet v \cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & q \leftarrow\bullet u \cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & L_{p^-} \equiv findall^B(Tl_{p \leftarrow\bullet p^-}, B_{p^-}, Bag_p) \\ \wedge & \theta^\flat = unify^\flat(H_{q[1]}, X^\flat_{q \leftarrow\bullet u}, B_{p^-}, X^\flat_{p^- \leftarrow\bullet v}) \end{array} \right\} \tag{3.35}$$

$$\sqcup [\![\mathcal{V}_{p[1]}]\!] \left\{ \theta^\flat \;\middle|\; \begin{array}{ll} & p \leftarrow\bullet q \in \mathcal{E}^6_{P,G} \\ \wedge & q \leftarrow\bullet p^- \in \mathcal{E}^5_{P,G} \\ \wedge & p^- \leftarrow\bullet v \in \mathcal{E}_{P,G} \\ \wedge & L_{p^-} \equiv findall^B(Tl_{p \leftarrow\bullet p^-}, B_{p^-}, Bag_p) \\ \wedge & \eta^\flat = unify^\flat([Tl_{p \leftarrow\bullet q}], X^\flat_{q \leftarrow\bullet p^-}, Bag_p, X^b_{p^- \leftarrow\bullet v}) \\ \wedge & \zeta^\flat = unify^\flat([], X^\flat_{q \leftarrow\bullet p^-}, Bag_p, X^b_{p^- \leftarrow\bullet v}) \\ \wedge & \theta^\flat = \zeta^\flat \sqcup [\![\mathcal{V}_{p[1]}]\!] \eta^\flat \end{array} \right\} \tag{3.36}$$

We extended the abstract semantic $[F^\flat_{P,G}]_{p \leftarrow\bullet q}$ (row 3.33) by abstractions of the $findall^B$ meta call, the exit of the call and unification of the list of bindings and the Bag_p variable. The meta-call in row 3.34 is equivalent to the common positive literal call 3.15. The call exit (row 3.35) is similar to the return of the positive literal (see sub equation 3.16), but additionally renames the variables with ρ. The set 3.36 abstracts from 3.32. For substitutions complying to the antecedent of Lemma 3.2.4 $\eta^\flat$ is a valid abstraction. Additionally we need to consider the case of an empty set of substitutions in sub-equation 3.32. In this case Bag_p is unified with an empty list resulting in a ground mode. Therefore we can define a discrete abstract substitution $\zeta^\flat$ for this case.

Theorem 3.2.5. $lfpF^{\#f}_{P,G} \sqsubseteq lfpF^{\flat f}_{P,G}$.

We base the proof for theorem 3.2.5 on [130, theorem 4.3.3], which proofs $lfpF^{\#}_{P,G} \sqsubseteq lfpF^{\flat}_{P,G}$, the general case for generic abstract substitutions. The antecedents of theorem 4.3.3 are already fulfilled, since we restricted $F^{\flat f}_{P,G}$ to the mode analysis abstract domain.

Lu reduced the proof of $lfpF^{\#}_{P,G} \sqsubseteq lfpF^{\flat}_{P,G}$ to the poof that $X^\flat \in D^\flat, F^{\#}_{P,G} \cdot \gamma^\flat(X^\flat) \sqsubseteq^{\#} \gamma^\flat \cdot F^\flat_{P,G}(X^\flat)$, the same consideration follow for the findall extension. Let $\sigma \in [F^{\#f}_{P,G} \cdot \gamma^\flat(X^\flat)]_{p \leftarrow\bullet q}$. We need to proof that $\sigma \in [\gamma^\flat \cdot F^{\flat f}_{P,G}(X^\flat)]_{p \leftarrow\bullet q}$. The proof of the meta-calls (3.34) follows the proof of the abstraction of the subformula for positive calls 3.15 and the proof for clause exits (3.35) are equivalent to the exits of positive literals (3.16). The case left to proof is subformula 3.36. We divide it into two parts. The first for non-empty answer substitution lists **(a)**, the second for the empty list **(b)**.

a) First, we transform the substitutions of the collection semantics, where σ_i are substitutions and ρ_i are disjoint renaming substitutions, with $i \in [1, n]$:

$$unify^{\#}([Tl_{p\leftarrow\bullet q}\sigma_1\rho_1,\ldots],\{\epsilon\},Bag_p,X^{\#}_{p-\leftarrow\bullet v}) =$$
$$unify^{\#}([Tl_{p\leftarrow\bullet q}\rho_1,\ldots],\{\sigma_1\rho_1\circ\ldots\},Bag_p,X^{\#}_{p-\leftarrow\bullet v})$$

Now we consider all permutations of subsets of the set of all substitutions $\sigma_i \in X^\flat_{q\leftarrow\bullet p-}$:

$$\begin{aligned}
\sigma \;\in\; & (\,unify^{\#}([Tl_{p\leftarrow\bullet q}\rho_1],\gamma[\![\mathcal{V}_{vars(\rho_1)}]\!](X^\flat_{q\leftarrow\bullet p-}\rho_1),Bag_p,\gamma[\![\mathcal{V}_{p[1]}]\!](X^\flat_{p-\leftarrow\bullet v})) && (3.37)\\
\cup\;\; & \vdots \\
\cup\;\; & unify^{\#}([Tl_{p\leftarrow\bullet q}\rho_1,\ldots,Tl_{p\leftarrow\bullet q}\rho_n],\gamma[\![\mathcal{V}_{vars(\rho_1,\ldots,\rho_n)}]\!](X^\flat_{q\leftarrow\bullet p-}\rho_1\circ\ldots\circ X^\flat_{q\leftarrow\bullet p-}\rho_n),\\
& \qquad Bag_p,\gamma[\![\mathcal{V}_{p[1]}]\!](X^\flat_{p-\leftarrow\bullet v})\,)\\
\subseteq\;\; & (\,\gamma[\![\mathcal{V}_{p[1]}]\!]\cdot unify^\flat([Tl_{p\leftarrow\bullet q}\sigma_1\rho_1],X^\flat_{q\leftarrow\bullet p-}\rho_1,Bag_p,X^b_{p-\leftarrow\bullet v}) && (3.38)\\
\cup\;\; & \vdots \\
\cup\;\; & \gamma[\![\mathcal{V}_{p[1]}]\!]\cdot unify^\flat([Tl_{p\leftarrow\bullet q}\sigma_1\rho_1,\ldots],(X^\flat_{q\leftarrow\bullet p-}\rho_1)\circ\ldots\circ(X^\flat_{q\leftarrow\bullet p-}\rho_n),\\
& \qquad Bag_p,X^b_{p-\leftarrow\bullet v})\,)\\
=\;\; & \gamma[\![\mathcal{V}_{p[1]}]\!]\cdot unify^\flat([Tl_{p\leftarrow\bullet q}],X^\flat_{q\leftarrow\bullet p-},Bag_p,X^b_{p-\leftarrow\bullet v}) && (3.39)\\
\subseteq\;\; & \gamma[\![\mathcal{V}_{p[1]}]\!]([(F^{\flat f}_{P,G}(X^\flat)]_{p\leftarrow\bullet q}) && (3.40)\\
=\;\; & [\gamma^\flat\cdot F^{\flat f}_{P,G}(X^\flat)]_{p\leftarrow\bullet q} && (3.41)
\end{aligned}$$

The equation 3.39 is true by lemma 3.2.4.

b) Now we consider the empty list unification with Bag_p:

$$\begin{aligned}
\sigma \;\in\; & unify^{\#}([],\gamma^\flat[\![\mathcal{V}_{p[1]}]\!](\epsilon^\flat[\![\mathcal{V}_{p[1]}]\!]),Bag_p,\gamma^\flat[\![\mathcal{V}_{p[1]}]\!](X^\flat_{p-\leftarrow\bullet v}))\\
\subseteq\;\; & \gamma[\![\mathcal{V}_{p[1]}]\!]\cdot unify^\flat([],\epsilon^\flat[\![\mathcal{V}_{p[1]}]\!],Bag_p,X^\flat_{p-\leftarrow\bullet v})\\
\subseteq\;\; & \gamma[\![\mathcal{V}_{p[1]}]\!]([(F^\flat_{P,G}(X^\flat)]_{p\leftarrow\bullet q})\\
=\;\; & [\gamma^\flat\cdot F^\flat_{P,G}(X^\flat)]_{p\leftarrow\bullet q}
\end{aligned}$$

□

3.2.4 *Example*

The mode analysis algorithm is applied by fixpoint iteration of $unify^\flat$ and the least-upper bound calculation $\sqcup[\![\mathcal{V}_{p[1]}]\!]$ on the abstract substitutions at every program point. We illustrate with an example how this is applied for the following goal containing a findall meta-call:

```
findall(t(A,B),c(A,B),
  C)
```

We formatted the call with an additional line break to make space for the additional program point after the call of `c/2`. Let's assume only one fact for the predicate `c/2` is defined:

```
c(1,_)
```

The goal is called with the abstract substitution

```
?-
% <{A/{f},B/{f},C/{f}},{},{}>
  findall(t(A,B),c(A,B),
  C).
```

In a first step c(A,B) is unified with c(1,D), by application of rule 3.34. Since `B` has mode {f}, the mode of `D` is {f}:

```
c(1,D).
      %[D/{f}],[],[].
```

Now the exit edge 3.21 in the program graph is considered. The rule 3.35 is applied and c(1,D) is unified with c(A,B), resulting in mode {g} for A:

```
?-
  findall(t(A,B),c(A,B),
    % <{A/{g},B/{f}},{},{}>
  C).
```

At a last step rule 3.36 is applied. Two abstract unifications are evaluated. The first case considers a successful call of c, the second the failing case. As a first step $unify^\flat$ renames the variables of the first term and its substitution and unifies it with the abstract unification at the program point before the findall call. This reflects the findall semantics, where the evaluation of the goal in the second argument does not influence the binding of the variables at the call site. Now we will evaluate the unifications and the least upper bound of the resulting abstract substitutions:

$$
\begin{aligned}
\eta^\flat &= unify^\flat(<\{A2/\{g\},B2/\{f\},C2/\{f\}\},\{\},\{\}>,[c(A2,B2)],\\
&\qquad\quad <\{A/\{f\},B/\{f\},C/\{f\}\},\{\},\{\}>,C)\\
&= <\{A/\{f\},B/\{f\},C/\{o\}\},\{\},\{\}>\\
\zeta^\flat &= unify^\flat(<\{A2/\{g\},B2/\{f\},C2/\{f\}\},\{\},\{\}>,[],\\
&\qquad\quad <\{A/\{f\},B/\{f\},C/\{f\}\},\{\},\{\}>,C)\\
&= <\{A/\{f\},B/\{f\},C/\{g\}\},\{\},\{\}>\\
\theta^\flat &= \zeta^\flat \sqcup [\![\mathcal{V}_{p[1]}]\!]\eta^\flat\\
&= <\{A/\{f\},B/\{f\},C/\{g,o\}\},\{\},\{\}>
\end{aligned}
$$

So the mode for C is over-approximated with {g,o}, because the algorithm must assume that `c` might fail, resulting in an empty list, with mode {g}. The program point after the findall call has not been considered before, and is initialized with $\perp$. Since
$<\{A/\{f\},B/\{f\},C/\{g,o\}\},\{\},\{\}> \sqcup [\![\mathcal{V}_{p[1]}]\!]\perp = <\{A/\{f\},B/\{f\},C/\{g,o\}\},\{\},\{\}>$
we have evaluated the abstract substitution for the program point. The fixpoint algorithm stops here, because the edges in the program graph do not change anymore. The resulting program graph with modes is the following:

```
?-
    % <{A/{f},B/{f},C/{f}},{},{}>
  findall(t(A,B),c(A,B),
    % <{A/{g},B/{f},C/{f}},{},{}>
  C).
    % <{A/{f},B/{f},C/{g,o}},{},{}>
```

3.2.5 *Computational Complexity*

The computational complexity of the fixpoint iteration algorithm is not effected by the findall extension. The complexity of the basic algorithm is $O(dmax \times \#\aleph_C \times pmax^3)$ [130,

4.3.4] of $unify^\flat$ evaluations, where $pmax$ is the maximum number of predecessors of program points, $\#\aleph_C$ number of clauses and $dmax$ is the maximum height of the Hasse diagram for $Sub^\flat[\![\mathcal{V}_i]\!]$ for $i \in \aleph$. The complexity considerations are based on [152], which states that the worst case computing costs of the $lfpF^\flat_{\mathcal{P},\mathcal{G}}$ is proportional to the product of $dmax$ and the number of operations in $F^\flat_{\mathcal{P},\mathcal{G}}$. Lu estimated the worst case number of operation by $\#\aleph_C \times pmax^3$, which is the estimation of evaluations in 3.16, the most complex rule in $F^\flat_{\mathcal{P},\mathcal{G}}(X)$. For every program at the exit of a call up to $pmax^2$ $unify^\flat$ operations have to be evaluated. The call in the findall meta-call (3.35) has the same complexity. In rule 3.36, the aggregation of call results, only two calls of $unify^\flat$ occur. So the complexity of the algorithm is not affected.

3.2.6 *Application to Further All-Solutions Predicates*

The all-solution predicates bagof/3 and setof/3 are very similar in their semantics to findall, but need further considerations in the mode inference. We will not extend the given semantics in all steps, but rather illustrate the extensions that have to be made. In the following we only refer to bagof. The same principle applies for the setof predicate.

First, we ignore existential variables. They are syntactic sugar and can be realized by forwarding predicates, which omit the existential variables O'Keefe [153]. The other two variable types are considered separately.

The mode of the third argument (*Bag*) can be inferred by the findall inference algorithm above with a slight modification. Bag can never be bound to the empty list []. So the corresponding $bagof^B$ predicates must only consider non-empty subsets:

$bagof^B$(Template,Literal,Bag):-
  bagof(Template,Literal,BagTmp),
  non_empty_subset(Sub,BagTmp),
  permutation(Bag,Sub).

Further, the abstract semantics $F^{\flat f}_{P,G}(X^\flat)$ must omit empty lists in the unification of the template bindings and Bag (rule 3.36), which ensures that the mode list of *Bag* in findall/3 at least contains the ground mode $\{g\}$. The rule 3.36 is replaced the simplified rule:

$$\sqcup[\![\mathcal{V}_{p[1]}]\!] \left\{ \theta^\flat \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \in \mathcal{E}^6_{P,G} \\ \wedge & q \leftarrow\!\bullet p^- \in \mathcal{E}^5_{P,G} \\ \wedge & p^- \leftarrow\!\bullet v \in \mathcal{E}_{P,G} \\ \wedge & L_{p^-} \equiv bagof^B(Tl_{p\leftarrow\bullet p^-}, B_{p^-}, Bag_p) \\ \wedge & \theta^\flat = unify^\flat([Tl_{p\leftarrow\bullet q}], X^\flat_{q\leftarrow\bullet p^-}, Bag_p, X^b_{p^-\leftarrow\bullet v}) \end{array} \right\}$$

The predicate bagof/3 backtracks over all variables that are not used in the template argument. So, we can apply the normal mode analysis for these variables. To keep the modification of the algorithm minimal, we apply the following pre-processing for the mode analysis:

bagof(*Template*,*Goal*,*Bag*) $\rightarrow$ $bagof^B$(*Template*,*Goal*, *Bag*),*Goal*

But, this modification will also influence the mode of the variables in *Template*. To avoid this mode update we define the new rule 3.42, an extension of rule 3.16 that considers call exits. Rule 3.42 is applied if a literal *Goal* directly follows a bagof/3 call. In this case a renaming substitution Ψ is added that renames all variables in *Template* before it is unified with $H_{q[1]}$:

$$\sqcup[\![\mathcal{V}_{p[1]}]\!] \left\{ \theta^\flat \;\middle|\; \begin{array}{ll} & p \leftarrow\!\bullet q \in \mathcal{E}^2_{P,G} \\ \wedge & B_{p-2} = bagof^B(Template, Goal, Bag) \\ \wedge & \Psi = ren.subst.for\,vars(Template) \\ \wedge & p^- \leftarrow\!\bullet v \cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & q \leftarrow\!\bullet u \cdot \in \mathcal{E}_{\mathcal{P},\mathcal{G}} \\ \wedge & \theta^\flat = unify^\flat(H_{q[1]}, X^\flat_{q\leftarrow\bullet u}, B_{p^-}, X^\flat_{p^-\leftarrow\bullet v}\Psi) \end{array} \right\} \tag{3.42}$$

This ensures that the modes of variables in *Template* are not updated.

3.3 SUMMARY

This chapter extended the mode analysis approach by Lu [130]. The approach is based on abstract interpretation with a product domain that considers modes, variable aliasing, and variable sharing. We extended the concrete, collection, and abstract semantics with an approximation of the findall/3 Prolog predicate and have proofed that the least fixpoint over the abstract domain is a safe approximation of the concrete semantics.

Further we illustrated how the abstract semantics can be extended to cover the Prolog meta-predicates bagof/3 and setof/3, too.

4 OBJECT-ORIENTED LOGIC CONTEXT QUERY LANGUAGE

This chapter presents the specification and semantics of an Object-oriented logic Context Query Language (OCQL) that operates on RDFS-modeled context data. The OCQL is designed to be embeddable into Java-based languages. In Section 4.1 we present a mapping from RDF Schema classes to Java types, bridging the gap between these different type systems. Section 4.2 introduces OCQL which operates on the mapped Java types.

4.1 RDF SCHEMA - JAVA MAPPING

For this thesis we decided to use RDF Schema for context modeling. The Semantic Web stack provides the more expressive ontology language family OWL, but RDFS is sufficient to model a large class of context taxonomies with a small and easily understandable set of constructs. Complex relationships between concepts will be expressed on the level of predicates in the query language, see also [126]. Nevertheless, OWL context sources can be used in our approach. Under certain restrictions[1] OWL can be serialized into RDFS and combined with other RDFS graphs.

Although the RDFS class and property system is similar to the type system of object-oriented languages, it is very different in its type definition. It follows a "property-centric approach ... [, which] allows anyone to extend the description of existing resources" [44]. Properties of a class can be defined in several parts that are aggregated to form a concrete class specification. Not only the class specifications, but also the instances can be scattered over several of these parts. To bridge the conceptual gap between RDFS classes and Java types this chapter introduces a mapping.

Various proposals for mappings from Semantic Web classes to Java have been made, but all have limitations that make them unsuitable for our needs. Jena Schemagen [178] lacks static typing of properties in general. It uses a generic class to represent the set of related properties. All other approaches do not handle multiple definition of `rdfs:range` definitions correctly or fall back to an untyped solution in this case. We will now analyze this issue in more detail. RDF Schema allows the definition of more than one range declaration for a property. An instance of a property must therefore be a subtype of all range declarations. The mapper RDFReactor [202] maps each RDF class to a Java class. Since Java does not support multiple inheritance, there is no way to define subclasses for two classes that are not in an inheritance relationship.

For example, consider a property `nearbyLocation` with the `rdfs:domain Contact` and two `rdfs:range` declarations:

```
@prefix ex: <http://www.example.org/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
ex:nearbyLocation rdf:type    rdf:Property;
                  rdfs:domain ex:Contact;
                  rdfs:range  ex:Nearby;
                  rdfs:range  ex:Location.
```

Here, the property's type must be a subtype of both Nearby and Location. Except for type variables such an enumeration of types is not legal in Java member declarations, e.g., consider a method definition whose return enumerate all types separated with an '&'[2]:

1 see Section 2.2.3
2 This source code is not legal Java syntax. It is just used for illustrative purpose. We use the syntax of multiple bounds for type variables, which are separated with '&' and have similar semantics.

```
interface Contact {
  Nearby & Location nearbyLocation();
}
```

RDFReactor does not offer a solution to this issue and ignores multiple `rdfs:range` definitions. It selects the last defined range definition as the property's type. Jastor [116] uses an *untyped* list in this case, which is guarded at runtime by a *VetoableChangeListener*, a concept defined by the Java JDK [3]. Jastor uses the listener to ensure that arguments of a class's property are subtypes of all declared range definitions.

In the following, we present a mapping of RDFS class specifications to Java types, which is statically typed and considers type definitions for multiple `rdfs:range` definitions. The prerequisite for the mapping is an rdfs and xsd-consistent [98] rdf graph. Here, consistency means the graph does not contain contradicting triples and no *datatype clashes* [98, Section 5], such as a property with two disjunctive datatypes[3]. The mapping is based on the rdfs entailment[4] of a schema. The RDFS entailment for instances always contains the following triple for every URI reference XXX:

```
XXX rdf:type rdfs:Resource
```

For this reason, there are at least the following `rdfs:range` and `rdfs:domain` definitions for each property XXX:

```
XXX rdfs:range rdfs:Resource
XXX rdfs:domain rdfs:Resource
```

For the mapping and the followup chapters we define a set of terms. Let XDT be the set of predefined xml datatypes [203, Section 3], limited to the suitable types defined in [98, section 5].

Definition 4.1.1. The function $rtj_p : XDT \rightarrow JPTW$ maps all datatypes to Java primitive type wrappers classes $JPTW \subset JT$, with JT being the set of all Java types.

An example for the mapping is:

```
xsd:int -> java.lang.Integer
```

We have chosen primitive wrapper types over Java primitive types to have the common super type Object for all types in OCQL and to avoid considerations of auto-boxing [88, 5.1.7] in the type checking algorithm.

However, there is a drawback in this solution - the wrapper types are only subtypes of Number (resp. Object) and no *widening primitive conversion*[5] is applied as it is only defined for primitive types [88, 5.1.2]. Widening for wrapper classes is not possible in Java, because a conversion of a class C to class D, where D is not subtype of C, is illegal. We change the subtype relationship for primitive type wrappers here to reflect widening primitive conversion for primitive types, e.g., Integer extends Double. This allows us to apply conversions consistently with the type hierarchy. Section A.1 lists the complete mapping.

Let RS be the set of RDF schemas, RC be the set of all subclasses of `rdfs:Resource` and LVT be the set of all literal value types with $LVT := XDT \cup \{\texttt{rdfs:Literal}, \texttt{rdfs:XMLLiteral}\}$.

Let Id be the set of legal Java identifiers, RT be the set of all rdfs types, $RNVT := RT \backslash LVT \cup rdfs : Resource$ the set of RDF non-literal classes and JI the set of Java types representing types from $RNVT$. Figure 4.1 illustrates how these sets are related to each other.

3 for example, xsd:string and xsd:int
4 see Section 2.2.2
5 If the domain of a primitive type is a subset of another type it is *convertible* to this type. For example, an int value can be assigned to a double variable and is converted to a double value.

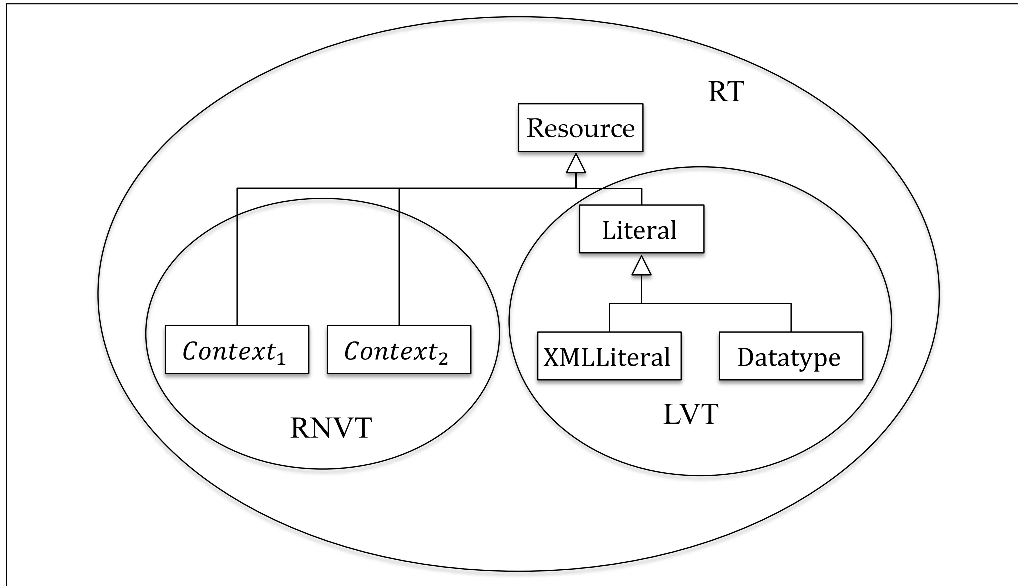


Figure 4.1: RDFS classes considered subsets

$C_i \in RNVT$ denotes an RDF class, $I_i \in JI$ the corresponding Java interface for C_i, in which the index i links the RDFS class with its Java interface. With p we refer to an RDF property and $p_{T_i}, i = 1, .., n$ denotes the RDF types of p. The corresponding Java method representing p is denoted with $m_p : \tau$ a, where τ stands for return type of the method. We write $C <: D$ when C is a subtype of D. The subtype relation is a *partial order* - the reflexive, anti-symmetric and transitive closure of the `rdfs:subclassOf` relationship.

In the following, we will refer to the usage of the types as the *importing context*. The specification of an RDFS class can be distributed among several RDF schemas, of which only a subset $S \subset RS$ may be imported by context C. C is type checked against the RDFS class fragments defined in its schema import S. The types described in the following are therefore not necessary globally defined, but only valid for the imported schemas in the scope of an importing context.

4.1.1 *Namespace Binding*

We adopted the prefix namespace binding from XML namespaces [43] in order to distinguish class and property names with different namespaces. Prefixes are defined in the header of the OCQL expression similar to xml namespace definitions, but we employ a more Java-like syntax:

```
namespace <ns> = "<URI Prefix>";
```

The namespace *ns* is mapped to a Java package, which can be imported as a regular Java package. The following expression imports all RDF classes of the namespace *ex*:

```
namespace ex = "http://www.example.com";
import ex.*;
```

4.1.2 *Class and Property Mapping*

In a first step, we describe the mapping of a single RDFS class to Java interfaces and their subclass relationships. We will also refer to mapped classes as *context classes*.

We denote this base set of Java interfaces $JI_b \subseteq JI$. The function $rtj : RNVT \rightarrow JI_b$ map a non-literal (uri-referenced) class C_i to Java interface I_i with the same simple name and

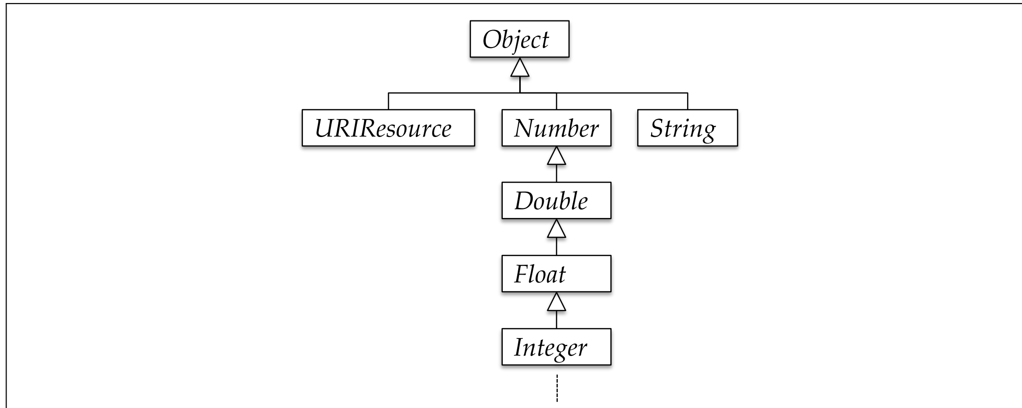


Figure 4.2: Hierarchy of mapped Java objects

package name following the namespace mapping defined in the Section 4.1.1. The simple names of classes and properties must be in *Id*.

The restriction to legal Java identifiers restricts the set of mappable RDF schemas. Legal URIs in RDF are XML qualified names (QName) as defined in [43, 4]. QName allows a superset of Java identifier strings in its local part: Prefix:LocalPart., e.g., `ns:local-name` is a valid QName. This illegal characters could be quoted in a Java identifier, but we ignore this issue in the following for simplicity reasons.

Figure 4.2 illustrates the target Java class hierarchy JT of the mapping. The class `rdfs:Resource` is mapped to java.lang.Object, the common super type for all types. Additionally, we introduce a common super interface for all non-literal classes, the interface `rdfs.URIResource`. It inherits all properties whose `rdfs:domain` is `rdfs:Resource`. This common super type is necessary because java.lang.Object cannot be extended with additional methods. Since RDF literals cannot have properties [98, section 4.3] this aligns well with the RDF semantics.

The following, preliminary mapping has the prerequisite that exactly one `rdfs:range` is defined for each property. The case of more than one `rdfs:range` definition is considered in Section 4.1.2.

All properties p and sub-properties[6] defined by `rdfs:domain` definitions are mapped to methods m_p with an array return type. The methods always have an array return type because rdfs does not have the means to declare multiplicities on properties. Therefore we have to assume that an arbitrary number (0..r) of properties are defined.

The name of the method equals the simple property name if there are no ambiguities (see Section 4.1.2) and the component type is the mapped `rdfs:range` type. Each C_1 `rdfs:subclassOf` C_2 property is mapped to the I_1 `extends` I_2 relationship. The predefined property `source` links each uri-referenced class to the class `Source`:

```
@prefix core:
    <http://sam.iai.uni-bonn.de/rdf/core.rdfs#>.
core:Source rdf:type rdfs:Class.
core:source rdf:type rdf:Property;
rdfs:domain rdfs:Resource;
rdfs:range core:Source.
```

Context sources, which are further elaborated in the context management Section 5.1, may additionally define arbitrary meta data, e.g., accuracy, in their RDF Schema definition. This meta-data is added as properties to the `Source` class by convention.

Figure 4.3 illustrates this with the class `Contact` and one property `surname` with `rdfs:range xsd:string`. RDF containers and collections are represented in the same

6 `rdfs:subPropertyOf`

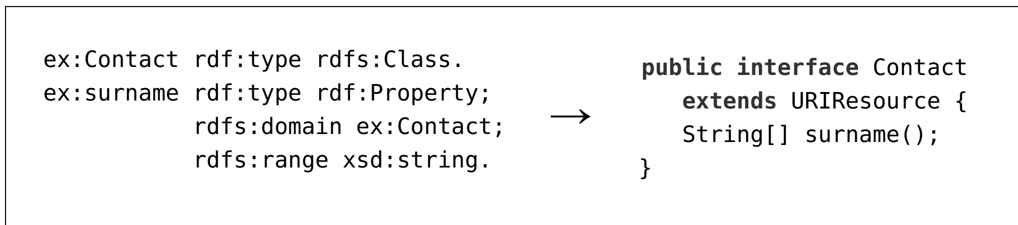


Figure 4.3: Exemplary mapping of the rdfs class `Contact`

way, but annotated as collection properties[7]. Since `rdfs:range` definitions do not allow for parametric polymorphism the return type of the method always is `Object[]`.

Namespaces and Property Names

Property name ambiguities may occur if two or more properties share the same domain and simple name, but have different namespaces. A property access *v.<propname>()* is not well defined in this case. Therefore we qualify the property names in case of ambiguities with the corresponding namespace. This is only necessary if properties with the same name are defined in the same domain.

The following example illustrates the use of a namespace prefix for a property name. The local name `ex` of the prefix and the separator character `$` are prefixed to the property name.

```
namespace ex = "http://www.example.com";
public aspect A {
  ... var = context.ex$PropertyName();
}
```

Multiple range definitions

Rdf schema allows the definition of more than one `rdfs:range` [45], with the following restriction:

> "Where [predicate] P has more than one `rdfs:range` property, the resources denoted by the objects of triples with predicate P are instances of all the classes stated by the `rdfs:range` properties".

The simple mapping of rdf properties above will not work in case of multiple `rdfs:range` definitions, if the types are not in a subtype relationship. For example, consider the two classes `geo:SpacialThing` `ex:Place` representing a geo location and place:

```
@prefix ex:  <http://www.example.com>.
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>.
ex:spacialPlace rdf:type rdf:Property;
                rdfs:domain rdfs:Resource;
                rdfs:range geo:SpacialThing;
                rdfs:range ex:Place.
```

The property `spacialPlace` declares its objects to be of type SpacialThing and Place. We cannot assign a type for the corresponding method, yet, because no subtype of both is defined in JI_b. Let's assume a property p has several range definitions p `rdfs:range` p_{T_1}, p `rdfs:range` p_{T_2},... . To assign a Java type to such a property we extend the JI_b type hierarchy with types JI_e, containing interfaces representing sets of interfaces from JI_b with the following properties:

7 They will be handled differently by the context query language, see Section 4.2 for details.

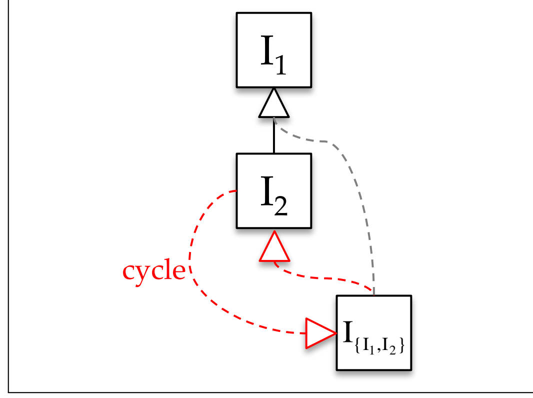


Figure 4.4: Cycle in case of naive powerset mapping. The dotted lines are subtype relationships introduced by $E_{\wp}$.

Definition 4.1.2. Let JI_e be the set of *set interfaces* $I_S \in JI_e$ with $S \subseteq JI_b$. JI_b is embedded into JI_e in the following way: $JI_b \subset JI_e$, where $\forall I \in JI_b, I_{\{I\}} \in JI_e : I = I_{\{I\}}$. We define a subtype relationship $<:_e$ on JI_e with the following properties:

1. $I_S <:_e I_{S'} \Leftrightarrow \forall I' \in S' : \exists I \in S : I <: I'$
2. $< JI_e, <:_e >$ is a partial order

   The subtype relationship $<:$ in property 1 is defined by the extends relationship on the interfaces in JI_b. Property 2 states that the set interfaces represent a taxonomy, especially that $<:_e$ must not contain non-reflexive cycles.

A self-evident starting point for the construction of JI_e is the power set of JI_b:

Definition 4.1.3. *Let $G_{\wp}$ be a directed graph induced by the power set of JI_B and the subtype relationship in JI_B,* where *$G_{\wp} =< \wp(JI_b), E_{\wp} >$, with*
$E_{\wp} = \{(S, S') \in \wp(JI_b) \times \wp(JI_b) \,|\forall I' \in S' : \exists I \in S : I <: I' \}$.

Lemma 4.1.4. *I_S with $S \in \wp(JI_b)$ and subtype relationship $E_{\wp}$* fulfills property 1 from definition *4.1.2*.

Proof: Directly from its definition of $E_{\wp}$.

$G_{\wp}$ does not fulfill property 2 from definition 4.1.2. Every subtype relationship in JI_b induces a cycle in $E_{\wp}$. Figure 4.4 illustrates this issue. Every real subtype relationship in JI_b induces a strongly connected component (SCC) in $G_{\wp}$, and therefore illegal cycles in the subtype relationship. The reason is that set interfaces contain a subtype interface I and $I's$ supertypes are equivalent with the interface I itself. In this context equivalent means that they represent the same set of RDFS classes. To remove these cycles we need remove the SCCs from the $G_{\wp}$.

Definition 4.1.5. The function $st(\wp(JI_e)) \rightarrow \wp(JI_e)$ returns the supertype closure of a set of types including the types themselves: $st(S) \overset{def}{=} \{r|s \in S \wedge r \in JI_b \wedge s <: r\}$.

Lemma 4.1.6. *A cycle between two set interfaces I_S and $I_{S'}$ exists if and only if $\forall I \in S : \exists I' \in S' : I' <: I \wedge \forall I' \in S' : \exists I \in S : I <: I'$.*

Proof: From right to left the proof is directly from definition 4.1.3. For the left to right case let's assume there is a cycle between two interfaces I_S and $I_{S'}$ with $S \neq S'$. To build a cycle there must be a path in the graph $G_{\wp}$ from I_S to $I_{S'}$ and vice versa. For every edge $(I_R, I_T) \in E_{\wp}$ the supertype closure is either reduced or the same $st(I_T) \subseteq st(I_R)$, so the path is monotone. In case there is a cycle between I_S and $I_{S'}$ the closure must be the same: $st(S) = st(S')$. This implies the condition from lemma 4.1.6.

□

From lemma 4.1.6 SCCs only contain set interfaces with the same supertype closure. So the solution is to remove all redundant set interfaces and replace the SCCs with a minimal representative, where $I_S \in SCC$ is minimal iff $\forall I_{S_i} \in SCC \backslash \{I_S\} : |S_i| > |S|$. This is the case if S does not contain interfaces in a subtype relationship anymore. For this purpose we define the function *remst*[8]:

Definition 4.1.7. Let $S \subseteq JI_b$ and $remst : \wp(JI_b) \rightarrow \wp(JI_b)$ defined as $remst(S) \stackrel{def}{=} \{s \in S | \neg \exists r \in S : r <: s\}$.

Lemma 4.1.8. *$S_m = remst(S)$ is the minimal representative for the set SCC of S and does not contain interfaces in a subtype relationship.*

Proof: Since the subtype relationship <: on JI_b is monotone S_m must be minimal. The second condition follows directly from the definition 4.1.7.

We can now define the graph $G_{\wp_{red}}$ based on the graph induced by the SCCs and the original subtype relationship on JI_b:

Definition 4.1.9. *Let* $G_{\wp_{red}} =< T_{min}, E_{min} >$ be the minimal solution of $G_\wp$, *with* $T_{min} = \{I_S | S' \subseteq JI_e \wedge S = remst(S')\}$ *and*

$E_{min} = \{(S, S') \in T_{min} \times T_{min} | \forall I' \in S' : \exists I \in S : I <: I'\}$.

Theorem 4.1.10. *$G_{\wp_{red}}$ fulfills the properties of JI_e .*

Proof: Property 1 of definition 4.1.2 follows directly from the definition of E_{min}. By lemma 4.1.6 $G_\wp$ does not contain any non-reflexive cycles. Therefore the graph is anti-symmetric. Since the subtyping relationship <: on JI_b is transitive and reflexive the relation E_{min} is transitive and reflexive directly from its definition. Therefore E_{min} is a partial order.

□

Based on JI_e we determine type m_p with $mapts : \wp(RC) \rightarrow JT$. The function is defined as follows assuming the `rdfs:entailment` of an RDF schema definition:

Definition 4.1.11. The function $mapts : \wp(RC) \rightarrow JT$ maps a set of RDFS classes to a unique type from JT, with $mapts(\{p_{T_1}, ...\}) :=$

$$\begin{cases} rtj_p(p_T) & \{p_{T_1}, ...\} \backslash \{\texttt{rdfs:Resource}, \texttt{rdfs:Literal}\} \subset XDT \\ & \wedge p_T = glb(\{p_{T_1}, ...\})\ (1) \\ mapnvs(\{rtj(p_{T_1}), ...\}) & \{p_{T_1}, ...\} \subset RNVT\ (2) \\ String & \texttt{rdfs:XMLLiteral} \in \{p_{T_1}, ...\}\ (3) \\ String & \{\texttt{rdfs:Literal}\} = \{p_{T_1}, ...\} \backslash \{\texttt{rdfs:Resource}\}\ (4) \\ Object & \{p_T\} = \{\texttt{rdfs:Resource}\}\ (5) \end{cases}$$

with

$$mapnvs(\{\tau_1, ...\}) := \begin{cases} \tau_r & remst(\{\tau_1, ...\}) = \{\tau_r\} \\ I_{\{\tau_1, ...\}} & else \end{cases}$$

Since we assume an rdfs-consistent graph the `rdfs:range` definitions of a property cannot contain disjunctive types, we can distinguish between datatypes, plain literals, XMLLiteral and non-literal classes. Case 1 of *mapts* considers a set of datatypes XDT. Since datatypes are subtypes of `rdfs:Resource` and `rdfs:Literal` they are substracted from the set of p_{T_i} for the subset test. Then the most specific subtype of all p_{T_i} is determined. Section A.2 shows the whole type hierarchy, a partially ordered set of

8 REMove Super Types

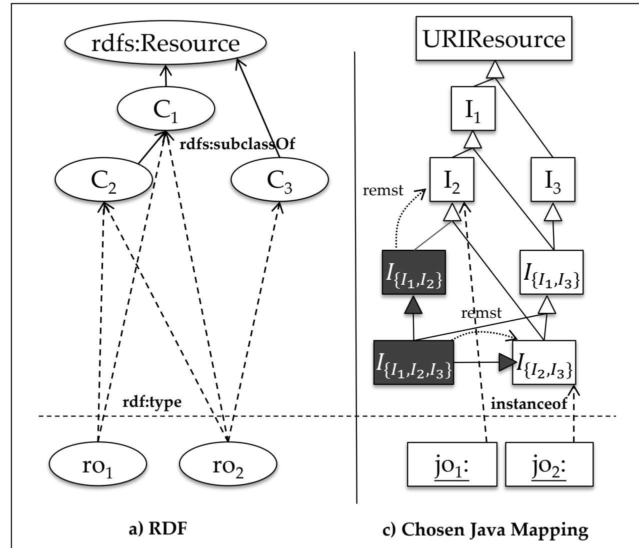


Figure 4.5: Illustration of RDF instance to JI_e mapping

primitive and *derived types*. By determining the greatest-lower bound (*glb*) in the subtype graph, the most-specific subtype p_T can be determined. The function rtj_p returns the corresponding (primitive) Java type.

Case 2 considers the case that all range definitions are non-literal types. The function *mapnvs* facilitates *remst* to determine either a type from JI_b that is a subtype of all $p_{T_1}, ...$ or, in case *remst* returns a set, the corresponding type from JI_e. Case 3 - 5 consider the special cases of XMLLiteral (3), plain literal (4) and `rdfs:Resource` (5).

Reconsidering the example from the beginning of the chapter we can now map the property `nearbyLocation` to Java interfaces. For simplicity reasons, let's assume the set interfaces are mapped to a regular Java interface name by an alphabetic-ordered concatenation of the interface names:

$$mapts(\{\texttt{Location}, \texttt{Nearby}\}) = I_{\{\texttt{Location},\texttt{Nearby}\}} = \texttt{LocationNearby}.$$

And here are the resulting Java interface declarations:

```
interface Contact {
  LocationNearby[] nearbyLocation();
}
interface LocationNearby extends Location,Nearby{}
```

Java Instances

Literals are mapped to Java objects determined by rtj_p. Non-literals with types $T_1, ...$ given by `rdf:type` triples are instantiated as Java objects of type $mapts(\{T_1, ...\})$.

Figure 4.5 illustrates the type mapping with an example. The rdf instances of an instance ro_1 and ro_2 are represented by the Java objects jo_1 and jo_2. The instance ro_1 is an instance of C_1 and C_2. And ro_2 is an instance of C_1, C_2 and C_3. Since C_2 is a subtype of C_1 $remst(\{I_1, I_2\}) = \{I_2\}$ and $remst(\{I_1, I_2, I_3\}) = \{I_1, I_3\}$. Therefore jo_1 is an instance of I_2 and jo_1 is an instance of $I_{\{I_2,I_3\}}$.

The concrete instantiation of the objects depends on the concrete implementation. Possible options are instances of anonymous classes or the generation of one class for every JI_e interface, However, is not relevant for the type system.

```
public interface Statement {
  Object getSubject();
  Object getObject();
  String getPredicate();
}
```

Figure 4.6: RDF reification `Statement` interface

4.1.3 *RDF Reification*

RDF reification is used to make statements on a statement. This can be a dependent statement or extra information about a property, e.g., time stamp or author of a statement. Reification statements are represented as four triples, also called *reification quad* [133, Section 4.3]:

```
:reificationNode rdf:type rdf:Statement.
:reificationNode rdf:subject <some rdfs:Resource>.
:reificationNode rdf:predicate <some rdfs:Resource>.
:reificationNode rdf:object <some rdfs:Resource>.
```

They explicitly denote the subject, predicate and object of a node. The `:reificationNode` can then be referenced by another predicate, which may refer to other properties. We can only provide minimal typing information about reification statements, because we cannot restrict the range of `rdf:object` for special statements. Let's assume we define a statement about products:

```
:productDetails rdfs:subClassOf rdf:Statement.
:timeStamp a rdf:Property;
  rdfs:domain :productDetails;
  rdfs:range  xsd:double.
```

Now we have static information about the properties of `productDetails`. But, there are no further means to restrict the subject or object relationships. The following triple would restrict the range for all statements:

```
rdf:object rdf:range :productStatement
```

For this reason we introduced a general `Statement` interface, see Figure 4.6. For a subclass of `Statement`, like `productDetails`, additional properties, such as `timeStamp`, can be added as regular rdf properties.

By this means details about a product are statically known:

```
product1 -> productDetails -> timeStamp
```

This is just an idiom that can be employed by a schema creator. If the creator does not consider sub-statements in the schema definition, only the minimal types defined in the `Statement` interface is known.

```
Start ::= Condition
Condition ::=
  '!' ( ParenCondition | BooleanExpr )
   | BooleanExpr
   | ParenCondition [ ConditionLogicOp ]
ParenCondition ::=
  '(' [ Condition ] ')'
ConditionLogicOp ::=
  '&&' | '||' Condition
EmptyArray ::=
  '{' '}'
BooleanExpr ::=
  If
   | ExistsCheck
   | Pcd
   | ContextExpression
   | CompOperation
If ::=
  'if' '(' Condition ')'
  'then' '(' Condition ')'  'else' '(' Condition ')'
ContextExpression ::=
  QualifiedName [ ExprOnQualifiedName ]
ExprOnQualifiedName ::=
  ClassPredicate ( PropertyAccess | MapPredicate )*
ClassPredicate ::=
  MapOperator ( Pull | ArrayOp )
PropertyAccess ::=
  '.' Identifier
ArrayOp ::=
  ClassPredicateName '(' [ DeclareVariable ] [ Condition ] ')'
DeclareVariable ::=
  Identifier '|'
ClassPredicateName ::=
  'one' | 'select'
Pull ::=
  Identifier '(' TupleInit ')'
MapOperator ::=
  '-' '>'

ExistsCheck ::=
  'exists' '(' QualifiedName ')'
CompOperation ::=
  TermOrParen Comparator TermOrParen
Comparator ::=
  '='| '<' | '>' | '<=' | '>='
TermOrParen ::=
  ParenTerm | Term
ParenTerm ::= '(' Term ')'
Term ::=
  TupleInit
   | ContextExpression
   | Literal
   | ArithOperation
   | EmptyArray
TermList ::= Term ( ',' Term )*
MapPredicate ::=
  MapOperator Identifier '(' [ TermList ] ')'
ArithOperation ::=
  '(' Term ArithOperator Term ')'
ArithOperator ::=
  '+' | '-'| '*' | '/'
Tuple ::=
  'Tuple' '{' TupleParam ( ',' TupleParam )* '}'
TupleParam ::=
  Identifier ':' QualifiedName [ InitTupleParam ]
InitTupleParam ::=
  '=' Term
TupleInit ::=
  '{' Init ( ',' Init )* '}'
Init ::=
  Identifier '=' Term
QualifiedName ::=
  ( 'this' | Identifier ) ( '.' Identifier )*
Pcd ::=
  Identifier '(' Term ( ',' Term ) * ')'
Literal::=
   Float
   | Int
   | String
```

Figure 4.7: EBNF of OCQL expressions

4.2 OBJECT-ORIENTED LOGIC CONTEXT QUERY LANGUAGE

This chapter introduces the Object-oriented logic Context Query Language (OCQL[9]), a statically typed logic language with meta predicates[10] based on a polymorphic type system with subtyping. The language is the central conceptual contributions of our approach. It is used to query context models in the later presented context management infrastructure (see Chapter 5).

OCQL operates on the JI_e types, which we introduced in Section 4.1. The language was designed to be embeddable into Java language extensions. The type system is a subset of Java's type system. The OCQL is based on *context predicates* that combine the syntax and semantics of the Object-Constraint Language (OCL) [51] with predicate logic. We have chosen OCL as a role model, because it is a popular object-oriented constraint language, provides good means for object graph traversals, and fits well into an object-oriented host language. Figure 4.7 shows the OCQL syntax in EBNF.

We will now explain the language by examples. A formal description is given in Section 4.3 and following, in which we present the type system and the semantics of the language via a mapping to Prolog.

Contexts are queried by predicates over JI_e types (see Definition 4.1.2). Context instances of a class C are either queried one at a time (with C->one), or the whole collection of

9 A more meaningful abbreviation would have been OLCQL, but OCQL is more readable and underlines the mixture of the abbreviations OCL and CQL.
10 We provide the *all-solution* predicates *findall*, *bagof* and *setof* known from Prolog [52].

classes is queried at once (*C*->`select`). In the first case the OCQL expression backtracks[11] over all resolvable variable bindings.

All logic variables used in an expression must be declared and typed. The following expression declares a variable `c` of type `Contact` and binds it to contacts with first name `"Peter"`:

```
(Contact c) : c = Contact->one(firstname = "Peter")
```

The query retrieves all solutions for the expression by backtracking over the solution space. The results are collected in a list of tuples. The `one` construct takes logic expressions as an argument. It operates in the context of the enclosing expression's type and can refer to its properties via their name. The following example extends the previous one and bind the surname of Peter to the variable `name`:

```
(Contact c, String name) :
   c = Contact->one(firstname = "Peter" && surname = name )
```

The query results in the binding:

c	name
http://laj.iai.uni-bonn.de/contacts/1	"Schmidt"
http://laj.iai.uni-bonn.de/contacts/2	"Meyer"

Assuming there are two contacts named Peter Schmidt and Peter Meyer in the context database, the variable `t` is bound to both instances - one after the other. A property of an instance can be bound with notation *expr.propertyName.* In the following example we bind the surname of a contact with first name Peter to the variable `name`:

```
(String name) : name = Contact->one(firstname = "Peter").surname
```

The query results in the binding:

name
"Schmidt"
"Meyer"

If the property has an `rdfs:range` of `rdfs:Container` or collection (`rdf:List`)[12] it will be bound to the array of all elements. Otherwise it will backtrack over all defined properties, e.g., in case `Contact->one().name` over all defined `name` properties.

Property accesses can be arbitrarily nested. The query below binds the `street` property of contacts' addresses:

```
(String street) : street = Contact->one().address.street
```

The expression *Classname*`->select(`*condition*`)` selects a subset of a *Classname*'s instances for which *condition* holds. In the following query the variable `contacts` is bound to `Contact` instances whose `firstname` property equals `"Peter"`.

```
(Contacts[] contacts) :
   contacts = Contact->select(firstname = "Peter")
```

The select expression is always true and returns an empty array if no instance of the set satisfies the condition. Selections of properties on an array are *mapped*. Here, mapping means that for all array elements for which the property exists, the property is added to the returned array. So the following expression binds all addresses of contacts named Peter*:*

11 see Section 2.3.1
12 See Section 2.2.1.

```
(Address[] addresses) :
   addresses = Contact->select(firstname ="Peter").address
```

The condition in a *one* or *select* expression is evaluated in the scope of the queried type. Here, the identifier `this` refers the currently analyzed context instance. Further, the instance of the queried context classes can be named. The example below illustrates the explicit naming and the use of the `this` identifier. In the query instances of the class `Time` are bound to the variable `t`. This has the same semantics as unqualified property names that refer to properties of the type of the enclosing `one` expression, in this example the second reference to the `hour` property:

```
(Time time) : time = Time->one( t | t.hour > 12 && this.hour < 13 )
```

The naming is necessary if there are name clashes between property names and advice parameters or once the instance is used in a nested expression. The following expression that selects today's calendar entries gives an example:

```
(CalendarEntry[] entries) :
   entries = CalendarEntry->select(cal|
      Time->one(cal.day=day && cal.month=month && cal.year=year) )
```

Following OCL, the notation for universally quantified statements is *expr*->`forall`(*c*) and the existential quantification is written as *expr*->`exists`(*c*). These are boolean expressions and the primary logic operations (and, or, not) can be applied to them, e.g., the following conditions bind `c` to a `Contact` object and compares its name property with the string "`Zoe`".

```
(Contact c) : c = Contact->one() && c.name = "Zoe"
```

In case the referenced property does not exist, the expression fails silently. The `exists` predicate tests the existence of a property explicitly:

```
(Contact c) : c = Contact->one() &&
if( c->exists(name) )(
  name = c.name
) else (
  name = "No name given"
)
```

4.2.1 *Arrays and Tuples*

Two kinds of data structures can be used in OCQL: arrays and tuples. Arrays are constructed as Java arrays with curly brackets $\{v_1,...\}$ and a range of array manipulation predicates are provided. The following expression represents an array of integers:

```
( int[] ints ) : ints = {1,2}
```

The type of the expression is inferred from the contained elements. In case a member contains a term with variables the component type must be the same as the declared type of the variable.

Tuples are data types following the OCL syntax for both declaration and construction. A tuple declaration must state the types for all tuple properties: Tuple$\{name_1{:}T_1,...\}$. The tuple instantiation has the following syntax: Tuple$\{name_1[{:}\ T_1]{=}expr_1,...\}$, in which $expr_i$ may contain any expression from the ValueExpr from Figure 4.7. If the type is omitted, it is inferred from the expression. The query below gives an example for a declaration and a tuple instantiation:

```
( Tuple{value1:String, value2:String} t ) :
    t = Tuple{value1="user1", value2="user2"}
```

```
FindallCall ::= 'findall' '('
   [ Ident | 'Tuple' '{' KeyValue ( ',' KeyValue )* '}' ] ','
   BooleanExpr ',' Ident ')'

KeyValue ::= Ident:QualifiedName '=' Term
```

Figure 4.8: Syntax of the `findall` predicate

4.2.2 *Predicates*

Queries in OCQL can be structured with predicates, whose body are context expressions:

```
predicate [ < TP1,...> ] name ( Type pname1, ... ) [ ( Type var1, ... ) ] :
   ContextExpression
```

All parameters and local variables must be declared and typed. The type parameters TP_i are described in Section 4.2.4. The declaration of local variables follows after the parameter declaration and is enclosed by parentheses. Predicates can be recursively defined. As in Prolog, a predicate may consist of several *clauses* that have the same signature and describe alternative cases that form a predicate. The exemplary predicate below prefixes the string `"get"` to the first argument and binds the resulting string to the second argument:

```
predicate toGetter(String name, String getter) :
     concat("get",name,getter);
```

4.2.3 *Built-in Predicates*

The OCQL contains a range of predefined operations[13], for example set operations (union, intersection, difference), instanceof checks, string operations, and the all-solution predicate `findall` known from Prolog (see Section 2.3.3). Figure 4.8 shows the EBNF of the findall predicate. The tuple in the first argument encapsulates solutions of the goal in the second argument. The last argument collects all solutions in an array of tuples of the first argument. The example below illustrates the use of findall. The predicate `names` collects all first and last names bound by predicate `contactNames` to variable `l`:

```
predicate names(Tuple{first:String,last:String}[] l) :
   (String v, String v2) :
   findall( Tuple{first=v,last=v2},
            contactNames(v,v2),l).
```

OCQL adopts two further Prolog built-ins. The predicate `var/1` tests if a variable has not been been bound yet. The predicate `ground/1` tests if the variable is bound to a term that does not contain any variables.

4.2.4 *Generic Types & Mapping Predicates*

Predicates in OCQL allow the use of type parameters with similar typing semantics to type parameters in generic Java methods. For instance, the predefined predicate *sortedBy* sorts the elements from the first argument based on the list of arguments in the second argument and binds the sorted list to the argument `out`:

```
predicate <T> sortedBy(T[] in, double[] values, T[] out)
```

13 A full list of built-in predicates is provided in Appendix A.2.3.

```
Predicate ::= 'predicate' [ TParams ] Ident '(' Formals ')' [ ':' '(' Formals ')' ] ':' Condition ';'
TParams ::= <' TParam ( ',' TParam)* '>'
Formals ::= FormalDec ( ',' FormalDec)*
TParam::= Ident ( 'extends' QualifiedTypeName ( '&' QualifiedTypeName )* )
QualifiedTypeName::= Ident ( '.' Ident )*
ArrayTypeName::= QualifiedTypeName ( '[]' )*
Tuple::= 'Tuple' '{' TupleParam ( ',' TupleParam )* '}'
TupleParam::= TupleParam ':' TypeName'
TypeName::= ( ArrayTypeName | Tuple )
FormalDec ::= QualifiedTypeName Ident
```

Figure 4.9: EBNF of generic predicates

The type arguments for type parameters are inferred. We rely on the type inference algorithm for generic Java methods and therefore a unique data flow between variables must be guaranteed. For example, without unique input or output arguments the type of the type parameter T in `sortedBy`, would not be defined in the following expression[14] (with $Sub < Sup$):

sortedBy(e_1:Sup[], e_2:double[], e_3:Sub[])

The Sections 4.3.5 and 4.3.3 elaborate on the applied mode analysis used to analyze possible dataflows and the inference algorithm adopted from Java.

The type parameters are more restricted compared to Java, since polymorphism is only allowed in predicates, but not on the context classes. Figure 4.9 summarizes the syntax for generic predicates. Type parameters have an optional set of bounds. Variables, whose type declaration contain type parameters, must only be bound to variables of the same type. The following code is only type correct if `t` is already bound when unified with `s`:

```
predicate <T extends C> p(T t,C s) :
   t = s; // <-- t must be bound
```

Again, the mode analysis is facilitated to ensure that these cases are detected.

Using the sortedBy predicate directly is cumbersome. First we have to extract the values from the input array and then call *sortedBy*. To simplify this application we provide the concept of *predicate mapping*. Predicates with 2+n-arguments can be mapped onto the results of a *ContextExpression* (*see* Figure 4.7). The syntax for predicate mapping is

$$expr\text{->}p(e_1,...,e_n)$$

The predicate p must have 2+n arguments, the first argument takes the expression $expr$ as an input, the expressions e_i are evaluated on the $expr$ and the argument a_{n+1} binds the result. This can be used for two kinds of predicate mapping kinds: $N \rightarrow N$ and $N \rightarrow R$:

$N \rightarrow N$ Mapping

This mapping is applied if the type of $expr$ is an array type and its component type is a subtype of the first parameter type τ_0 of $p(\tau_0, \tau_1, ..., \tau_{n+1})$. The subtype relation must also hold for all the types of the arguments e_i evaluated on elements on $expr$ and τ_i. The predicate p is applied to all successive tuples of elements $expr$ and the e_i applied on corresponding elements of $expr$. The bindings of the last argument of p are aggregated into an array and represent the result of the whole mapping expression. For example, consider the predicate `toLower(String, String)` and the calls

```
v = {"U","P"}->toLower()
```

will result in the array `{"u","p"}` and is equivalent to

14 To each sub-expression e_i we appended a colon and the type of the expression.

```
findall(vTmp,(member(m,{"U","P"}),toLower(m,vTmp)),v)
```

with declared variables vTmp and m.

$N \to R$ Mapping

This mapping calls p with $expr$ as its first argument and the results of the expressions $expr.e_i$ are passed to p's arguments 2 to n-1.

The last argument is an output element that represents the result of the whole mapping expression. This mapping can be used for a variety of predicate calls. The most simple is a $1 \to 1$ mapping. Consider the toLower predicate again. This time we map it onto the town property of a Location instance and compare it with the string "cologne":

```
Location(town->toLower() = "cologne")
```

This is equivalent to passing the town property and a result variable name to the toLower predicate:

```
(String name):
   Location(toLower(this.town,name) && name = "cologne")
```

The predicate can also be used for $N \to 1$ aggregations, like summing up an array of numbers or calculating the average. OCQL does not offer a special syntax for array access. It is realized by the predefined generic predicate index/3 and typically applied as a $N \to 1$ mapping:

```
value = arrayInstance->index(1)
```

The index predicate is realized based on the predicate arrayAccess/3[15], another predefined predicate that binds the element at index i of array to out:

```
predicate <T> index(T[] array, int i, T out) :
  arrayAccess(i,array,out);
```

We now come back to the sorting predicate example, which motivated generic predicates. We can use the sortedBy(..) predicate in a $N \to N$ mapping to retrieve the most precise location, by sorting an array of Location instances by their precision and selecting the last element of the sorted array:

```
predicate contactLocations(Location l):
  myLocation = Location->select(l|
               Contact->one(me).locatedAt(self))->
               sortedBy(source.precision)->last();
```

The expression source.precision is evaluated on each Location object and the array of all values is passed to the second argument of sortedBy(..).

If both predicate mappings are possible, e.g., in case the first argument is an unrestricted type parameter, which is a supertype of both arrays and objects, the first mapping is chosen. The reason for this is that the second semantics is simpler to achieve without predicate mapping.

Besides of being syntactic sugar for more complex expressions the static checks for predicate mappings ensure that the first n arguments are in-arguments and the last argument is a ground output argument. On the one hand this underlines that predicate mapping is similar to a function call[16], on the other hand this simplifies type checking of nested calls on generic predicates. Consider the following example, in which the property latitude is selected on the result of the generic predicate sortedBy:

```
Locations->select()->sortedBy(source.precision).latitude
```

To resolve the latitude property the type of sortedBy's third parameter must be known and therefore the first argument must be ground. Section 4.3.2 elaborates this further.

15 For the implementation of arrayAccess/3 see Section A.2.2.
16 except for possible backtracking

4.2.5 *Context History*

By default an OCQL query always queries a snapshot of context data. For some context analysis it is essential to refer to older context data. Section 5.1 will discuss context snapshots in detail, for now we just assume that an arbitrary number of snapshots exist. The OCQL offers a meta-predicate to switch the evaluation to an older context snapshots:

```
(long ts): history(ts,<condition>)
```

The history predicate binds for all snapshots the associated timestamp ts and backtracks over all snapshots and conditions. The current snapshot is always the latest and can be retrieved via the built-in predicate `currentSnapshot/1`.

Let's assume we would like to get the current compass orientation of a device. Since users cannot keep the device completely steady in their hands, the compass sensor read is typically fluctuating. To avoid these minimal movements often low-pass filters are applied, which smooths sensor read changes, by taking previous measurements into account:

```
predicate lowpass(float[] old,float current, float alpha, float result):
  if( old->size(0) ) then (
    result = current
  ) else (
    result = current + (alpha * (old->index(0) - current)) &&
    lowpass(old->rest(), result, alpha*alpha, result)
  );
predicate smoothCompass(float degree): (long ts, Compass[] cs):
  findall(c, history(ts,c = Compass->one()),cs) &&
  lowpass(cs->rest().degree, cs->index(0).degree, 0.25, degree);
```

The `smoothCompass` predicate retrieves the compass instances of all existing snapshots, and passes all old entries to the first argument of `lowpass` and the current `degree` to the second argument. The third argument is a constant factor by which the older values are taken into account.

The predicate `lowpass` recurses over the array of old values and adds the difference between the old and the current value, multiplied by $alpha^i$ to the current value. The index i stands for the index in the array in the i-th recursion.

4.2.6 *Querying Context Sources*

Querying data on a (mobile) client has a major restriction. We cannot hold and query large sets of data on the client side. To access such information or parts of it[17] we introduce *query context sources* (QCS). These are context sources with predefined queries to external systems. Examples for such sensors are social network APIs providing access to their databases via Web services. They take a set of parameters and return a structured set of data. QCSs are queried with the following expression:

var = *SensorTypeName*->*methodName*({$paramName_1 = value_1$,...})

The definition of the QCSs themselves will be introduced in Section 5.2.

The predicate `lastfm` shows how this can be applied to a concrete Web Service, the tasteometer service from last.fm. We assume that the Web Service was wrapped by the Java class `Lastfm`. The method takes as an argument a tuple with two last.fm usernames (`name1` and `name2`) and returns a set of artists that both users like. This predicate will be later used in the realization of the last.fm example.

17 e.g., a view onto a relational database

```
predicate lastfm(Artist[] artists,
         String myName, String otherName) :
   artists = Lastfm->tasteometer(
     {name1=myName,name2=otherName});
```

4.2.7 *Type Checks & Casts*

OCQL offers built-in predicates for type-tests that can be called by predicate mapping:

***expr*->isTypeOf(*Typename*)** is a boolean expression that checks that an expression is of a type *Typename.*

***expr*->asType(*Typename*)** is a type cast to subtype *Typename*. It fails if the expression is not of type *Typename*.

***exprs*->typeSelect(*Typename*)** operates on an array of expressions. It selects and returns all elements of type *Typename*.

Let's assume the two classes `Contact` and `Colleague` exist and `Colleague` is a subtype of `Contact`. A cast of `Contact` to `Colleague` is evaluated as follows:

```
predicate getContact(Contact c, Colleague cl):
   cl= c->asType(Colleague);
```

The predicate `asType` can also be applied on array types. Expressions in OCQL do not have a fixed runtime component type, as in Java. So asType tests all elements in the array if the cast is legal:

```
predicate getContact(Contact[] cs, Colleague[] cls):
   cls= cs->asType(Colleague[]);
```

4.3 TYPE CHECKING

Typing of OCQL is based on an adaptation of Java's type system. In the first place, the adaptation is needed, since the data flow in logic programs is not easily recognizable from the syntax of the language. In particular, the data flow direction is not a property of a predicate, but can differ depending on each of its call sites. However, mode information can be facilitated to analyze the direction of dataflow between logic variables[18], and a suitable mode analysis has been described in Chapter 3.

Local variables in OCQL must be typed, as it is common in the Java type system.[19] Arithmetic expressions, comparisons and string operations in SLD resolution fail if unbound variables are used in the wrong arguments. We tackle both problems, type checking and groundness checks, by a three-step analysis.

The next Section repeats the syntax for modes and explains the general idea of our approach. In the follow-up section we present OCQL's type language and the three-step analysis.

4.3.1 *Overview*

The central idea of this approach is to reduce all typing problems to the unification of two typed terms. In case they have the same type, the unification is valid for any "dataflow"

18 as we discussed in Section 2.4.2

19 OCQL's type system is very reduced compared to typed terms in logic programming. Type inference for unary type constructors with subtyping (inclusion polymorphism) and parametric polymorphism becomes decidable in this case [85]. Since it is common that all variables in Java are typed, we think type inference for local variables is not essential and consider an application to OCQL's local variables as possible future work.

between the terms. In case their types differ, a mode analysis is applied to test if the unification is valid, otherwise no further handling is need. Especially in the presence of type parameters, a number of considerations have to be made. This section only gives a first intuition, which problems can already occur in the non-generic case.

In the following, we look at the unification of the two variables $v_1 = v_2$, where either the type of v_1 is a subtype of type of v_2 or vice versa. We assume, the mode of the variables before and after the unification is already known. The analysis (see Chapter 3) uses the following mode kinds:

f for a variable,

g for ground term, and

o if it is neither a variable nor ground term.

The analysis is a safe approximation of SLD resolution, in the worst case resulting in the set of all modes *{f,g,o}* instead of a single possible mode {*f*} or {*g*}. Variables with mode *o* and *f* must not be passed to an argument expecting a ground variable. In the following a *ground* variable has mode *{g}* and a *bound* variable might have mode *{o}*, *{g}*, or {g,o}.

For all unifications we first check if the declared types of the unified variables[20] $v_1 : \tau_1$ and $v_2 : \tau_2$ are in a subtype relationship: $\tau_1 <: \tau_2$ or $\tau_1 :> \tau_2$, and signal a type error if they are not. In case they are in a *proper subtype relation* $\tau_1 < \tau_2$, where $<$ stands for the non-reflexive subtype relationship, we check for potentially illegal flows with the inferred mode annotations. The typing rule below illustrates the principle[21]:

$$\frac{\tau_1 < \tau_2, mo(v_1, pp) = \{g\}}{(v_1 : \tau_1 = v_2 : \tau_2)_{pp}}$$

The function $mo(V, pp) \rightarrow \{f, g, o\}$ represents the mode of the variable V at the program point pp. The typing rule states that the unification of two typed variables $v_1 : \tau_1$ and $v_2 : \tau_2$ with : $\tau_1 <: \tau_2$ is legal if the variable v_1 is ground at pp. The reverse case is analogous.

We demonstrate the rule with the OCQL query below, where $pp_{1,2}$ is the program point before the unification of `o` and `s`:

```
(String[] strs, String s, Object[] objs) : strs = {"str"} && (objs = strs)pp1,2
```

Applied to the unification typing rule we get the following:

$$\frac{\texttt{String[]} < \texttt{Object[]}, mo(\texttt{s}, pp_{1,2}) = \{g\}}{(\texttt{s} : \texttt{String[]} = \texttt{o} : \texttt{Object[]})_{pp_{1,2}}}$$

If the mode of v_1 is not ground, an illegal binding of a subtype to its supertype might occur in this unification. For example, in case that $mo(v_1, pp) = \{o\} \wedge mo(v_2, pp) = \{o\}$, the unification of $v_1 = v_2$ might assign a subtype to a member to a partially ground array, e.g., in the following query:

```
(String[] strs, String s, Object[] objs, Object o) :
    strs = {"str",s} && objs = {o,1} && (objs = strs)pp1,2
```

We illustrate in an example how the analysis is applied. Each program point in the `friends` predicate is annotated with modes for each declared variable v. We indicate the inferred mode information with [*v/m*] for every variable in a clause, before and after each program point[22], where the mode m is $m \subseteq \{f, g, o\}$. Let's assume the two types `Contact`

20 constants are processed analogously
21 The typing rule only demonstrates the general idea and is not part OCQL's type system.
22 A program point is a location in a logic program before or after a literal, see Section 3.1.1 for details.

and `Colleague` exist and are in the subtype relation supertype `Colleague` < `Contact`. We want to calculate the closure of all friends of a contact with a maximum search depth implemented in the predicate `friendsClosure`[23]:

```
01 predicate friendsClosure(Contact p,Contact p2,int depth,int max): (Contact temp):
02 // [p/{o}, p2/{f}]
03   p = p2
04 // [p/{o}, p2/{o}]
05   ||
06 // [p/{o}, p2/{f}, temp/{f}, depth/{g}, max/{f}]
07   (  depth < max &&
08      p->hasFriend(temp) &&
09 // [p/{g}, p2/{g}, temp/{g}]
10      friendsClosure(temp, p2, depth + 1)
11 // [p/{g}, p2/{g}, temp/{g}]
12   );
13
14 predicate knownColleague(Colleague c, Colleague c2) :
15   .. c=Colleague->one() &&
16 // [c/{g}, c2/{f}, max/{f}]
17   friendsClosure(c,c2,0,max); // error
18 // [c/{g}, c2/{g}, max/{f}]
```

Let's assume the predicate `friendsClosure` is called with the mode (*g,f,g,f*). The algorithm will detect two errors. In line 6 the variable `max` is not bound and has been passed to the expression depth < max. In line 16 the unbound variable `c2` is passed to the second parameter `p2` of `friends`, which has the declared supertype `Contact`.

OCQL offers type casts of value expressions (e.g., location->asType(GPSLocation)) and type checks (location->isTypeOf(GPSLocation)). These expressions are only allowed to be called with ground variables. The cast is treated as a boolean expression that either succeeds or fails. No runtime errors or exceptions are raised if the check fails. With the help of a cast we can rewrite the code to make it type safe:

```
01 predicate knownColleague(Colleague c, Colleague c2) : (Contact tmp) :
02   .. c=Colleague->one() &&
03 // [c/{g}, tmp/{f}]
04   friendsClosure(c,tmp,0,3) &&
05 // [c/{g}, c2/{f}, tmp/{g}]
06    c2 = tmp->asType(Colleague); // no compiler error
07 // [c/{g}, c2/{g}, tmp/{g}]
```

In the following, we present the complete type checking algorithm, which is structured into three major steps. At the beginning we apply an initial type checking (Section 4.3.2) for all expressions where the mode information is not relevant. In this step all variables and properties are resolved in their corresponding context. Some further assumptions are made about modes in this phase, which have to be confirmed once the mode inference has been carried out. Section 4.3.4 covers the transformation to Prolog and finally Section 4.3.5 describes the application of the mode analysis as a final type checking step. Furthermore, the assumptions about modes from the first step are validated and the modes are used to check valid inputs for arithmetic expressions, string operations, etc.

4.3.2 *Pre-Mode Analysis Type Checking*

This Section describes OCQL's type system. Section 2.4 sums up the different kinds of type systems for logic programming and their motivation. The purpose of our type system is to statically guarantee the well-typedness of a goal. This means that in the

23 For brevity, we ignore cycle checks in this example.

untyped resolution of a goal each variable is either unbound or (partially) bound to a value of the declared type or subtype. So types can be ignored at runtime, in contrast to typed logic programming approaches such as Prova [123] and Protos-L Beierle [28].

The type system contains three kinds of types: elementary, array and tuple types. The elementary types in OCQL are the interface types defined in JI_e (see Definition 4.1.2) and the data types $JPTW$ in the co-domain of rtj_p, see Definition 4.1.1. In OCQL the methods of types represent the properties of the corresponding RDFS class as defined in the RDF mapping Chapter 4.1. Array types can have elementary or tuples as their *component types*. Tuple arguments are considered to be alphabetically sorted by their names. This will be relevant in the later translation process to Prolog. The language of types τ is defined as follows:

T ::=

C	an elementary type
$\mid X$	a type parameter
$\mid Tuple\{\overline{a} : \overline{\tau}\}$	tuple with named arguments and their types

τ ::=

T	
$\mid a(T, Dim)$	array with *component type* `T` and dimension `Dim`
$\mid \tau(\tau_1)$	instantiation of the parametrized type τ with type τ_1
$\mid \tau_0 \rightarrow \tau$	typing of a method declared in type τ_0 with the return type τ, a rdf property p of type τ is represented by the method $\tau\ p()$
$\mid \tau_0 \times \ldots \times \tau_n$	predicate signature

One further restriction applies for typle types - they must not contain type variables. We made this restriction to simplify the description of the type checking algorithm, but consider an extension to tuples straight forward.

By this restriction, only array types may contain type parameters. The type parameters of the predicates are not explicitly represented in the signature, because an explicit representation in the predicate signature will not be necessary.

We define the two helper functions *comp* and *arrayOf*, which decrease and increase the dimension of an array:

$$comp(\tau) \triangleq \begin{cases} \tau & if\ \tau \text{ is not an array} \\ \tau' & if\ \tau = a(\tau', 1) \\ a(\tau', Dim - 1) & otherwise \end{cases}$$

$$arrayOf(\tau) \triangleq \begin{cases} a(\tau, 1) & if\ \tau \neq a(\tau', Dim) \\ \tau = a(\tau', Dim + 1) & otherwise \end{cases}$$

We write typing rules in the natural deduction style [26], where a rule consists of an antecedent and a conclusion, separated by a line, based on the production rules of the context-free grammar given in Figure 4.7.

Properties given in the antecedent must be proven to conclude the conclusion. We assume that every type from JI_e is already resolved, based on the RDF Schema imports defined in the OCQL queries's preamble.

The set of class types are defined by the imported RDF schemas. A context $\mathcal{E}$ maps variable names v to their type τ. $\mathcal{E}(v) = \tau$ means v has type τ in $\mathcal{E}$. $\mathcal{E}$ also contains the subtype relation between types, which are derived from the type hierarchy of the imported context classes.

We define a type environment Δ with a finite mapping from type variables to bounds, predicate types $predicateName : \tau_0 \times \ldots \times \tau_n$, property method typing $methodName : \tau_0 \rightarrow \tau$, and subtype relationships. We write $\tau <: \tau'$ to state that τ is a subtype of τ'. The

Bound of type:

$$bound_\Delta(X) = \Delta(X)$$
$$bound_\Delta(a(T, Dim)) = a(bound_\Delta(T), Dim)$$
$$bound_\Delta(C) = C$$
$$bound_\Delta(Tuple\{\overline{a} : \overline{\tau}\}) = Tuple\{\overline{a} : \overline{\tau}\}$$

$$\frac{\Delta \vdash \tau <: \zeta, \Delta \vdash \zeta <: \tau'}{\Delta \vdash \tau <: \tau'} \quad (trans)$$

$$\frac{\Delta \vdash \tau <: \tau'}{\Delta \vdash a(\tau, Dim) <: a(\tau', Dim)} \quad (array)$$

$$\Delta \vdash \bot <: \tau \quad (bottom)$$

$$\Delta \vdash \{\} : a(\bot, 1) \quad (emptyarray)$$

$$\frac{\Delta \vdash e : a(\bot, n)}{\Delta \vdash \{e\} : a(\bot, n+1)} \quad (nestedea)$$

$$\frac{\Delta \vdash \tau <: \tau'}{\Delta \vdash \tau[] <: \tau'[]} \quad (arraycov)$$

Figure 4.10: Type conformance

term $\Delta \vdash \tau \lesseqgtr \tau'$ denotes either $\Delta \vdash \tau <: \tau'$ or $\Delta \vdash \tau :> \tau'$. We call two types *compatible* in this case.

Figure 4.10 lists the subtyping rules for component types and arrays and introduces the bottom type $\bot$ that is a subtype of all types. The $\bot$ type will be used to type check empty arrays. The rules *emptyarray* and *nestedea* recursively define the type of nested empty arrays. Array types are covariant, as noted by rule *arraycov*.

The function $bound_\Delta$ retrieves the bound for τ. For type τ without type variables it returns τ, otherwise it is recursively called on all component types. Since tuple types must not contain type variables, $bound_\Delta$ is the identity function for tuples.

In a pre-processing step all bounds of type variables $X\, extends\, \overline{\tau}$ are replaced by $mapts(\overline{\tau})$, see Definition 4.1.11. This does not change the semantics of the predicates, since

- all subtypes of $\overline{\tau}$ are subtypes of $mapts(\overline{\tau})$ by definition,
- $mapts$ ensures that all occurrences of $\overline{\tau}$ are replaced by $mapts(\overline{\tau})$, and
- all supertypes of $\overline{\tau}$ are supertypes of $mapts(\overline{\tau})$ by definition of JI_e.

This replacement simplifies the typing rules, since only one bound has to be considered instead of n.

Figure 4.11 lists the typing rules for the logic operators and first order predicates. The rules are defined on non-terminals of the EBNF[24]; the environment $\mathcal{E}$ is initialized with the parameters and local variables of the enclosing predicate.

The first four rules describe the straight-forward typing of boolean operators. The *var-access* rule states that if $Identifier$ has type τ in $\mathcal{E}$ then the expression $Identifier$ has type τ. The *emptyarray* rule assigns the array of bottom type ($\bot[]$) to the empty array. This ensures that it can be assigned or compared to expressions of arbitrary arrays types with dimension 1. The rule *constant* states that the constant $c \in Const$ has its pre-assigned primitive type inferred from the EBNF syntax during parsing.

24 see Figure 4.7

The rule *prop-access* covers property accesses on a non-array expression, and *prop-access-array* on array types. A property access on an array results in an array of the same dimension. Nested arrays therefore result in nested property arrays. The predicate call *pred-call-pre* checks for potential applicability of a predicate disregarding the yet unknown dataflow direction.

The rules *var-access* and *constant* cover unqualified identifier resolution and constant typing. The rule *prop-access-this* covers an unqualified property access equivalent to this.*propname*. Since identifiers in $\mathcal{E}$ take precedence over implicit property accesses, $\mathcal{E} \not\vdash Identifier : \tau$ ensures that no *Identifier* is already defined in the current scope $\mathcal{E}$. The rule *prop-access* considers the variable access on an arbitrary *ContextExpression*.

The rule *arith-plus* is a placeholder for all arithmetic expressions. The typing rules follow *binary numeric promotion* [88, 5.6.2], meaning that a widening conversion is performed in case the operands have different types, e.g., 1 + 1.1 is typed Double.

The *unify-pre* rule states that the unification of two expressions must be type compatible for at least one dataflow direction. The rule takes the $bound_\Delta$ of τ and τ' since they may contain type variables and the subtype relationship is only defined on non-generic types. The *unif-pre* rule is a pre-step and the final type checking step for unification is postponed until information on the direction of the data flow at that unification has been inferred.

Figure 4.12 lists the typing rules for class predicates and meta predicates. Newly defined identifiers, e.g., defined in *select* or *one* expressions, hide existing identifiers in a context. Therefore we define the operator $\cup^\setminus$ on the environment $\mathcal{E}$ that subtracts existing identifier associations which are defined in the right set from the left set and then takes the union of both sets, e.g., $\{a : \tau, b : \tau'\} \cup^\setminus \{a : \tau''\} = \{a : \tau'', b : \tau'\}$.

The *select* rule assigns an type $C[]$ to the select expression and updates the environment in which *Condition* is type checked, with the variables v and `this` with type C. The *one* case is analogous. The *findall* rule considers findall calls with a variable as a first parameter. The third argument collects all results from findall in v_{bag} which declared type must be compatible with v's declared type. This is also a pre-check. Possible dataflows between v and v_{bag} are not defined at this point. The variable v_{bag} might have been bound beforehand and v might not be bound in *Condition* at all. The rule *findall-t* considers a tuple type for v_{bag}. The argument names b_i must all be declared in the type v_{bag} and their types τ_{a_j} must be *type compatible* with the corresponding type τ_{v_i}.

Figure 4.13 considers *predicate mapping*, as described in Section 4.2.4 and 4.2.4. In contrast to the *pred-call-pre* rule the mode of the first argument is restricted here. With this premise, we are able to infer the type of the predicate based on the type inference algorithm elaborated in 4.3.3. For now, we only informally define the function $infer(\overline{X}, \overline{B}, \overline{\zeta}, \overline{\tau}, \tau) \rightarrow \zeta$. The function infers the output type ζ for arguments $\overline{\zeta}$ for a predicate p($\overline{\tau} \times \tau$), with type variables $\overline{X}$ with associated bounds $\overline{B}$. The notation $\overline{\tau}$ stands for a list of types $\tau_1, ..., \tau_n$ and $\overline{\tau} \times \tau$ for $\tau_1 \times ... \times \tau_n \times \tau$.

The rule *map-N-N* considers a predicate *pred*, which is mapped onto the component types of the array bound by *CxtExpr*, an abbreviation for the EBNF non-terminal *ContextExpression*. The premise checks that the type of *CtxExpr* is an array type ζ and the type of the first parameter of the mapped predicate *pred* is a supertype of the component type of ζ. The arguments $\overline{e}$ of the mapping call are evaluated in the context of the component type ζ_{comp}, therefore the identifier `this` with type ζ_{comp} is added to the environment in which $\overline{e}$ are type checked. The types of arguments must be subtypes of the predicates second to n-1 parameter and finally the inference must succeed with type η. The type of the whole expression η' is $\eta[]$. Consider the following query, which sorts contacts by their surname:

```
(Contact[] cs, Contact[] sorted):
  cs = Contact->select() &&
  sorted = cs->sort(surname);
```

The query calls the following generic sort predicate:

$$\frac{\mathcal{E} \vdash e : Boolean, \mathcal{E} \vdash e' : Boolean}{\mathcal{E} \vdash e \,\&\&\, e' : Boolean} \; (and)$$

$$\frac{\mathcal{E} \vdash e : Boolean, \mathcal{E} \vdash e' : Boolean}{\mathcal{E} \vdash e || e' : Boolean} \; (or)$$

$$\frac{\mathcal{E} \vdash e : Boolean, \mathcal{E} \vdash e' : Boolean,, \mathcal{E} \vdash e'' : Boolean}{\mathcal{E} \vdash \texttt{if}(e)\texttt{then}(e')\texttt{else}(e'') : Boolean} \; (if)$$

$$\frac{\mathcal{E} \vdash e : Boolean}{\mathcal{E} \vdash !e : Boolean} \; (not)$$

$$\mathcal{E} \vdash Identifier : \tau \quad (var\text{-}access)$$

$$\mathcal{E} \cup \{Literal : \tau\} \vdash Literal : \tau \quad (constant)$$

$$\frac{\mathcal{E} \not\vdash Identifier : \tau, \mathcal{E} \vdash \texttt{this} : \tau_0, \Delta \vdash Identifier : \tau_0 \to \tau}{\mathcal{E} \vdash Identifier : \tau} \; (prop\text{-}access\text{-}this)$$

$$\frac{\mathcal{E} \vdash e : \tau_0, \tau_0 \neq a(\zeta, Dim), \Delta \vdash Identifier : \tau_0 \to \tau}{\mathcal{E} \vdash e.Identifier : \tau} \; (prop\text{-}access)$$

$$\frac{\mathcal{E} \vdash e : \tau_0, \tau_0 = a(\tau', Dim), \Delta \vdash prop : \tau' \to \tau}{\mathcal{E} \vdash e.prop : a(\tau', Dim)} \; (prop\text{-}access\text{-}array)$$

$$\frac{\mathcal{E} \vdash e : \tau, \mathcal{E} \vdash e' : \tau', \tau <: Double, \tau' <: Double, lub(\tau, \tau') = \zeta}{\mathcal{E} \vdash (e : \tau + e' : \tau') : \zeta} \; (arith\text{-}plus^*)$$

$$\frac{\mathcal{E} \vdash e : \tau, \mathcal{E} \vdash e' : \tau', bound_\Delta(\tau) \lesseqqgtr bound_\Delta(\tau')}{\mathcal{E} \vdash e{=}e' : Boolean} \; (unify\text{-}pre)$$

$$\frac{\Delta \vdash pred : (\overline{\tau}), \mathcal{E} \vdash \overline{e} : \overline{\zeta}, bound_\Delta(\overline{\zeta}) \lesseqqgtr bound_\Delta(\overline{\tau})}{\mathcal{E} \vdash pred(\overline{e}) : Boolean} \; (pred\text{-}call\text{-}pre)$$

*A representative for all arithmetic operations. See Section A.2.3 for all arithmetic operations.

Figure 4.11: Typing rules - core

$$\frac{\mathcal{E} \cup^{\setminus} \{v : C, \texttt{this} : C\} \vdash Condition : Boolean}{\mathcal{E} \vdash C\texttt{->select}(v|Condition) : a(C, 1)} (select^*)$$

$$\frac{\mathcal{E} \cup^{\setminus} \{v : C, \texttt{this} : C\} \vdash Condition : Boolean}{\mathcal{E} \vdash C\texttt{->one}(v|Condition) : C} (one^*)$$

$$\frac{\mathcal{E}(v) = \tau, \mathcal{E} \vdash Condition : bool, \mathcal{E} \vdash v_{bag} : \tau', bound_{\Delta}(\tau[]) \lesseqgtr bound_{\Delta}(\tau')}{\mathcal{E} \vdash \texttt{findall}(v, Condition, v_{bag}) : Boolean} (findall)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash Condition : Boolean, \mathcal{E} \vdash v_{bag} : Tuple\{a_1 : \tau_{a_1}, ...\}, \mathcal{E} \vdash v_1 : \tau_{v_1}, ... \\ \{b_1, ...\} \subseteq \{a_1, ...\}, \forall b_i, a_j : b_i = a_j \Rightarrow bound_{\Delta}(\tau_{v_i}) \lesseqgtr bound_{\Delta}(\tau_{a_j})\end{array}}{\mathcal{E} \vdash \texttt{findall}(\{b_1 = v_1, ..., b_n = v_n\}, Condition, v_{bag}) : Boolean} (findall\text{-}t)$$

*The variants without the variable declaration v only differ by the lack of v in the premise and conclusion of the rule.

Figure 4.12: Typing rules for `select`, `one`, and `findall`

$$\frac{\begin{array}{c}\Delta \vdash pred : (\tau \times \overline{\tau} \times \tau'), \mathcal{E} \vdash CxtExpr : \zeta, \zeta = a(\zeta'', Dim), comp(\zeta) = \zeta_{comp}, \\ bound_{\Delta}(\zeta_{comp}) <: bound_{\Delta}(\tau), \mathcal{E} \cup^{\setminus} \{\texttt{this} : \zeta_{comp}\} \vdash \overline{e} : \overline{\zeta}, \\ bound_{\Delta}(\overline{\zeta}) <: bound_{\Delta}(\overline{\tau}), infer(\zeta_{comp}\overline{\zeta}, \tau\overline{\tau}, \eta) = \eta, \eta' = arrayOf(\eta)\end{array}}{\mathcal{E} \vdash CxtExpr- >pred(\overline{e}) : \eta'} (map\text{-}N\text{-}N)$$

$$\frac{\begin{array}{c}\Delta \vdash pred : (\tau \times \overline{\tau} \times \tau'), \mathcal{E} \vdash CxtExpr : \zeta, \\ \zeta = a(\zeta'', Dim) \wedge bound_{\Delta}(\tau) \neq Object \wedge bound_{\Delta}(\overline{\tau}) \neq \overline{Object}, \\ comp(\zeta) = \zeta', bound_{\Delta}(\zeta') <: bound_{\Delta}(\tau), \mathcal{E} \cup^{\setminus} \{\texttt{this} : \zeta'\} \vdash \overline{e} : \overline{\zeta}, \\ bound_{\Delta}(\overline{\zeta}) <: bound_{\Delta}(\overline{\tau}), infer(\zeta \times \overline{\zeta}, \tau \times \overline{\tau}, \tau') = \eta\end{array}}{\mathcal{E} \vdash CxtExpr- >pred(\overline{e}) : \eta} (map\text{-}N\text{-}R)$$

Figure 4.13: Type checking rules - first pass

```
<T> sort(T in, String[] compareValue, T out)
```

The query is translated into a semantically equivalent variant without predicate mappings:

```
(Contact[] cs, Contact[] sorted, String[] sns):
  cs = Contact->select() &&
  sns = cs.surname &&
  sort(cs,sns,sorted);
```

The only difference is that the mode {g} of cs and sns is already checked in the pre-mode analysis type checking phase. The predicate sort is called with the modes sort({g},{g},{f}). The typing rule *map-N-N* is applied and *infer*({X},{Object},{`Contact`[],`String`[]},{`T`[],String[]},`T`[]) evaluates to `Contact`[].

The rule *map-N-R* considers the simpler case where *pred* is directly called on $CxtExpr$ and an array of expressions on $CtxExpr$. The second row rejects the case where the argument 1 to n-1 of *pred* all have the bound Object. Otherwise both premises of *map-N-N* and *map-N-R* would overlap in this case.

4.3.3 *Type Inference for Generic Predicates*

In case the modes for all variables in a predicate call are known, we can infer the type of a predicate's output parameters.

Definition 4.3.1. The function $infer(\overline{X}, \overline{B}, \overline{\zeta}, \overline{\tau}, \tau) \rightarrow \zeta$ infers the output type ζ for the argument types $\overline{\zeta}$, the predicate input parameters $\overline{\tau}$ and the output parameter type τ. The first argument is a list of type variables $\overline{X}$, the second argument is the list of bounds $\overline{B}$ associated with $\overline{X}$.

The inference is independently applied for each output parameter. It is a reduced form of the inference algorithm of [88, 15.12.2.7], "Inferring type arguments based on actual arguments". In our algorithm, we do not need to consider primitive type boxing, wildcards, type parameters in type parameter bounds. The only parametric types are arrays. Type parameters used in "out" parameters must be present in "in" parameters, because otherwise only the empty array could be assigned the variable as in the following example:

```
<T> p(T[] t):
  t = {};
```

The empty array is the only expression containing bottom type $\perp$ that is compatible with arbitrary concrete array types. Since this scenario is of very limited use, OCQL does not support type inference from inner-predicate calls.

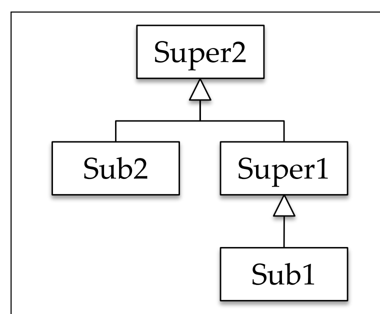
Figure 4.14 cites the Java inference algorithm from [88, 15.12.2.7], where all parts related to primitive type boxing, wildcards, type parameters in type parameter bounds have been removed.

Since no generic types can be defined in OCQL no capture conversion [88, 5.1.10] needs to be applied in the determination of the predicate return type as it is necessary for the general Java case [88, 15.12.2.6].

In a final step we apply $remst(\{W_1, ..., W_r\}) = \text{IT}_j$ to determine the greatest lower bound for the W_i to retrieve just one type instead of several bounds.

Example

This paragraph applies the type inference algorithm exemplarily. Let's assume the following type hierarchy is given:



Further, the following predicate definitions are given:

```
01 <T> p(T[] p1_in, T p2_in, T[] out).
02
03 <T2 extends Sub2> q(Sub1[] p1, T2 p2, Super2 p3) :
04   // [p1/{g},p2/{g},p3/{f}]
05   p(p1, p2, p3);
```

The notational conventions used in this section are

- Type expressions : A, F, U, V and W, where A denotes the type of an actual parameter and F is only used to denote formal parameters.
- Type parameters are represented by T

Inference begins with a set of initial constraints of the form A <: F, A = F, or A :> F, where U <: V indicates that type U is convertible to type V by widening reference conversion [88, 5.1.5], and U :> V indicates that type V is convertible to type U by widening reference conversion.

- If F does not involve a type parameter T_j then no constraint is implied on T_j.
- Otherwise, F involves a type parameter T_j.
  - if the constraint has the form A <: F
    - if F = T_j, then the constraint T_j :> A is implied.
    - If F = U[], where the type U involves T_j, then if A is an array type V[], or a type variable with an upper bound that is an array type V[], where V is a reference type, this algorithm is applied recursively to the constraint V <: U.
  - Otherwise, if the constraint has the form A = F
    - If F = T_j, then the constraint T_j = A is implied.
    - If F = U[] where the type U involves T_j, then if A is an array type V[], or a type variable with an upper bound that is an array type V[], where V is a reference type, this algorithm is applied recursively to the constraint V = U.

  Next, for each type variable T_j, $1 \leqslant j \leqslant n$, the implied equality constraints are resolved as follows:

For each implied equality constraint T_j = U or U = T_j:

- If U is not one of the type parameters of the method, then U is the type inferred for T_j. Then all remaining constraints involving T_j are rewritten such that T_j is replaced with U. There are necessarily no further equality constraints involving T_j, and processing continues with the next type parameter, if any.
- Otherwise, if U is T_j, then this constraint carries no information and may be discarded.
- Otherwise, the constraint is of the form $T_j = T_k$ for $k \neq j$. Then all constraints involving T_j are rewritten such that is replaced with T_k, and processing continues with the next type variable.

Then, for each remaining type variable T_j, the constraints T_j :> U are considered. Given that these constraints are T_j :> U_1 ... T_j :> U_k, the type of T_j is inferred as lub(U_1, ..., U_k), computed as follows:
For a type U, we write *ST*(U) for the *set of supertypes* of U, and define the *erased supertype* set of U, *EST*(U) = { V | W in *ST*(U) and V = *erasure*(W) }, where the *erasure* of

- an array type T[] is erasure(T)[].
- a type parameter is the erasure of its bound.
- every other type is the type itself.

The *erased candidate* set for type parameter T_j, *EC*, is the intersection of all the sets *EST*(U) for each U in U_1 .. U_k. The *minimal erased candidate* set for T_j is *MEC* = { V | V in *EC*, and for all W≠V in *EC*, it is not the case that W<: V}.
Then the inferred type for T_j is lub(U_1, ..., U_k) = W_1 & ... & W_r where W_i, 1⩽i⩽r, are the elements of *MEC*.

Figure 4.14: Our simplification of the Java type parameter inference algorithm [88, 15.12.2.7]

In line 5 the predicate `p` is called with the modes {*g*}, {*g*}, {*f*}. Now we infer the type of `T` of `p`'s `out` parameter for this call.

From the parameters of `p` the constraints `T` :> `Sub1` and `T` :> `T2` are inferred, resulting in the erased super types as defined in Figure 4.14:

EST(`Sub1`) `={Sub1,Super1,Super2,Object} and`
EST(`T2`) `={Sub2,Super2,Object}.`

As the next step we calculate *EC*, the intersection of all *EST* sets:

EC(`{{Sub1,Super1,Super2,Object},{Sub2,Super2,Object}} = {Super2,Object}`

Now we evaluate *MEC* on the result of *EC*, which removes all supertypes of types in *EC*:

MEC(`{Super2,Object}) = {Super2}`

Finally, we apply the *remst* function to the result of *MEC*, which just preserves the type in this case:

remst(`{Super2}) = Super2`

So, the resulting type for the out argument is `Super2`[]. In case we call the predicate with the arguments `p`(`T2`[], `T2`, `Super2`) the inferred type for `T` is `T2`.

4.3.4 *Translation To Prolog*

This section defines the semantics of the language by defining a translation[25] to a subset of Prolog. The predicates of OCQL represent logic expressions that naturally map to Prolog predicates and facts. In this step all Prolog variables are annotated with their associated types. They will be used in final type checking phase (in Section 4.3.5), but this type information is solely used for static checks and is removed afterwards.

OCQL's context model is based on RDF Schema. RDF triples are represented by Prolog fact `rdf/3`. Consider the following context query, which selects a contacts email and lastfmUsername:

```
c = Contact.one(email=eMail && lastfmUsername=lastfm)
```

This expression is translated to the following Prolog code:

```
subtype(SubContact,'http://.../Contact'),
rdf(C,rdf:type,SubContact),
rdf(C,'http://.../email',EMail),
rdf(C,'http://.../lastfmUsername',LastFm)
```

Slightly more complicated is the translation of *predicate mapping*:

```
Location.select().sortedBy(source.precision).last()
```

The expression is translated to the following query that binds the variable `Last` to the location with the highest precision:

```
subtype(Location,'http://.../Location'),
findall(_L,rdf(_L,rdf:type,SubContact),Locs),
findall(P, ( member(L,Locs),
             rdf(L,'http://.../source',Source),
             rdf(Source,'http://.../precision',P),
),Ps),
sortedBy(Locs,Ps,SortedLocs),
last(SortedLocs,Last)
```

25 We use the term *translation* here to avoid confusion with predicate *mapping*.

Notation

The target language for our translation is a subset of the Prolog language. The syntactic categories used in the following have already been introduced in Section 2.3.4 and are only repeated for easier reference:

$$V \in LVars, t \in Term, L \in Literal, B \in PosLiteral,$$
$$A \in Atom, \phi \in Formula, Cl \in Clause, \mathcal{P} \in Program, \mathcal{G} \in Goal$$

The single elements have the following syntax

$$\begin{array}{rcl} t & ::= & VT \mid f(t_1, ..., t_n) \mid A \\ VT & ::= & V \mid tuple(t_1, ..., t_n) \\ B & ::= & A(t_1, ..., t_n) \\ L & ::= & B \mid \backslash + B \\ A & ::= & string \mid number \\ \phi & ::= & \epsilon \mid L \mid \phi_1, \phi_2 \mid t_1 = t_2 \mid \phi_1 - > \phi_2; \phi_2 \mid findall(t, L, V) \mid V \: is \: f(t_1, ..., t_n) \\ Cl & ::= & A : -\phi \\ \mathcal{P} & ::= & Cl_1, ..., Cl_n \\ \mathcal{G} & ::= & ? - \phi \end{array}$$

where $f(t_1, ..., t_n)$ is restricted to arithmetic functions and arrays.

The following translation is based on the RDF entailment triples, therefore no subclasses or sub-properties have to be considered in the mapping. An implementation may use an RDFS reasoner, such as Cliopatria [2], to infer the entailment or replace the triples by predicates in the generated queries to reflect rdfs semantics. Section 6.2 elaborates this in more detail.

Let Id be the set of identifiers. The EBNF to Prolog translation is defined by the two functions

- $T_B[\![\mathcal{V}, \mathcal{E}, E]\!] \rightarrow \mathcal{S}, \phi$ translates boolean expression E to auxiliary predicates ($\mathcal{S}$) and a Prolog formula ϕ.
- $T_E[\![\mathcal{V}, \mathcal{E}, E]\!] \rightarrow \mathcal{S}, V : \tau/\phi$ translates expression E to auxiliary predicates ($\mathcal{S}$), a typed variable V and an optional Prolog formula ϕ that represents a prefix for the currently translated expression.

$\mathcal{E}$ represents the expression scope stack with scope elements $se \subset \mathcal{E}$, $se = x : T$, where $x \in Id$, $v \in LVar$. Capital letter E stands for EBNF expressions (terminal or non-terminal). The second parameter of T_E either uniquely matches an OCQL input expression, where each E_i takes the place of sub-expressions, or an appended *where* clause restricts the clause to a certain case. For comprehensibility, some of the expressions are named starting with a capital letter, e.g., *Template*.

Let $\mathcal{S}$ be the set of sub goals that are represented by terms of the form $pred(\mathcal{E}, Head, \{E_1, ...\})$, where $Head$ is the predicate head and E_i are OCQL expressions from the set of expressions, named $\mathcal{O}$. Section 4.3.4 describes how the $pred/3$ term is translated to Prolog clauses. Let e, V and t be translated Prolog terms, where e denotes an arbitrary term, V a Prolog variable, and t a tuple term of the form *tuple*(...). The argument names of tuples are represented by a and b. Finally, we define two functions to retrieve the corresponding RDF classes and property names for Java classes and methods. The function $c2r := JI \rightarrow \wp(RNVT)$ returns the corresponding set of non-value RDF classes representing a Java interface from JI:

$$c2r(\tau) := \begin{cases} rtj_p^{-1}(\tau_r) & \tau \in JI_b \\ \{rtj_p^{-1}(\tau_1), ...\} & \tau = I_{\{\tau_1, ...\}} \in JI_e \end{cases}$$

The function rtj_p^{-1} is the reverse function of rtj_p defined in Definition 4.1.1 and $RNVT$ is the set of RDF non-literal classes.

From the definition of JI_e this set is minimal, meaning it does not contain any classes in a subtype relationship. The partial function $m2r : JI \times Id \rightarrow URI$ translates a method name back to it corresponding property's URI. The context $\mathcal{E}$ maps identifiers to their types and the variable environment $\mathcal{V} \in \mathcal{V}s$ is a finite set of associations between identifiers and Prolog variables $V \in LVar$, in which $\mathcal{V}s$ is the set of variable environments. $\mathcal{V}(Id) = V$ means the identifier Id is mapped to V.

We use the symbol $\stackrel{\triangle}{=}$to denote the translation operation.

Translation

The first rule we consider, handles negated expressions:

$$T_B[\![\mathcal{V}, \mathcal{E} \vdash !E : boolean]\!] \stackrel{\triangle}{=} let\, h = \omega(\mathcal{V}, E)\, in \quad \text{(not)}$$
$$\mathcal{S} \cup \{pred(\mathcal{E}, \mathcal{V}, h, \{E\})\},\ \text{not(}\ h\ \text{)}$$

This rule creates a new predicate h based on the variables in expression E. It uses the auxiliary function $\omega : \mathcal{V}s \times \wp(\mathcal{O}) \rightarrow Literal$ that takes a variable environment and a set of OCQL expressions as arguments and returns a literal with a unique functor and all translated Prolog variables as arguments. For example, the expression

```
(String a, String b) : !p(a,b)
```

is translated to

```
not(pred_1(A,B))
```

The variable `a` is mapped to the Prolog variable `A` and `b` is mapped to `B`. The generated literal `pred_1(A,B)` is added to the set of sub goals $\mathcal{S}$, together with the environments $\mathcal{V}$ and $\mathcal{E}$.

The next case case handles the `if` condition, which is translated to the Prolog built-in if operator (`->`):

$$T_B[\![\mathcal{V}, \mathcal{E} \vdash \texttt{if}(E_1)\texttt{then}(E_2)\texttt{else}(E_3)]\!] \stackrel{\triangle}{=} \quad \text{(if)}$$
$$let\, \mathcal{S}_1, e_1 = T_B[\![\mathcal{E}, \mathcal{V}, E_1]\!]\, in$$
$$let\, \mathcal{S}_2, e_2 = T_B[\![\mathcal{E}, \mathcal{V}, E_2]\!]\, in$$
$$let\, \mathcal{S}_3, e_3 = T_B[\![\mathcal{E}, \mathcal{V}, E_3]\!]\, in$$
$$\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3, e1 \texttt{ -> } e_2 \texttt{ ; } e_3$$

The logical *or* expression $E_1 \,||\, E_2$ is translated to a predicate with two clauses, one for E_1 and one for E_2:

$$T_B[\![\mathcal{V}, \mathcal{E} \vdash E_1 \,||\, E_2]\!] \stackrel{\triangle}{=} let\, h = \omega(\{E_1, E_2\})\, in \quad \text{(or)}$$
$$\{pred(\mathcal{E}, \mathcal{V}, h, \{E_1, E_2\})\}, h$$

For the logical *and* expression E_1 && E_2, T_B recurses on the two sub expressions E_1 and E_2, builds the union of all sub goals generated in this recursion and returns the formula of comma-separated Prolog expressions e_1 and e_2.

$$T_B[\![\mathcal{V}, \mathcal{E} \vdash E_1 \ \&\&\ E_2]\!] \stackrel{\triangle}{=} \quad \text{(and)}$$
$$let\, \mathcal{S}_1, e_1 = T_B[\![\mathcal{E}, E_1]\!]\, in$$
$$let\, \mathcal{S}_2, e_2 = T_B[\![\mathcal{E}, E_2]\!]\, in$$
$$\mathcal{S}_1 \cup \mathcal{S}_2, e_1 \texttt{ , } e_2$$

The translation of the unification expression is similar to the last rule. But here the expressions are typed and the application of T_E returns prefix formulas $prefix_1$ and $prefix_2$ that bind the variables V_1 and V_2 and that are placed infront of the Prolog unification of the two variables. The types of the variables are not part of the final Prolog code. They annotate the code for the final type checking phase after the mode analysis has been applied. At the end of the type checking phase the type annotations are removed.

$$\begin{aligned}
&T_B[\![\mathcal{V},\mathcal{E} \vdash E_1 = E_2]\!] \triangleq \qquad\qquad \text{(unify)}\\
&\quad let\ \mathcal{S}_1, V_1 : \tau_1/prefix_1 = T_E[\![\mathcal{E},\mathcal{V},E_1]\!]\ in\\
&\quad let\ \mathcal{S}_2, V_2 : \tau_2/prefix_2 = T_E[\![\mathcal{E},\mathcal{V},E_2]\!]\ in\\
&\qquad \mathcal{S}_1 \cup \mathcal{S}_2, prefix_1,\ prefix_2\ , V_1 : \tau_1 = V_2 : \tau_2
\end{aligned}$$

The next two rules describe the translation of the singe-variable template findall and the tuple template variant:

$$\begin{aligned}
&T_B[\![\mathcal{V},\mathcal{E} \vdash \texttt{findall}(Var, Condition, BagVar)]\!] \triangleq \qquad\qquad \text{(findall-v)}\\
&\quad let\ \mathcal{S}, e = T_B[\![\mathcal{E},\mathcal{V}, Condition]\!]\ in\\
&\quad let\ V_1 = pvar(Var)\ in\\
&\quad let\ V_2 = pvar(BagVar)\ in\\
&\qquad S, \text{findall}(t,e,V)\\
&\text{where } \mathcal{E}(Name) = a(C, Dim)
\end{aligned}$$

$$\begin{aligned}
&T_B[\![\mathcal{V},\mathcal{E} \vdash \texttt{findall}(Var, Condition, v)]\!] \triangleq \qquad\qquad \text{(findall-v-t)}\\
&\quad let\ \mathcal{S}, e = T_B[\![\mathcal{E},\mathcal{V}, Condition]\!]\ in\\
&\quad let\ t = template(\mathcal{E},\mathcal{V}, tuple(a_i :\\
&\tau_{a_i}, ..), Template, Name)\ in\\
&\quad let\ V = \mathcal{V}(v)\ in\\
&\qquad S, \texttt{findall}(t,e,V)\\
&\text{where } \mathcal{E}(Name) = a(Tuple\{a_i : \tau_{a_i}, ..\}, 1)
\end{aligned}$$

The auxiliary function *template* generates a template term with all arguments bound to the variables selected in the template tuple and an anonymous variable for all ignored arguments:

$$\begin{aligned}
&template(\mathcal{E},\mathcal{V}, tuple(a_i : \tau_{a_i}, ..), \{b_1 =\\
&v_1, ...\}, Name)]\!] \triangleq\\
&\quad let\ V_1 = \begin{cases} \mathcal{V}(v_j) & if\ a_1 = b_j \\ _ & otherwise \end{cases}\ in\\
&\quad let\ V_2 = ...\\
&\qquad in\\
&\qquad tuple(V_1, ...)
\end{aligned}$$

For example, for the call `findall({a=v,c=v2},p(v,v2),bag)` with bag:Tuple{a:τ_1,b:τ_1,c:τ_1} the template returns `tuple(V,_,V2)`, assuming $\mathcal{V}$(`v`) =`V` and $\mathcal{V}$(`v2`) =`V2`.

Every arithmetic operation is translated to a separate evaluation literal, which is typed according to Figure 4.11, rule (arith-plus):

$$\begin{aligned}
&T_E[\![\mathcal{V},\mathcal{E} \vdash E_1\ BinaryArithOp\ E_2 : \tau]\!] \triangleq \qquad\qquad \text{(arithmetic expression)}\\
&\quad let\ \mathcal{S}_1, V_1 : \tau_1/p_1 = T_E[\![\mathcal{V},\mathcal{E} \vdash E_1 : \tau_1]\!]\ in\\
&\quad let\ \mathcal{S}_2, V_2 : \tau_2/p_2 = T_E[\![\mathcal{V},\mathcal{E} \vdash E_2 : \tau_2]\!]\ in\\
&\quad \mathcal{S}_1 \cup \mathcal{S}_2, V : \tau/p_1, p_2, V\ \texttt{is}\ V_1\ BinaryArithOp\ V_2
\end{aligned}$$

Empty arrays are type compatible with arrays of arbitrary component types:

$T_E[\![\mathcal{E},\{\}]\!] \stackrel{\triangle}{=} \emptyset, [] : \mathtt{a}(\bot, \mathtt{1})$ (empty array)

Nested arrays are recursively processed:

$$\begin{aligned} & T_E[\![\mathcal{E},\{\{Expr\}\}]\!] \stackrel{\triangle}{=} \qquad \text{(nested array)} \\ & \quad let\ V : a(\tau, N)/\phi = T_E[\![\mathcal{E},\mathcal{V},\{Expr\}]\!]\ in \\ & \quad let\ N1 = N+1\ in \\ & \qquad \phi, [V]: \mathtt{a}(\tau, \mathtt{N1}) \end{aligned}$$

A property access on a context expression first evaluates the context expression the identifier is selected on:

$$\begin{aligned} & T_E[\![\mathcal{E},\mathcal{V}, ContextExpression.Ident]\!] \stackrel{\triangle}{=} \qquad \text{(property selection)} \\ & \quad let\ V_0 : \tau_0/Prefixes = T_E[\![\mathcal{E},\mathcal{V}, ContextExpression)]\!]\ in \\ & \quad let\ V : \tau/BindSimpeName = \\ & simpleName(\mathcal{E},\mathcal{V},\tau_0, V_0, Ident) \\ & \qquad V : \tau/Prefixes{,}BindSimpeName \end{aligned}$$

Processing the ContextExpression results in the *Prefixes* formula and the variable V_0. The function *simpleName* is called, which translates *Ident* in the context of τ_0. In case the identifier is selected without further qualification, the context is retrieved from $\mathcal{E}$ and $\mathcal{V}$:

$$T_E[\![\mathcal{E},\mathcal{V}, Ident]\!] \stackrel{\triangle}{=}$$

$$\begin{cases} \mathcal{V}(Ident) : \tau/true & \mathcal{E} \vdash Ident : \tau \\ simpleName(\mathcal{E},\mathcal{V},\mathcal{E}(\mathtt{this}),\mathcal{V}(\mathtt{this}), Ident) & \mathcal{E} \not\vdash Ident : \tau \end{cases}$$

We will now consider the different cases of the *simpleName* function. A property access on a non-tuple, non-array type maps directly to the corresponding RDF triple:

$$\begin{aligned} & simpleName(\mathcal{E},\mathcal{V},\tau_0, V_0, Ident) \stackrel{\triangle}{=} \\ & \quad let\ pName = m2r(\tau_0, Ident)\ in \\ & \quad let\ \tau_{arg} = \Delta(Ident : \tau_0)\ in \\ & \qquad V : \tau_{arg}/\mathtt{rdf(}V_0\mathtt{,}pName\mathtt{,}V\mathtt{)} \\ & \text{where } \tau_0 \neq \text{Tuple}\{Args\} \wedge \tau_0 \neq a(\tau', Dim) \end{aligned}$$

The function $m2r$ resolves the full property name and via Δ the type of the property. In case that τ_0 is a Tuple type *simpleName* returns a prefix, which binds the corresponding tuple argument to the variable V_{Arg}:

$$\begin{aligned} & simpleName(\mathcal{E},\mathcal{V},\tau_0, V_0, Ident) \stackrel{\triangle}{=} \\ & \quad let\ V_{Arg} : \tau_{arg}, TupleArgs = tupleQueryArgs(\text{Tuple}\{Args\}, Ident) \text{ in} \\ & \qquad V_{Arg} : \tau_{arg}/V_0 \mathtt{\ =\ tuple(}TupleArgs\mathtt{)} \\ & \text{where } \tau_0 = \text{Tuple}\{Args\} \end{aligned}$$

$$\begin{aligned} & tupleQueryArgs(Tuple\{a_1 : \tau_1, ...\}, ArgName) \stackrel{\triangle}{=} \\ & \quad let\ \{\tau\} = \{\tau_i | a_i = ArgName\} \\ & \quad V : \tau/if\ (a_1 = ArgName)\ V\ else\ _\ ,if\ (a_2 = ArgName)\ V\ else\ _\ ,.... \end{aligned}$$

For example, consider the variable declaration

```
Tuple{a:int,b:double} v
```

and the argument access

```
v.a
```

The argument access is translated into:

V_0 `= tuple(`V`,_)`

The function *tupleQueryArgs* is applied to generate a comma separated list of arguments, which equal the first returned variable and otherwise anonymous Prolog variables (_), e.g.,

`tupleQueryArgs(Tuple{a:int,b:int},a)=V:int/V,_`

The next clause handles array types. The findall predicate backtracks over all members of V_0 and calls the formula *ComponentExpr* generated for the component type. The *comp* and *arrayOf* functions defined in the beginning of this section are utilized to decrease resp. increase the dimension of the array type.

$$
\begin{aligned}
&simpleName(\mathcal{E}, \mathcal{V}, \tau_0, V_0, Ident) \stackrel{\triangle}{=} \\
&\quad let\ \tau_{0comp} = comp(\tau_0)\ in \\
&\quad let\ V_{member} \in LVar\ in \\
&\\
&let\ V_{comp} : \tau_{comp}/ComponentExpr = simpleName(\mathcal{E}, \mathcal{V}, \tau_{0comp}, V_{member}, Ident)\ in \\
&\quad let\ \tau = arrayOf(\tau_{comp})\ in \\
&\qquad V_{List} : \tau/\texttt{findall}(V_{comp}, \texttt{member(}V_{member}\texttt{,}V_0\texttt{),(}ComponentExpr\texttt{),}V_{List}\texttt{)} \\
&\text{where } \tau_0 = a(\tau', Dim)
\end{aligned}
$$

For example, called with a property access p on a nested array $C[][]$, $simpleName(\mathcal{E}, \mathcal{V}, C[][], V_0, p)$ with $m2r(p) = prop$ results in the formula:

$$
\begin{aligned}
&\texttt{findall(}V_{comp}\texttt{,} \\
&\ \texttt{(member(}V_{member}\texttt{,}V_0\texttt{),} \\
&\quad \texttt{findall(}V_{comp'}\texttt{,} \\
&\qquad \texttt{(member(}V_{member'}\texttt{,}V_{member}\texttt{),rdf(}V_{member'}\texttt{,}prop\texttt{,}V_{comp'}\texttt{)} \\
&\qquad \texttt{),}V_{List'}\texttt{)} \\
&\ \texttt{),}V_{List}\texttt{)}
\end{aligned}
$$

The class predicates $C->$ `select` and $C->$ `one` define the implicit variable `this` $: C$ and may define an explicit instance variable v. We only consider the case where the instance variable v is defined. Both v and `this` are added to the type and variable environment. In the `select` case a `findall` expression gathers all bindings that fullfill the condition *Condition*:

$$
\begin{aligned}
&T_E[\![\mathcal{E}, \mathcal{V}, C-> \texttt{select}(v|Condition)]\!] \stackrel{\triangle}{=} && \text{(select)} \\
&\quad let\ call = \omega(\{Condition\})\ in \\
&\quad let\ V \in LVar\ in \\
&\{pred(\mathcal{E} \cup^{\backslash} \{v : C, \texttt{this} : C\}, \mathcal{V} \cup^{\backslash} \{v : V, \texttt{this} : V\}, h, \{Condition\})\}, \\
&\quad V_{All} : a(C,1)/\texttt{findall(}V\texttt{,}call\texttt{,}V_{All}\texttt{)}
\end{aligned}
$$

In case of the `one` predicate only the environments are extended and T_E is called recursively on *Condition*:

$$
\begin{aligned}
&T_E[\![\mathcal{E}, \mathcal{V}, C-> \texttt{one(v}|Condition)]\!] \stackrel{\triangle}{=} && \text{(one)} \\
&\quad let\ V \in LVar\ in \\
&\quad let\ \mathcal{S}, V_0/prefix = \\
&\qquad T_E[\![\mathcal{E} \cup^{\backslash} \{v : C, \texttt{this} : C\}, \mathcal{V} \cup^{\backslash} \{v : V, \texttt{this} : V\}, Condition]\!]\ in \\
&\mathcal{S}, V : C/prefix
\end{aligned}
$$

Property existence checks are processed similarly:

$T_B[\![\mathcal{E},\mathcal{V}, contextExpression-> \texttt{exists}(Ident_1....Ident_n)]\!] \stackrel{\triangle}{=}$ (exists check)
$let\, \mathcal{S}, V : \tau, \phi = T_E[\![\mathcal{E} \cup^{\setminus} \{\texttt{this} : C\}, \mathcal{V} \cup^{\setminus} \{\texttt{this} : V\}, Ident_1....Ident_n]\!]\, in$
$\mathcal{S}, /\texttt{once}((\phi))$

Here the resulting literal is a pure check, which is wrapped by the once/1 literal, so backtracking is avoided. Now we consider the different predicate mapping cases described in Section 4.2.4. The first case is $N_0, N_1...N_r \rightarrow N$. The mapping is applied on a context expression with an array type. A predicate p is mapped onto this expression and its first argument type is a supertype of the context expression's component type:

$T_E[\![\mathcal{E},\mathcal{V}, contextExpression : \tau-> p(e_1 : \tau_{a_1}, ..., e_r : \tau_{a_r})]\!] \stackrel{\triangle}{=}$ (map-n-n)
$let\, \tau_0 \times \tau_1 \times ... \times \tau_{r+1} = \Delta(p)\, in$
$let\, \mathcal{S}, V_{exprs} : \tau, \phi_{prefix} = T_E[\![\mathcal{E},\mathcal{V}, contextExpression]\!]\, in$
$let\, \{V, V_{out}, V_{all}\} = uniqueVars()\, in$
$let\, \mathcal{S}_p, \{V_{p1}, ...V_{pr}\}/\phi_{params} = genParamsList(\tau, V, e_1, ..., e_r)\, in$
$\mathcal{S} \cup \mathcal{S}_p, V_{all} : a(C,1)/\phi_{prefix},$
$\texttt{findall}(V_{out},$
$(\texttt{member}(V, V_{exprs}), \phi_{params}, p(V, V_{p1},..., V_{pr} V_{out})),$
$V_{all})$
where $\tau = a(\tau', Dim) \wedge \tau_0 :> comp(\tau) \wedge \tau_{r+1} = a(\tau'_{r+1}, Dim_{r+1}) \wedge Dim = Dim_{r+1}$

The predicate facilitates the *genParamList* function that evaluates the arguments of the mapping in the context of the component type. Therefore we replace or add `this` to the environment and evaluate T_E on the sub-expressions.

$genParamsList(\tau, V, t_1, ..., t_n) \stackrel{\triangle}{=}$
$let\, \mathcal{S}_1, V_1, \phi_1 = T_E[\![\mathcal{E} \cup^{\setminus} \{\texttt{this} : \tau\}, \mathcal{V} \cup^{\setminus} \{\texttt{this} : V\}, t_1]\!]\, in$
...
$let\, \mathcal{S}_n, V_n, \phi_n = ...\, in$
$\bigcup_i \mathcal{S}_i, \{V_1, ...V_n\}/\phi_1,...,\phi_n$

The second mapping case $N_0, N_1...N_r \rightarrow R$ is shown below:

$T_E[\![\mathcal{E},\mathcal{V}, contextExpression : \tau-> p(t_1 : \tau_1, ..., t_r : \tau_r)]\!] \stackrel{\triangle}{=}$ (map-n-r)
$let\, \tau_0 \times \tau_1 \times ... \times \tau_{r+1} = \Delta(p)\, in$
$let\, \mathcal{S}, V_{exprs} : \tau, \phi_{prefix} = T_E[\![\mathcal{E},\mathcal{V}, contextExpression]\!]\, in$
$let\, \{V_1, ..., V_r, V_{all_1}, ..., V_{all_r}, V_{out}\} = uniqueVars()\, in$
$let\, \mathcal{S}_p, \phi_{param_1}, ..., \phi_{param_r}, \{V_{p1}, ...V_{pr}\} = genParamsList(\tau, V, t_1, ..., t_r)\, in$
$\mathcal{S} \cup \mathcal{S}_p, V_{out} : a(C,1)/\phi_{prefix},$
$\texttt{findall}(V_{all_1}, (\texttt{member}(V, V_{exprs}), \phi_{params_1}), V_{all_1}), ...$
$\texttt{findall}(V_{all_r}, (\texttt{member}(V, V_{exprs}), \phi_{params_r}), V_{all_r}),$
$p(V_{exprs}, V_{all_1}, ..., V_{all_r}, V_{out})$
where $\tau = a(\tau', Dim) \wedge \tau_0 :> comp(\tau) \wedge \tau_{r+1} = a(\tau'_{r+1}, Dim_{r+1}) \wedge Dim = Dim_{r+1}$

Initially, the solutions for the *contextExpression* are bound to V_{exprs}. The function *genParamsList* generates the Prolog formulas for all sub-expressions t_i. Since they are evaluated separately for each member of V_{expr} a findall literal is added to each formula ϕ_{params_i} to collect the bindings in the variables V_{all_i}. Now the mapped predicate p is called with V_{exprs} and the lists of mapped parameters V_{all_i}. This call exclusively binds the output parameter V_{out}.

Sub-predicate Processing

To complete the translation process, we iterate on the collected set S of sub-predicates $pred(\mathcal{E}, \mathcal{V}, h, \{E_1, ..\})$. For every expression E_i we generate a new clause with head h and the translation of expression E_i as its body. The clause *sub-pred* formalizes this step. It is iterated on all the sub-expressions S_i of E_i. This iteration terminates since we only operate on sub-expressions.

$$
\begin{aligned}
& T_E[\![pred(\mathcal{E}, \mathcal{V}, h, \{E_1, ..., E_n\})]\!] \triangleq && \text{(sub-pred)} \\
& \quad let\ \mathcal{S}_1, \phi_1 = T_E[\![\mathcal{E}, \mathcal{V}, E_1]\!]\ in \\
& \quad ... \\
& \quad let\ \mathcal{S}_n, \phi_n = T_E[\![\mathcal{E}, \mathcal{V}, E_n]\!]\ in \\
& \quad \mathcal{S}_1 \cup ... \cup \mathcal{S}_n, h\texttt{:-}\phi_1 . ... h\texttt{:-}\phi_n .
\end{aligned}
$$

The transformation rules are applied recursively on an OCQL expression until it is completely transformed into a type-annotated Prolog goal and predicates.

4.3.5 *Mode Analysis and Final Type Checking*

Now we apply our adaptation of Lu's mode analysis from Chapter 3 to the generated Prolog code. The analysis annotates every program point pp with mode information $[v/m]$ about every variable v used in the enclosing clause.

We reconsider all unifications and predicate calls that were postponed in the first check. The function $mo(V, pp) \to \{f, g, o\}$ represents the mode of the variable V at the program point pp.

First, we validate that the mode of each predicate mapping *expr->p*(a_1, ...) is according to the assumptions we made in the pre-mode analysis type checking phase. The predicate mapping is translated to a predicate call $(\mathrm{p}(V_1,...,V_n))_{pp}$, where the following premise must hold:

$$
\frac{mo(V_1, pp^-), ..., mo(V_{n-1}, pp^-) = \{g\}, mo(V_n, pp) = \{g\}}{\mathcal{E} \vdash (p(V_1, ..., V_n))_{pp} : Formula}
$$

Meaning, before the call (program point pp^-) the variable V_1 bound to the result of *expr* must be ground and all the expressions that have been evaluated on the *expr* ($V_2, ..., V_{n-1}$). After the evaluation (pp) the variable in the last argument must be ground, too ($mo(V_n, pp) = \{g\}$). This is necessary, since the types for type parameters of mapped predicates are inferred under this assumption in the initial type checking part.

Second, the type check of unifications and predicate calls is completed. Since the initial type checking step already tested the type compatibility relation $\lesseqgtr$ we do not need to include the subtype condition in the premise.

Figure 4.15 lists the corresponding typing rules.

The rule *unify-g* and *unif-t-eq* consider the trivial cases of ground operands and operands of the same type. In both cases no illegal dataflow between subtypes is possible.

The following example illustrates a problem in presence of type parameters. The superscript ^{m}, with $m \subseteq \Delta$, denotes a mode of a variable at the given program point:

```
<T extends C> pred(T p) : (S_l l):
  l=S_l->one() &&
  p^{f}= l^{g}; // typing error, although bound_Δ(T) :> S_l
```

Let's assume S_l <: C and S_v <: C, and the predicate `pred` is called with an unbound variable v of type S_v:

```
(S_v v): p(v)
```

The parameter `p` is unified with variable `v` of type S_v. In the body of `pred`, the parameter `p` is unified with `l` of type S_l. This unification is not type safe. Therefore non-ground

$$\frac{mo(V_1, pp) = mo(V_2, pp) = \{g\}}{\mathcal{E} \vdash (V_1 : \tau_1 = V_2 : \tau_2)_{pp} : Formula} \; (unify\text{-}g)$$

$$\frac{\tau_1 = \tau_2}{\mathcal{E} \vdash (V_1 : \tau_1 = V_2 : \tau_2)_{pp} : Formula} \; (unify\text{-}t\text{-}eq)$$

$$\frac{\begin{array}{c}\tau_1 \neq \tau_2, mo(V_1, pp) = \{g\}, mo(V_2, pp) \neq \{g\}, \\ tp(\tau_2) = \emptyset, bound_\Delta(\tau_1) <: \tau_2\end{array}}{\mathcal{E} \vdash (V_1 : \tau_1 = V_2 : \tau_2)_{pp} : Formula} \; (unify\text{-}no\text{-}tp)^*$$

$$\frac{\begin{array}{c}\Delta \vdash < \overline{T} > p : \overline{\tau}, in(pp, \overline{\zeta}, \overline{\tau}) = (\overline{\zeta'}, \overline{\tau'}), infer(\overline{T}, \overline{\tau'}, \overline{\zeta'}) = \overline{T} : \overline{\eta}, \\ out(pp, \overline{\zeta}, \overline{\tau}) = (\overline{\zeta''}, \overline{\tau''})\end{array}}{\mathcal{E} \vdash p(a_1 : \zeta_1, ...)_{pp} : Formula} \; (gen\text{-}pred\text{-}call)$$

$$\frac{mapped(pp), mo(V_1, pp) = ...mo(V_r, pp) = \{g\}}{\mathcal{E} \vdash p(V_1, ...V_r, V_{out})_{pp} : Formula} \; (mapped\text{-}pred\text{-}call)$$

*the case where lhs and rhs are switched is analogous

Figure 4.15: Final typing rules for delayed checking

variables that contain type parameters in their type τ must not be unified with a (concrete) subtype of τ or its bounds.

The premise of rule *unify-no-tp* rules out the case where V_2 contains type parameters because the concrete type of the parameter is not known and must only be unified with variables with the exact same type (parameter), which is covered by *unify-t-eq*.

In the Java type system array types are covariant, meaning $\tau_1 <: \tau_2 \Rightarrow \tau_1[] <: \tau_2[]$. This can lead to runtime errors, when values are assigned to an array element, because the component type of the array can differ from the declared type of a variable that has statically been type checked:

```
Integer ints = new Integer[1];
Object[] objs = ints;
objs[0] = "value"; // <- runtime error
```

Arrays are also covariant in OCQL, but we avoided the need for runtime type information. Since arrays are immutable once bound, the rule *unify-no-tp* statically ensures that modifications after the unification are not possible. Considering the example above, the assignment of `ints` to `objs` is only valid if `ints` is ground. This assignment is also valid in case that τ_1 contains a type parameter. Again, the array cannot be modified and therefore a variable can safely be bound to a variable $v : \tau_2$, where $bound_\Delta(\tau_1) <: \tau_2$[26].

The rule *gen-pred-call* considers generic predicate calls and facilitates the *infer* function from Section 4.3.3 to infer the output types.

Rule *mapped-pred-call* ensures that the first r arguments of a mapped predicated call are ground, because the translation to Prolog is based on this assumption. It facilitates a function *mapped*, which checks if a program point represents a mapped predicate call.

Figure 4.16 shows the final typing rules for findall calls with single variable templates. They are similar to the first rules in Figure 4.15. Instead of an explicit unification they type the unification of the array of bindings for V and the V_{bag} variable. The mode for the V_{bag} variable is taken from the program point pp before the call of findall. The mode for the array of bindings is known at the next program point pp^+, after findall has backtracked over all solutions for $Condition$. The rule *findall-no-tp* tests the mode $\{g\}$ for V and not the mode of the array of all bindings. But for mode $\{g\}$ the modes are obviously the same. The typing rules for the findall tuple template are very similar and are therefore omitted.

26 This is similar to the typing of *out* type parameters in C#, a construct that allows read-only covariance in C# Generics Albahari and Albahari [8].

$$\frac{mo(V, pp^{+}) = mo(V_{bag}, pp) = \{g\}, \mathcal{E} \vdash Condition : Formula}{\mathcal{E} \vdash \texttt{findall}(V : \tau_1, Condition, V_{bag} : \tau_2)_{pp} : Formula} \; (\textit{findall-g})$$

$$\frac{\tau_1 = \tau_2, \mathcal{E} \vdash Condition : Formula}{\mathcal{E} \vdash \texttt{findall}(V : \tau_1, Condition, V_{bag} : \phi_2)_{pp} : Formula} \; (\textit{findall-eq})$$

$$\frac{\begin{array}{c} mo(V, pp^{+}) = \{g\}, mo(V_{bag}, pp) \neq \{g\}, tp(\tau_2) = \emptyset, \\ bound_{\Delta}(\tau_1[]) <: \tau_2, \mathcal{E} \vdash Condition : Formula \end{array}}{\mathcal{E} \vdash \texttt{findall}(V : \tau_1, Condition, V_{bag} : \phi_2)_{pp} : Formula} \; (\textit{findall-no-tp})^{*}$$

Figure 4.16: Final typing rules for findall calls with a single variable template
*the case where lhs and rhs are switched is analogous

4.3.6 *Mode Checking for Other Unsafe Expressions*

Next to type checking, the mode analysis is used to detect unbound variables in arithmetic expressions and string operations. Variables with mode o and f must not be passed to an argument expecting a ground variable, e.g., the following expression is not valid, since the variable `i` is not ground:

```
(int i) : i + 1 > 0
```

Type casts are checked type conversions to a subtype, therefore casted expressions must be ground.

4.4 SUMMARY

This chapter introduced the syntax and semantics of the Object Context Query Language, a statically typed logic language based on a polymorphic type system with subtypes. The type system is based on the context interface hierarchy JI_e, which contains greatest lower bounds for the partial order on the subtype relationship. Next to context interfaces it builds on the Java basic type wrapper classes. In contrast to Java, the wrapper classes reflect the subset relationship of the contained values, e.g., Float is a subtype of Integer. These component types are complemented by polymorphic array types, and tuple types over array and component types.

In the second part we have shown a three-step type checking method based on typing rules, mode analysis and a reduced variant of the type parameter inference algorithm for Java. Finally, we described the semantics of the language by a mapping to a subset of Prolog.

5 CONTEXT MANAGEMENT INFRASTRUCTURE AND SERVICE ASPECTS

This Chapter presents an OSGi-based context management infrastructure (CMI), which facilitates RDF Schema (RDFS) for the specification of context models. We present the general architecture of the infrastructure, its adaptation facilities, and how OCQL can be used or embedded by host languages. We close the Chapter with an application of the CMI to aspect-orientation programming. We present the aspect-language CSLogicAJ, which embeds OCQL into its pointcut language and makes use of the adaptation and context query facilities.

5.1 CONTEXT MANAGEMENT INFRASTRUCTURE

The Semantic Web stack was designed to aggregate data and data schemas from different sources and reason on this aggregated information. A large number of context management approaches have argued for Semantic Web-based context modeling [169, 204, 205], since this matches the demands for aggregation of context information well.

The CMI uses RDF Schema since it is the common base for semantic web based knowledge representation languages. This ensures compatibility with a large number of existing RDF resources on the web. The semantically richer OWL family can be integrated via RDFS serializations [72].

The central component is an RDF-based *context management system* (CMS), responsible for context data aggregation, managing context requests, and evaluating queries. The Figure 5.1 depicts an overview of the system.

Context sources are realized as OSGi services implementing the interface IContextSource, see Figure 5.2. They either push RDF data to the context aggregator on context changes or the context aggregator polls for changes via the getSnapshot() method. Every context modification leads to a new snapshot of the context data following a common approach for context-management systems [213]. By default older snapshots are only kept while they are locked by a query. Context consumers may lock older snapshots in order to refer to historic contexts, e.g., previous locations. Figure 5.3 gives an example. At first the context data is updated by a context source resulting in snapshot s_1. After that, query

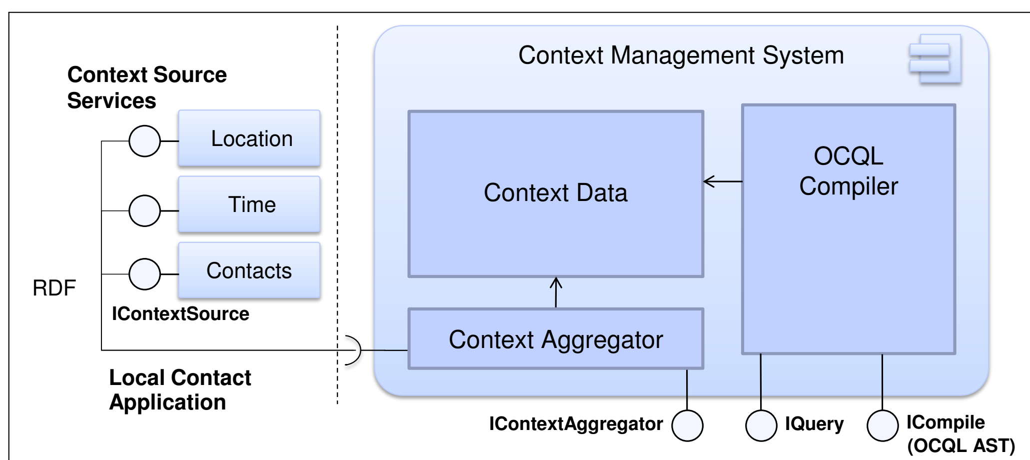


Figure 5.1: Overview of the context management system

```
public interface IContextSource {
    String getId();
    String[] getRDFSchemas();
void start(IContextAggregator aggregator);
void stop();
InputStream getSnapshot();
}
```

Figure 5.2: The IContextProvider interface

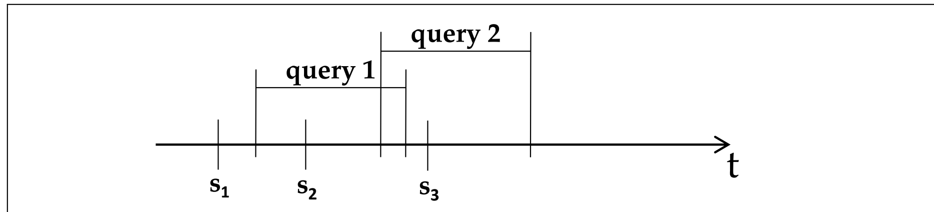


Figure 5.3: Context data snapshots

1 starts on snapshot s_1. A new snapshot s_2 is created that does not influence query 1. Now query 2 is starts and is evaluated on s_2. Once query 1 is processed, snapshot s_1 is released and after query 2 is finished snapshot s_2 is deleted. The snapshot s_3 is created while query 2 is running, but does not effect the queries.

An OCQL query can result in follow-up queries, since context objects are returned in a shallow fashion. Resources referenced by property relationships are retrieved lazily via a proxy and therefore a query must block its queried snapshot until all relevant data has been retrieved.

This cannot be automatized in general, since the system does not know which properties will be accessed[1]. Consequently, clients must manually manage the lifetime of query by explicitly invalidating a service request once the processing of query results has come to an end.

Clients use the `IQuery` interface shown in Figure 5.4 to query for contexts. A query returns an object of type `IResult` which encapsulates the binding of variables contained in the query and the snapshot timestamp. Once the result is fully processed a client must call the IResult.release() method. The system calls this method once `IResult` is garbage collected or the client bundle is stopped. But clients should free the snapshot immediately after the result is fully processed to release system resources. The request parameter will be explain in Section 5.1.3.

A query is always evaluated on a fixed set of context source snapshots as well as their associated schemata[2] and is not affected by snapshots created in parallel or other context source data requested by other queries. The RDF schemas are passed to the query methods in the first parameter. The second parameter takes the whole query. Alternatively a precompiled predicate can be called via the `queryCompiledPredicate` method. It has the same parameters as the `query` method, but additionally the caller must specify the *module*, the predicate was defined in. OCQL takes over the module concept from Prolog[3], Section 5.1.2 will explain predicate compilation and encapsulation in more detail.

1 Only if the whole transitive closure of referenced properties has been accessed.
2 including its entailed rdf triples, e.g., the closure of the subtype relationship
3 See Section 2.3.2.

```
public interface IQuery {
  IResult query(ContextRequest request,
                String[] rdfSchemas,
                String query) throws OCQLException;
  IResult queryCompiledPredicate(String module, ...);
  ...
}
public interface IResult {
  long getSnapshot();
  HashMap<String,Object> getBindings();
  release();
}
```

Figure 5.4: Requesting context - dynamic approach

5.1.1 *Context Listeners*

Besides normal context queries, the CMS supports context change listeners. For example, a client can react to updated sensor information or service arrival/departure. Figure 5.5 contains the corresponding listener registration methods. A client passes a `IContextChangeListener` instance to the query which is notified once the queried context information changes. The second to fourth parameters are equivalent to the synchronous query and take the request, rdf schemas, and query expression to be observed.

The last parameter defines under which constraints the listener is notified. A logic expression on mapped predicates compares the current results with the results from the previous notification. Each predicate is applied to variables declared in the query. The predicates are mapped to the list of current and the previous substitutions of the variable, following the scheme from Section 4.2.4. But here the list of the previous results is prefixed to the arguments, so that the predicates can compare the new and old values:

$predicateName(OldList, NewList, OptArg_1, \ldots)$

Two comparison predicates are predefined for changed expressions: `ne/2` and `th/3`. The first checks if the lists are equal, the second if a certain threshold is exceeded. The expression below triggers the listener if the value of variable `v1` has changed and the difference between the old and new value of v2 is greater than 1:

```
"ne(v1) && th(v2,1)"
```

The predicate `ne/2` compares primitive types and context class types. For the later it applies a shallow comparison of context class properties. First, it checks if the URI of the instance itself changed. Next, it checks if any direct property has changed. Arbitrary predicates, e.g., a deep equivalence check, can be defined and used instead.

5.1.2 *OCQL Compilation*

Queries and listeners can use pre-compiled predicates. Figure 5.6 shows an excerpt of the compile interface. The pre-compilation has two advantages. First, the pre-compilation avoids runtime compilation overhead. Second, the static correctness of the queries can be checked before the query is executed. Either at configuration time of the application or even statically by a compiler, as realized in the CSLogicAJ aspect language in Section 5.4. The `compile`() method takes a module name as a first parameter. All OCQL predicates contained in the src parameter are compiled into this module. To query these predicates `IQuery.queryCompiledPredicate`() takes the module name as its first argument.

```
public interface IQuery {
  ...
  void addContextListener(
     IContextChangeListener listener,
     ContextRequest request,
     String[] rdfSchemas,
     String query,
     String changedExpression ) throws OCQLException;
  void addCompiledContextListener(String module, ...) ... ;
}
public interface IContextChangeListener {
    void changed(IResult result);
    ContextRequest getRequest();
}
```

Figure 5.5: The query and listener interfaces

```
public interface ICompile {
  void compile(String module,
            String[] rdfSchemas,
            String src) throws OCQLException;
  void compileAST(...) throws OCQLException;
}
```

Figure 5.6: The compilation interface

The OCQL was designed to be embedded in other languages. Host languages can parse OCQL as part of the language's common compilation step. The parser builds an abstract syntax tree (AST), which does not have to be build again. The CMI takes the parsed AST as an input. It applies static analysis, type checking, and compiles the AST to Prolog. The input format for the AST is a nested compound term representation derived from the OCQL EBNF given in 4.7. Each parsed non-terminal (*NT*) is represented as a compound term with the lower letter functor *NT*. For example, the token `int` is compiled to `qualifiedName(identifier('int'),[])`, representing the non-terminal *QualifiedName*. Sub-expressions are represented as comma-separated arguments and star expressions like (*expr*)* are wrapped by brackets, corresponding to Prolog lists. Below is an excerpt of the OCQL EBNF we will need for the mapping:

Predicate ::= 'predicate' [**TParams**] **Ident** '(' **Formals** ')' [':' '(' **Formals** ')'] ':' **Condition** ';'
CompOperation ::= **TermOrParen Comparator TermOrParen**
Comparator ::= '='| '<' | '>' | '<=' | '>='
TermOrParen ::= **ParenTerm** | **Term**
ParenTerm ::= '(' **Term** ')'
Term ::= **QualifiedName** | **Int** | ...
QualifiedName ::= **Identifier** | 'this' ('.' **Identifier**)*

Figure 5.7 illustrates the mapping of a predicate `p` which compares the argument `var` with the value 1. At first an AST is build up from the concrete syntax. We use abbreviations for the CompOperation, QualifiedName, etc. Every non-terminal symbol is mapped to an AST node and references its right-hand side symbols. Each terminal node refers to a terminal symbol, which represent the leaf of the tree. In the serialized form each node wraps the sub-symbols as a compound term. Optional sub-rules, which have been

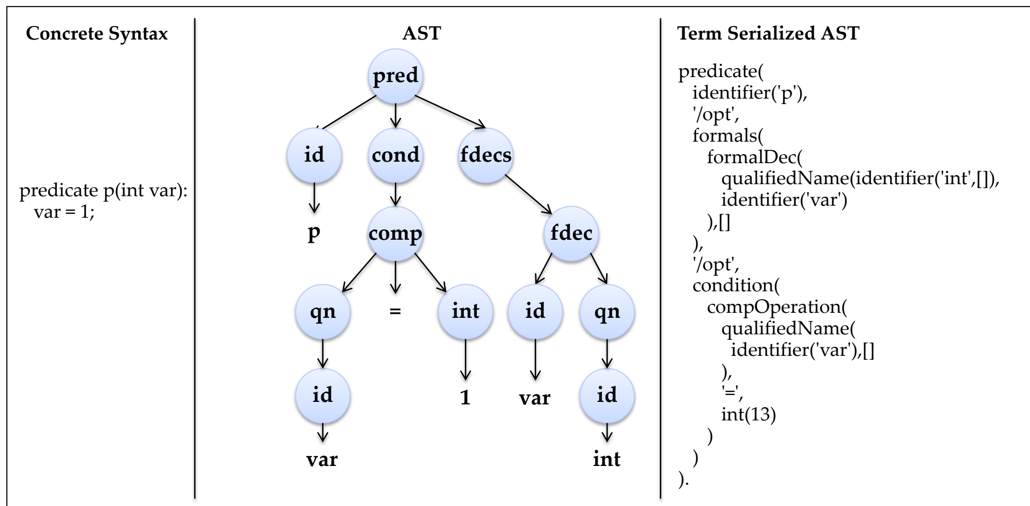


Figure 5.7: Mapping from concrete to abstract syntax and to a term-serialized form

rejected by the parser, are replaced by the `'/opt'` atom. The implementation Chapter 6 will describe the analysis and compilation of the AST to a Prolog predicate further.

5.1.3 *Requesting Context*

A calling client or the host language[4], which embeds the query, must specify which context sources should be queried. For this purpose the framework offers different ways to request context sources. A client can either request context sources explicitly by their type or attributes, or implicitly by specifying the RDF schemas a context source must provide.

The first is suitable if a concrete source, e.g., a GPS sensor, is needed by a client. The second is useful if only the context kind, e.g., location information, is relevant for the client and the active context source may freely be chosen and switched by the system. Additionally, both requests can be combined with a LDAP filter string to filter context sources via meta-data attached to the context source services. The LDAP filter matching of services is already built into OSGi service queries, see Section 2.1.2. The filter allows for fine-grained selection of context-sources based on attributes attached to context sources. The predefined filters include equality and comparison checks of attribute values of a single source.

LDAP filters lack the means of weighting service attributes, which is often necessary to select between different context sources. For this reason, we added the `min`(*attribute*) and `max`(*attribute*) conditions which are true for the sensor with the minimum resp. maximum property value. If several services have the same attribute value the condition is true for an arbitrary service. Figure 5.8 shows a service interface used to request contexts. The parameters of the request methods have the following meaning:

source is an array of context sources types, which must be subtypes of IContextSource.

schema is an array of schema URLs

strategy is filter expression string following the syntax described above. If left empty ("") no further strategy is applied. The system will use the filter (objectclass=org.cs3.context.IContextSource) in this case.

cardinality is the cardinality of the request. It is specified via an enumeration with the options "single" and "multiple". In case of the "single" option only one (arbitrary)

4 In the following we will use the term client to subsume both a runtime client and host language.

```
public interface IRequest {
  ContextRequest requestSource(
       BundleContext bc,
       Class source,
       String strategy,
       Cardinality cardinality,
       int history);
  ContextRequest requestSchema(
       BundleContext bc,
       String schema,
       String strategy,
       Cardinality cardinality,
       int history);
  public void removeRequest(ContextRequest schema);
...
}
public interface ContextRequest {
  void release();
}
```

Figure 5.8: Requesting context - dynamic approach

context is chosen which matches the filter. Otherwise all matching context sources are selected at once. In the later case the data from all context sources is available at once.

history is the number of snapshots that should be kept. The parameter is 0 if no additional snapshot should be kept. Queries can facilitate previous snapshot via the `history` predicate, see Section 4.2.5.

Context sources can also be requested and removed dynamically via the IRequest interface. The IRequest API allows a client to change the requested contexts at runtime, for instance based on program state or user interaction. For example, a user might be asked which address book service should be used to reason about contacts in context-sensitive application. Each request is represented as a ContextRequest instance and must be removed, once the context is not needed anymore by calling the `release`() method. Similar to the context snapshots this happens automatically once the enclosing bundle is stopped or the ContextRequest object is garbage collected.

5.1.4 *Queries and Language Integration*

Context-query languages typically follow two different approaches for integrating a context model into a programming language.

The first category of approaches define the context model in a host programming language like Java[5]. This has the advantages that only one compiler is needed, static type safety is easily achieved and processing of query results need no further type mapping.

Approaches of the second category use a different type system for the query language. Here, the query language types are optionally mapped to host language's types after completing a query. A lot of semantic-web based approaches [169, 204, 205] and object-relational mappers for relational databases belong to this category. A non-fixed type binding to a statically typed language is reasonable in case of a Semantic Web-based approach, since a class definition may vary over different queries in the system. For example, a type Contact has a lastfmUsername property for queries related to the last.fm

5 e.g., McFadden et al. [137]

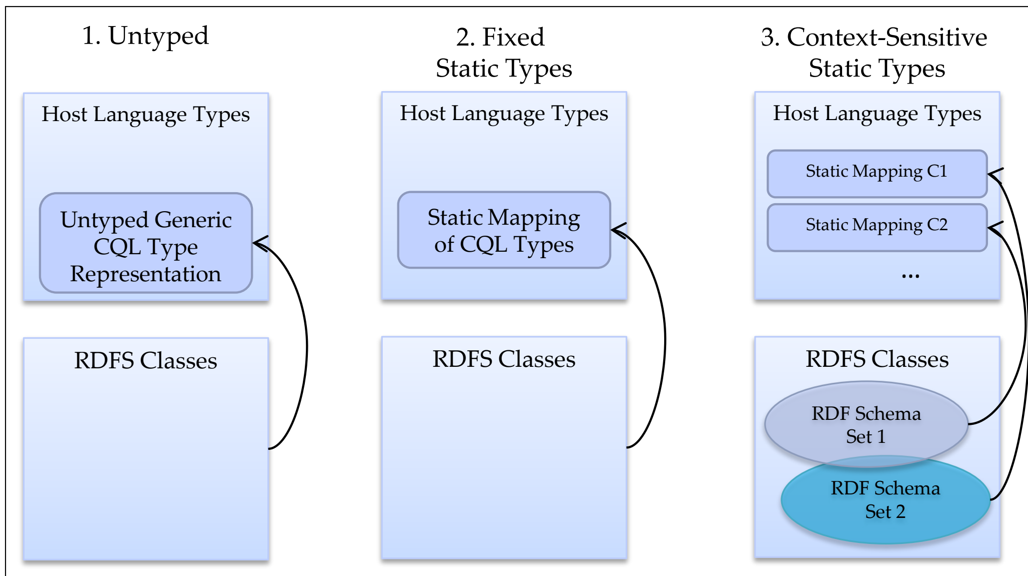


Figure 5.9: Three different alternatives of integrating OCQL with a host language

adaptation described in the introduction. Other queries, which have been independently developed, use the Contact type without this property. This approach is more flexible, since the type of a context entity may change dynamically when a new adaptation and sensors are loaded into the system.

But also more effort is necessary in processing query results, since context types do not have a fixed mapping to host language types. Since the tradeoff between the two approaches is not in general solvable the CMI supports different levels of host language integration with RDFS types:

1. A client does not embed OCQL. OCQL queries are send as strings to the query method of the IQuery interface. The results of the queries are bound to a generic objects graph that contains key-value pairs representing RDF properties. All nodes are of the type `IContext` depicted in Figure 5.10. In [175] we applied this variant to the context-oriented programming language JCop. The JCop OCQL integration is further discussed in Section 7.2.

2. All RDF schemas are statically known. The RDFS classes are once mapped to Java interfaces and clients are compiled against these interfaces. In case that no runtime loading of querying clients will happen, this is a suitable solution and does not involve further language extensions.

3. A language extension statically types context classes only against the referenced RDFS classes in their usage context. Meaning different adaptation/query modules[6] using different sets of RDFS schemas are compiled against different context classes. Once a context class is passed out of module the graph looses all its static typing. In this case the object graph based on `IContext` object from option 1 is passed to other modules. So the third solution is a tradeoff between option 1 and 2 and its application is most suitable when the results of a query is not passed outside the query's module. The aspect language presented in Section 5.4 follows this approach. Each aspect is considered as a module with a separated context model.

Figure 5.9 illustrates the three different approaches and different mappings of RDFS classes to Java types. The CMS offers two `IQuery` services. The first returns untyped

6 The adaptation module could be an OSGi plugin, an aspect or other modules encapsulating code.

results (service with attribute typed="false") and the second typed results (attribute typed="true"). Variant 1 and 3 use the untyped approach, variant 2 the typed approach. For the typed approach the calling client must ensure that corresponding Java context interfaces are available, their packages are exported and imported by the calling client bundle.

We will now illustrate variant 1 and 2 with small examples and combine the second variant with pre-compiled predicates. The most flexible but completely untyped solution uses the generic IContext interface for representing context. Let's assume we want to retrieve our own contact entry from the address book, which is provided by a context source based on the schema introduced in Section 4.1.2. We extend the RDFS class `Contact` with a boolean property `me`, marking the contact entry of the device's user:

```
@prefix ex: <http://www.example.org/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
ex:Contact rdf:type rdfs:Class.
ex:me rdf:type rdf:Property;
      rdfs:domain ex:Contact;
      rdfs:range xsd:boolean.
```

Untyped Dynamically Compiled Queries

At first we need a reference to the untyped IQuery service. Service retrieval in the CMI is based on the *ClientService*, which will be presented in detail Section 5.3. Here we assume an instance of the ClientService is already available. It provides the `getService`() method, which we use to retrieve a service of type `IQuery` with the attribute `typed=false`:

```
BundleContext bc;
IRequest ir = clientService.getService(IRequest.class);
ContextRequest cr =
  ir.requestSource(bc,AddressBook.class,"",Cardinality.single);
IQuery iq = clientService.getService(IQuery.class, "(typed=false)");
```

As a first parameter we pass an array of RDF schemata URLs the query is compiled with. The second argument is an OCQL query string. It queries for the users own Contact entry in the address book. The `Contact` instance is marked with the boolean property `me`:

```
IResult r = iq.query(
  cr,
  new String[]{"http://sam.iai.uni-bonn.de/cmi/contact.ttl"},
  "namespace ex = \"http://www.example.com\";" +
  "(ex.Contact c) : c = ex.Contact->(me)");
String surname =
 (String)((IContext)r.getBindings().get("c")).getProperty("surname")[0];
iq.releaseQuery(r);
```

The query is dynamically compiled against the `contact.ttl` RDF schema and returns an `IResult` object containing a list of variable bindings. The Contact object is represented by an object of type IContext. Via the getProperty() method we retrieve the `surname` property of type Object, which we cast to the type String. An application of this variant for context-oriented programming is presented in Section 7.2.

Typed Pre-Compiled Queries

The compiled variant is very similar, but separates the compilation and query step. The typed variant of the `IQuery` service returns JI_e types. The ICompile interface is retrieved similar to the IQuery interface, but here no further LDAP attributes must be given, since only one service exists. We assume the `ir` and `cr` instances are already set. The predicate

```
public interface IContext {
  public String getSensorId();
  public Object[] getProperty(String fieldName);
  public String[] getPropertyNames();
  public long getTimestamp();
  public String getContextClass();
  public IContext castTo(String uri);
  public boolean isSubTypeOf(String uri);
}
```

Figure 5.10: Generic context class IContext

me() is compiled into module module1 and queried from the same module. The bindings of the result are still typed Object, but we can now cast it to the type Contact, which contains the method getSurname().

```
ICompile ic = clientService.getService(ICompile.class);
IQuery iq = clientService.getService(IQuery.class,"(typed=true)");
// get IQuery and ICompile service
ic.compile("module1",
     cr,
     new String[]{"http://sam.iai.uni-bonn.de/cmi/contact.ttl"},
     "namespace ex = \"http://www.example.com\";" +
     "predicate me(ex.Contact c) :  c = ex.Contact->(me)");
IResult r = iq.query( "module1", "(Contact c) : me(c)");
String surname = ((Contact)r.getBindings().get("c")).getSurname()[0];
```

5.2 QUERY CONTEXT SOURCES

The *query context sources* (QCS), introduced in Section 4.2.6, are implemented in Java. They implement the ContextSource interface and contain one *query method* for each query. Since there is not necessary a Java representation for a context class[7] we cannot refer to mapped rdf classes in regular Java code. For this reason each query method has an RDFSClass annotation attached, representing the RDF class returned by the method. The only parameter must be of the type Map<String,Object> taking key value pairs with values of type string or basic types.

Figure 5.11 illustrates this with the last.fm testeometer Web service that is wrapped in an equally named method. Query methods must return an instance of the type *Snapshot* that contains the expiration time of the data, and the RDF graph. The type declaration in the @RDFSClass annotation describes the returned RDF type of the method, which is necessary for the static type safety of the query language.

The connection to the Web service and the mapping to RDF is in the responsibility of the ContextSource.

5.3 SERVICE DISCOVERY AND INTERCEPTION

The CMI builds on Ditrios[8] [174], an OSGi extension for service management and interception. The interception mechanism offers means for runtime service adaptation. A full description of Ditrios can be found in [180], here we only describe the parts relevant for this thesis.

7 except for mapping variant 2 in Section 5.1.4
8 **DI**stributed **TR**acking and **I**nterception **O**f **S**ervices

```
public interface Lastfm extends ContextSource {
  @RDFClass(type="http://fm.last.Artist",list=true)
  Snapshot testeometer(Map<String,Object> parameters);
}
interface Snapshot {
   int getExpirationTime();
   InputStream getData();
}
```

Figure 5.11: Lastfm context source

```
public interface ClientService {
   <T> T getService(Clazz<T> clazz);
   <T> T getService(Clazz<T> clazz, String ldapFilterExpression);
   ...
}
```

Figure 5.12: The client service interface, Ditrios' entry point for service searching and tracking

Searching, tracking and service provisioning is realized by Ditrios while remaining fully transparent to service consumer. Ditrios replaces the OSGi API for service searching and tracking with the *ClientService* interface. The ClientService establishes communication between client applications and Ditrios and thereby provides access to registered OSGi services. Figure 5.12 shows an excerpt of the ClientService interface and its `getService`() methods[9], which retrieve a service of type `clazz`. They configure and return a proxy instance to the consumer bundle[10]. This indirection enables flexible control over services and allows for generic adaptation triggered by method calls. Ditrios offers an service interception hook which is used by the aspect language CSLogicAJ for runtime weaving of aspects (see Section 5.4). A client inquires services via its ClientService by means of the common OSGi LDAP filter definition.

5.3.1 *Proxy Indirection*

All services belonging to the same filter are wrapped within a proxy object. An arbitrary service is chosen as the default service which is then transparently utilized by the client. This default service is switched automatically once the bound service becomes unavailable and alternatives are available. This behavior can further be configured by the service adaptation mechanism described in the Section 5.3.2. So clients usually do not have to be aware of the proxy indirection and let the architecture handle stale references, service unavailability, and service substitutions.

Clients initiate their communication with the Ditrios framework by acquiring a ClientService instance which is thenceforward exclusively assigned to it. Instead of retrieving and tracking a Ditrios API service the ClientService itself is designed as a service and only needs to be registered with the framework in order to be used. Configuration and tracking of the ClientService is realized by Ditrios. This dependency inversion has also been named the OSGi *Whiteboard pattern* [10].

9 Clients referring to all registered services are not considered in this thesis. Similar getServices() methods can easily be added to Ditrios, but would have bloated this Chapter with additional cases which mostly repeat the single service case.

10 The full specification of the `ClientService` can be found in [180], where it is named `DitriosClientService` instead.

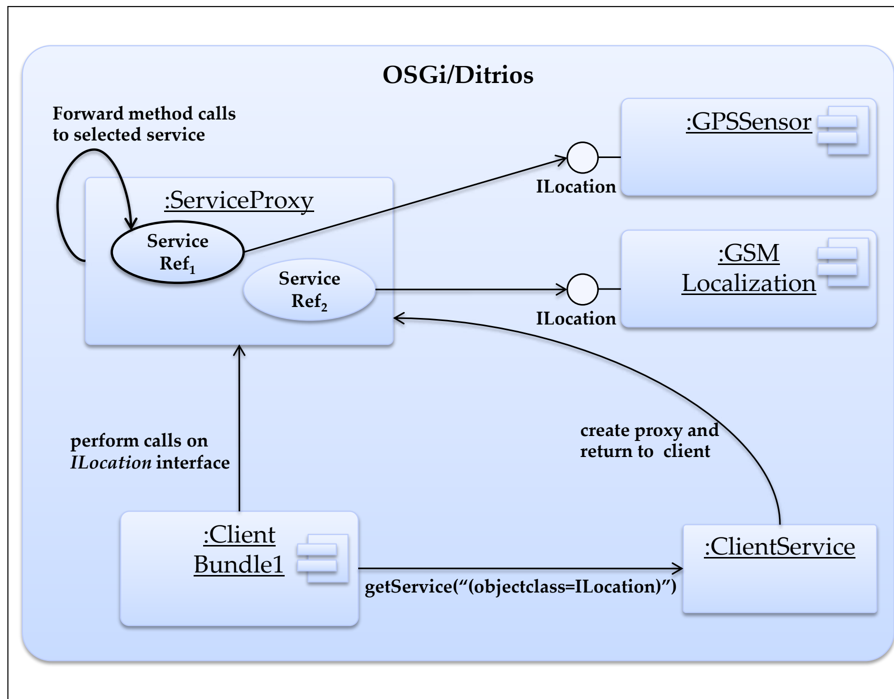


Figure 5.13: Ditrios service lookup workflow

After that the client requests one or more services by defining a search request. All requests are delegated with the help of the ClientService to the Ditrios core system that in turn attempts to track the corresponding services. A proxy instance wrapping all matching services will be created for each request. References to the proxies are then returned to the ClientService so that the owning client can utilize them over the API. Every status change of the requested services is reflected by an event, which informs corresponding clients provided that they implement the appropriate event listener.

Figure 5.13 depicts the workflow. An instance of `ClientBundle1` requests a service with interface `ILocation` from the `ClientService` instance, which returns a `Proxy` instance configured with references to services registered by `GPSSensor` and `GSMLocalization` bundles. One of the services is chosen by default, since no priority is defined in the filter expression.

5.3.2 *Service Adaptation*

In the CMI, service adaptation is achieved by modifying attributes, reconfiguration the service references, or by service call interception.

The later is supported by Ditrios, which offers an infrastructure for method call interception. Interceptors must implement the interface `DitriosInterceptor`, see Figure 5.14, and register themselves as services. Having done that, the interceptors are able to intercept all service method calls.

An interceptor calls the original method or activates the next interceptor by calling `processor.proceed`(..) following a common AOP technique. The target service and arguments might be altered in between. An interceptor might change the default service by accessing the corresponding ServiceProxy via the `DitriosFacade`.

An interceptor must provide a priority value; by default it is 0. Interceptors with higher values have precedence and are executed before interceptors with lower values.

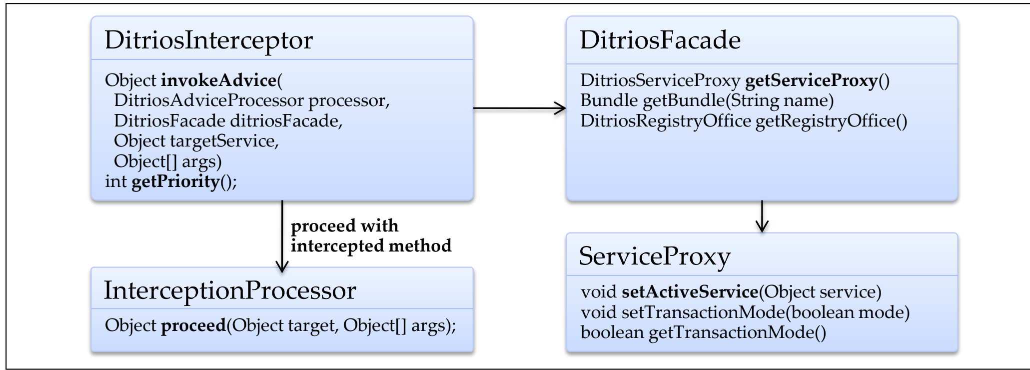


Figure 5.14: Ditrios Invocation Handler

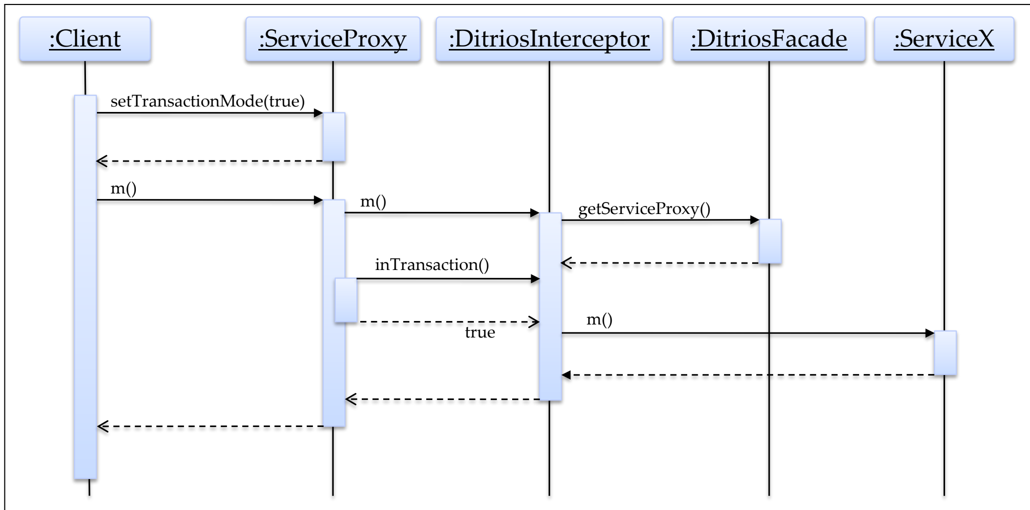


Figure 5.15: Services are transaction-aware

5.3.3 *Transaction-awareness*

Some service adaptations, such as upgrades and substitutions, must not occur while a client is in a conflicting processing state. For example, consider a table view processing a number of entries from a service. In case the service is switched while the tree is filled, the table will contain entries from different services. Hence, Ditrios offers the client to set transaction boundaries wherein no context change, service upgrade, etc. may affect the program flow. A client can cast a service reference to the ServiceProxy interface and set the transaction mode. An interceptor can access this information via DitriosFacade.getServiceProxy().getTransactionMode(). In case it tries to modify the active service the exception ProxyLockedException is thrown.

Figure 5.15 illustrates the usage in a sequence diagram. Let's assume a client has retrieved a service reference to a service with interface `ServiceX` containing the method m(), which should be set into transaction mode and one interceptor is registered in the system. At first the client sets the transaction by casting the service reference to the `ServiceProxy` interface. Then it calls the method `m()` on the original service interface. The service proxy redirects the call to the interceptor, which now checks if the proxy is in an transaction. The call returns true and the interceptor just forwards the call to the original instance. This forwarding happens in the `InterceptionProcessor` as described in the previous Section, but we left out this indirection here for brevity.

5.4 CONTEXT-SENSITIVE SERVICE ASPECTS

This Section presents *context-sensitive service aspects*, a context-sensitive adaptation means building on the CMI and OCQL, which follows the third variant of OCQL language integration from Section 5.1.4. Service aspects operate on service proxies and control the dynamic composition of services. Building up the glue between context information and services they enable the latter to be context-sensitive.

Service aspects are implemented in the language CSLogicAJ (Context-aware Service-oriented Logic Aspects for Java). CSLogicAJ combines a subset of the AspectJ language[11] - the advice constructs and the *call, target, this* and *args* pointcuts - with OCQL expressions and a *service* pointcut that binds OSGi service instances. We distinguish between two different kinds of advice, both of them share the same syntax, but have different semantics.

Synchronous Advice are applicable to the service message level. Such an advice intercepts calls to service interfaces, taking the program flow and all available context information into account. The commonly known *before*, *after* and *around* advices can be used to execute additional code. Furthermore a transparent (re)binding and (re)composition of services is possible due to the proxy concept. Synchronous advices correspond to the common dynamic aspect weaving.

Asynchronous Advice react exclusively on context changes, which are induced by *context sources*. These advices are comparable to event-condition-action (ECA) rules known from relational databases. The *event* is the change of the context, the *condition* is represented by a pointcut designator and the *action* by the advice code. The poincut designator must not contain the pointcuts *call, target, this* and *args*, since they select program join points and their execution context.

The terms *asynchronous* and *synchronous* refer to the execution time relative to the application work flow. Based on these two kinds of advice the architecture controls service composition and dynamic code weaving. All information gathered and provided by context sources is consolidated in the context management system, see Section 5.1. CSLogicAJ mostly provides a concrete syntax to combine the context query and context listener mechanisms with an aspect language.

CSLogicAJ aspects are encapsulated in OSGi bundles where each bundle can hold one or more service aspects. The aspects are woven/unwoven, by starting/stopping their enclosing bundle.

5.4.1 *Context Pointcut Language*

The *Context Pointcut Language* is a subset of the AspectJ pointcut language extended by OCQL and service-level join points. Figure 5.17 gives the syntax of the pointcut designators as an extension of OCQL. The Pcd non-terminal is extended by the AspectJ primitive pointcuts `call`, `target`, `this` and `args`, and the additional pointcut `service` which binds service instances to pointcut or predicate parameters. The pointcut designator of a synchronous advice must contain a call pointcut to select the service methods to intercept. The *target(param)* pointcut binds the called service to its argument.

AspectJ's pointcut language can be considered a logic language with a reduced form of logic variables. All pointcut and advice parameters are bound by the primitive pointcuts *args*, *target*, *this*. An implicit join point variable is bound by the enclosing advice and restricted by primitive pointcuts. The aspect-language *Carma* [92] even made the join point variable explicit in the pointcut language. The join point variable is explicitly named and bound by advices, and passed to primitive pointcuts, which test and restrict it further. Here is an exemplary syntax how an explicit join point variable would look like in CSLogicAJ:

11 see Section 2.6

```
( onchange | before | after | around )
 name(var_decls)[ changed(expr) ] :
   [ (var_decls) : ]
  PointcutExpression
{
      java_body_and_proceed
}
```

Figure 5.16: Syntax of CSLogicAJ's generalized advice construct.

```
before(JoinPoint jp):
 call(jp, * Musicplayer.play()) { ... }
```

In CSLogicAJ the join point variable is hidden from the developer, but has the same semantics as an explicit variable. By integrating OCQL, CSLogicAJ becomes a general logic language. It enables the definition of local variables and the binding of logic variables by unification. Pointcuts, just as OCQL predicates, can define parameters of primitive types, context classes, array and tuple types. All advice and pointcut parameters are universally quantified, resulting in potentially more than one binding for a common AspectJ join point. An advice might therefore be executed more than once.

Primitive pointcuts restrict the implicit join point variable. This implicit variable is passed over every named pointcut in the evaluation of a pointcut designator. Therefore calling (primitive) pointcuts only has semantics in the context of a pointcut designator and must not be used in conditionals or meta-predicates.

We decided to keep the join point variable of pointcuts implicit and distinguish more general between *pointcut* and *predicate* constructs. Pointcuts may contain AspectJ's primitive pointcuts, but are not allowed to be used in context conditions or meta-predicates.. Figure 5.16 shows an excerpt of CSLogicAJ's advice construct. The asynchronous advice is determined by the *onchange*[12] keyword.

Around advices can use the proceed construct like AspectJ around advices, but must omit all context arguments in the proceed call, since the pointcuts of each advice is evaluated independently. The exerpt of the filterModel advice gives an example:

```
List around filterModel(Contact[] nolfms,IAddress t) :
  call( * IAddress.getAddresses() ) &&
  target(t) &&
  contactsWithoutLastfm(nolfms)
{
   ...
   return proceed(t);
}
```

The advice intercepts the `getAddresses()` method of the `IAddress` service, which offer access to an address book model. The target pointcut binds the target object to the variable t and passes it to the proceed call. The predicate `contactsWithoutLastfm` binds all contacts without last.fm user names to the variable `nolfms`. Since it is a context variable it is not passed to the proceed call.

12 Which is an abbreviation for "on context change".

```
Pcd ::=
   Identifier '(' Term ( ',' Term ) * ')'
   | 'call' '(' MethodPat ')'
   | 'args' '(' ArgsPat ')'
   | 'target' '(' Identifer ')'
   | 'this' '(' Identifer ')'
   | 'service' '(' Identifer [ ',' LDAPString ] ')'
MethodPat ::=
  [ Modifiers ] TypePat '.' IdPattern '(' Formals ')'
Formals ::=
  TypePat ( ',' TypePat) * [ ',' '..' ]
  | '..'
IdPat ::=
  ( '?' | '*' | Character ) ( '?' | '*' | Character | Number ) *
TypePat ::=
  IdPat ( . IdPat )* ( '[]' ) * [ '+' ]
ArgsPat::=
  Identifier ( ',' Identifier) * [ ',' '..' ]
```

Figure 5.17: EBNF of Primitive Pointcuts

```
onchange locationChanged(Location l, MapView map) changed(ne(l)) :
    l = Location->one() &&
    service(map,{centered="true"})
{
   map.setCenter(l.longitude,l.latitude);
}
```

Figure 5.18: Service level logging call pointcut.

5.4.2 *Service Pointcut*

Advices refer to registered OSGi services via the *service* pointcut, with the following syntax:

'service' '(' **Identifer** [',' **TupleInit**]')'

The first argument is bound to service instances, whose objectclass property matches the declared type of the logic variable, e.g., the parameter `map` in the following advice:

```
onchange locationChanged(MapView map)  :
    service(map)  { ... }
```

Here, the variable `map` is bound to `MapView` services. Optionally the LDAP properties of the service can be queried in the second argument. The non-terminal **TupleInit**, defined in Figure 4.7, represents a list of LDAP property unifications. The example in the next section facilitates this query argument in Figure 5.18.

5.4.3 *Asynchronous Onchange Advice*

The pointcut expression is reevaluated every time the context information it refers to changes. But, the onchange advice is only triggered if at least one of the arguments of the `changed(..)` expression has changed. In case no changed expression is given, it reacts to any change of an advice parameter. The expression argument embeds the *changed expression* from context listeners in Section 5.1.1. Figure 5.18 gives an example for an onchanged advice. The advice `locationChanged` binds the current location and a service of type MapView with the OSGi property "centered" with the value "true".

```
public @interface RequestContext {
  Class[] source() default {};
  String[] schema() default "";
  String[] strategy() default "";
  Cardinality[] cardinality() default "single";
}
public enum Cardinality { single, multiple }
```

Figure 5.19: Requesting Context - Static Approach

5.4.4 *First-Class Join Point*

Advices can access the runtime context of a join point via the field `thisJoinPoint`, as defined in Figure 2.6. Compared to AspectJ, the semantics of the JointPoint instance is slightly changed. The `getThis()` method returns the calling bundle of the call, `getTarget()` returns the target service instance and `getArgs()` the arguments of the method. Further the join point has access to the `DitriosFacade`[13], which provides access to the service proxy and the bundle management. For example, the following advice starts up the address book bundle:

```
after ditriosEvent(...) :
{
    // open address book application:
    thisJoinPoint.getDitriosFacade().getBundle("org.cs3.AddressBook").start();
  }
}
```

5.4.5 *Referring to Context Sources*

The static definition of a context request is realized as an Java annotation attached to an aspect. Consider the following example:

```
@RequestContext(
   contextSource={LocationSensor.class},
   strategy={"max(precision)"}
aspect A {...}
```

The aspect A requests a context source of type `LocationSensor` with maximum precision of all registered services of `LocationSensor` type. As long as the aspect is active the context request is active and the corresponding set of context sources are updated based on the given strategy. The attributes, such as precision, may change at runtime. Section 2.1.2 illustrates how context sources can update their attributes at runtime. Figure 5.19 contains the source code of the `RequestContext` interface. Its members are aligned with the `IRequest` interface defined in Section 5.1.3.

Each aspect independently specifies which RDF type schemas and context sensors are needed by the aspect. Schemas are referenced via the directive `import_schema`, corresponding namespaces are defined as in OCQL.

For example, the LastFMAspect below defines a namespace `contact`, imports a contact schema from an URL, and refers to a `Contacts` and a `Nearby` context source:

```
namespace contact = "http://laj.iai.uni-bonn.de/contact/";
import_schema "http://sam.iai.uni-bonn.de/cmi/contact.ttl";
import contact.*;
```

13 The facade was introduced in Section 5.14.

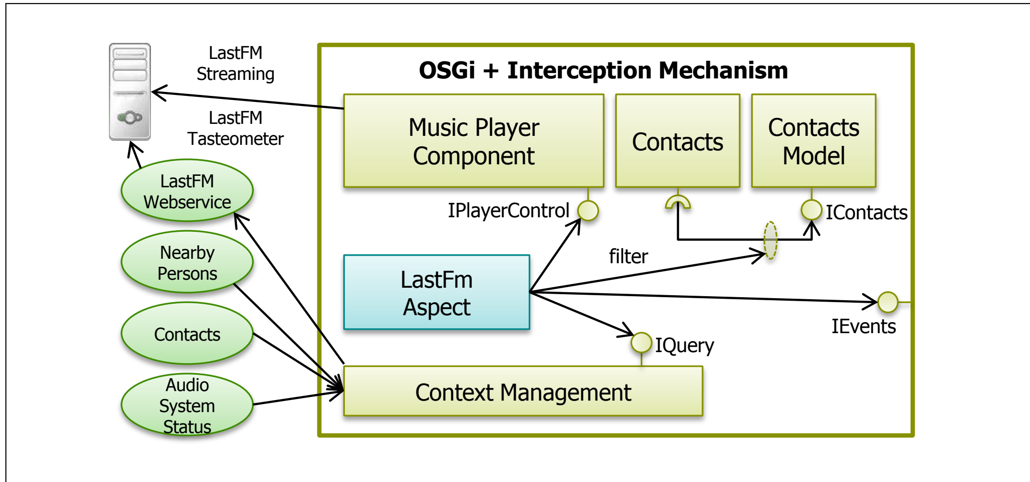


Figure 5.20: Context-Sensitive Music Player, modularized by an aspect

```
predicate nearby(Contact other, double maxDistance): (Nearby n):
  n = Nearby->one(distance < maxDistance) &&
  other = Contact->one(email = n.email);
predicate sharedArtistsWithOne(Artist[] artists,double maxDistance) :
  (Contact me, Contact other):
   me = Contact->one( this.isMe ) &&
   nearby(other,maxDistance) &&
   lastfm(artists,me.lastfmUsername,other.lastfmUsername);
```

Figure 5.21: Binding different pointcuts

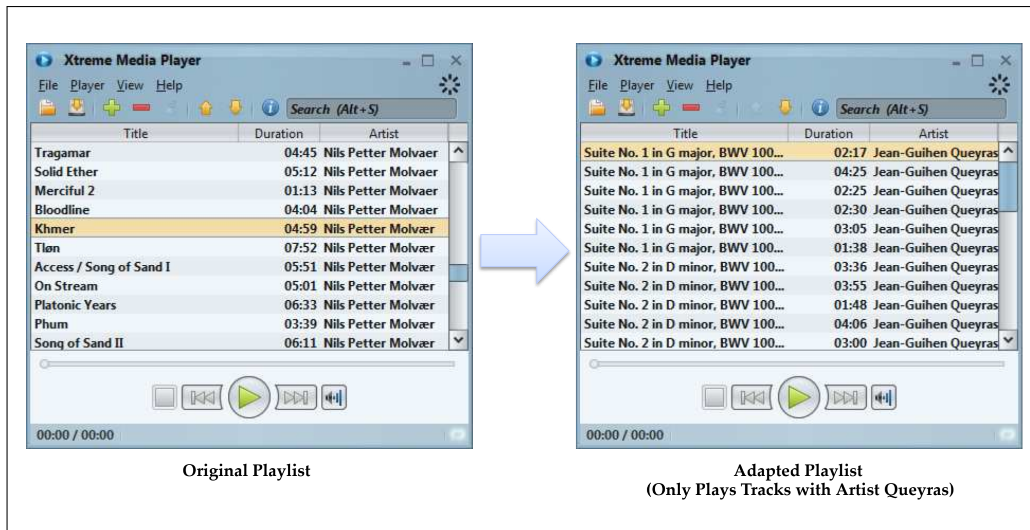
```
@RequestContext(sources={Nearby.class,Contacts.class})
aspect LastFMAspect { ... }
```

5.5 MUSIC PLAYER EXAMPLE REVISITED

In this section we use CSLogicAJ to realize the example from Section 1.1. We have built on the two open-source projects Xtreme Media Player [31] and Oracle's Java DB AddressBook example project [147] and converted them into OSGi bundles. The address book application was modularized into a view and a model bundle as depicted in Figure 5.20.

The main task of the LastFMAspect is to reconfigure the player's playlist based on context changes (nearby contacts, connected audio system, and tasteometer). Additionally, it reacts to the startup of the player. In case that there are contacts nearby, which do not have last.fm listed, the *Contacts* application is opened and only shows these persons in the contact's list.

For the first task, the aspect must aggregate information about the nearby contacts and their taste of music. In order to retrieve all artists, which a user and his nearby contacts both like, we introduce two pointcuts, `sharedArtistsWithOne` and `sharedArtists`. The purpose of the first one, `sharedArtistsWithOne`, is to retrieve a contact and the equally liked artists for this contact. The pointcut definition is given in Figure 5.21. The predicate makes use of the `Nearby` class, to bind all contacts currently in the vicinity of the user. The vicinity is indicated by `maxDistance`. Afterwards the `lastfm` Web service is utilized to retrieve the artists that the user's and contacts's both like. The `lastfm` predicate call



Original Playlist

Adapted Playlist
(Only Plays Tracks with Artist Queyras)

Figure 5.22: Context Sensitive Media Player Adaptation

```
predicate sharedArtists(Artist[] artists,double maxDistance):
    (Artist[] sharedWithOne, Artist[][] listOfList):
    AudioSystem->one( connected ) &&
    findall(sharedWithOne,
        sharedArtistsWithOne(sharedWithOne, maxDistance),
    listOfList) &&
    intersection(listOfList, artists);
```

Figure 5.23: The sharedArtist pointcut binds artist by collecting all artis from nearby contacts with the findall meta-call.

is only successful, if the `lastfmUsername` property is available for both, the user and the contact. By backtracking over this pointcut all contacts for whom these constraints hold true can be selected one after another.

The second pointcut is called `sharedArtists`. It checks that the device is connected to an audio system and aggregates mutually liked artists by backtracking over all results of the `sharedArtistsWithOne` pointcut and collects them in the variable `listOfList`.

Afterwards the intersection of these lists is calculated, resulting in a list of artists for whom a mutual preference is shared by all participants. See Figure 5.23 for more details on the implementation.

The LastFm aspect makes of use the pointcuts in the onchange advice `play`. It is triggered once the list of artists, bound by the `sharedArtists` predicate, changes. The second argument of the predicate is the maximum distance in meters to nearby contacts. See Figure 5.24.

Finally we notify the user about missing lastfm usernames of nearby contacts at the startup of the music player. Figure 5.25 shows how service events are intercepted and all nearby contacts with missing lastfmUsername property are bound to he contact array `contacts`. Figure 5.26 shows the filtered address book application. Only entries without last.fm entries are shown.

```
aspect LastFm {
// ... pointcuts ...
  onchange play(Artist[] artists) : changed(artist):
   sharedArtists(artists,100.0) {
 if(artists.length > 0){
  MusicPlayer.play(artists);
 }
}
```

Figure 5.24: Last.fm onchange advice

```
predicate contactsWithoutLastfm(Contact[] cs):
    cs = Contact->select(current | Nearby->one(current.email=email &&
                !exists(current.lastfmUsername)));
pointcut eventFired(ServiceEvent event) :
    execution( * EventManager.serviceEvent(..) )&&
    args(event);
after ditriosEvent(ServiceEvent event) :
      (Contact[] contacts, Contact contact, Nearby n) :
  eventFired(event) &&
  contactsWithoutLastfm(contacts)
{
  if(event.getType() == ServiceEvent.BUNDLE_STARTED && ..) {
    // notify user about missing lastfm usernames cs:
    ...
    // open address book application:
   thisJoinPoint.getDitriosFacade().getBundle("org.cs3.AddressBook").start();
  }
}
```

Figure 5.25: Intercepting service events

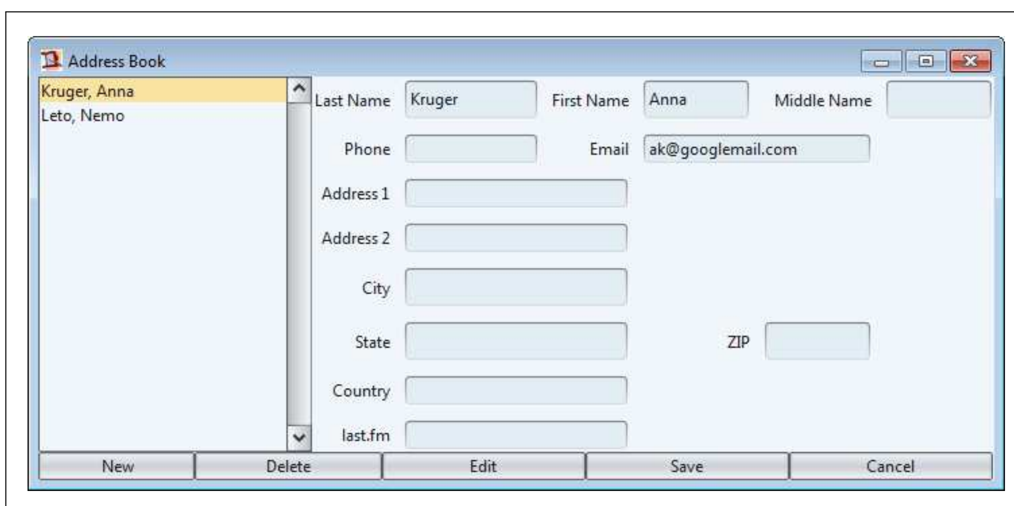


Figure 5.26: Filtered Address Book Entries

```
List around filterModel(Contact[] nolfms) :
  call( * IAddress.getAddresses() ) &&
  contactsWithoutLastfm(nolfms)
 {
    List addresses = proceed(nolfms);
    return filterList(nolfms,addresses);
}
```

Figure 5.27: Around Advice Example with Proceed Statement

5.6 RECONSIDERING REQUIREMENTS

This section analysis how CSLogicAJ adheres to the requirements for context-aware adaptation languages from Section 1.4.

CSLogicAJ and its embedded OCQL allow retrieving *sets of elements*, specification of *filters and conditions*, querying context *meta data*, *subscriptions on asynchronous context change events*, and the use of *logic operators* and therefore fulfill requirements 1-5. Context aggregation and selection (requirement 6) can either be realized by context requests by defining context selection strategies, or in OCQL by aggregating different context sources via predicates. The *history* predicate offers means to manage and access context histories (requirement 7).

Requirement 8 demands that context queries and sources are loosely coupled. CMI context sources do not have a reference to queries at all, and queries either refer to a context source interface or the RDF schema provided by a source. By no means it can directly refer to one concrete context source instance.

By requirement 9, context sources can reside on the device or are connected remotely. The CMI distinguishes between local *context sources* and *query context sources*[14], where the later enable predefined queries to external systems.

Requirement 10, the service communication interception, is a central concept of Ditrios and CSLogicAJ. Requirement 11 asks for an interception facility for infrastructure events and control over the component life cycle. The first is supported by the EventManager, the second by the DitriosFacade that gives aspects access to an component management API.

Ditrios aspects are deployed as bundles and each advice is represented as a service. By starting/stopping a bundle the aspects are woven/removed from the system. This fulfills the last requirements (12), the dynamic (de-)activation of adaptations. The implementation Chapter 6 will go into details how aspects are encapsulated in bundles.

5.7 SUMMARY

This Chapter introduced the Context Management Infrastructure (CMI) and its components. The CMI contains a context management system (CMS), which uses a snapshot approach for context data aggregation. The snapshots ensure that context queries are evaluated on a consistent state and supports storing and querying context histories. The CMS stores an RDFS-based context model and is queried by OCQL. The CMI offers different levels of OCQL language integrations with Java-based host languages, with different levels of static type safety and flexibility. The CMI extends the OSGi component platform with a service interception mechanism and enables runtime adaptations of OSGi components on the service level.

14 see Section 4.2.6

Finally, we presented the aspect language CSLogicAJ, which combines context analysis and aspect orientation to a context-aware runtime adaptation language and fulfills all requirements elicited in Section 1.4.

Part III

IMPLEMENTATION AND EVALUATION

6 IMPLEMENTATION

6.1 MODE ANALYSIS

The mode analysis implementation is based on Lu's Logic Program Analysis Engine (LPANE) project [129]. The LPANE mode analysis implementation does not consider variable aliasing in its abstract domain, so we integrated aliasing following the formalism defined in [130]. And we integrated the findall meta-predicate extension described in Section 3.2 by extending LPANE's fix-point operation implementation.

Lu's formalism and implementation only consider a subset of Prolog syntax: normal logic programs (NLP). Negation is only allowed on literals, not on arbitrary formulas and the operators or (";") and if ("->") are not implemented. In OCQL's Prolog subset, negation is also allowed on formulas and the ";" and "->" operator are part of the syntax[1]. To bridge this gap we realized a normalization step which translates these cases to equivalent normal logic program syntax.

We only illustrate the transformation exemplarily for the ";" operator. Consider the following formula:

```
(a(A;B),b(B,C));c(C,D)
```

First all variables V are extracted from the term and a unique predicate signature (n/A) is generated with arity $A = |V|$. Then, two clause are generated for each argument of the conjunction:

```
n(A,B,C,D):-
  a(A;B),b(B,C).
n(A,B,C,D):-
  c(C,D).
```

The transformation of negated formulas is similar. A new clause is generated containing the formula and the formula is replaced by the call of this clause. The "->" operator is transformed based on the definition:

```
(If -> Then; _Else) :- If, !, Then.
(If -> _Then; Else) :- !, Else.
```

The cut is ignored by the mode analysis. Since the result of the `If` literal call is unknown, both cases, success and failure, must be considered. The abstract semantics $F^{\flat}_{P,G}(X^{\flat})$ (see Section 3.1.4) abstracts from all possible execution stacks that lead to a program edge. The evaluation of clauses of the same predicate are therefore always abstracted into one abstract substitution.

6.2 CONTEXT MANAGEMENT SYSTEM

The context-management system is based on SWI-Prolog and its semantic-web library [209]. The semantic-web library provides a parsing framework for turtle and XML rdf syntax and an rdf triple store with optimized hash indexes as well as predicates to evaluate the RDFS entailment.

The context-management system and the query language are designed to be embedded into programming languages. A context-aggregator provides a flexible API for requesting

1 See Section 2.3.4.

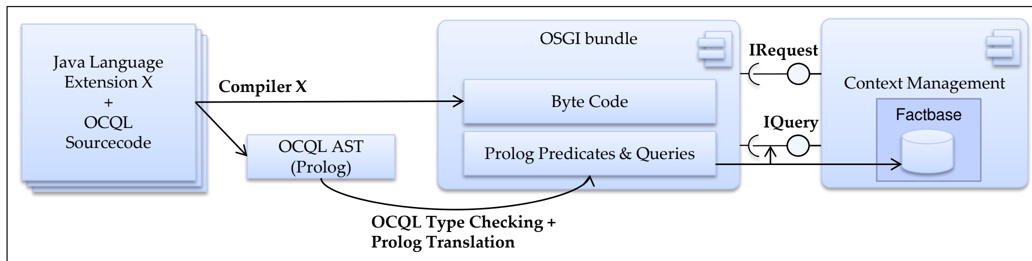


Figure 6.1: Compilation of a Java language extension with embedded OCQL

context sources. The query language is embeddable with other Java language extensions (JLE)[2] with little effort. The query interface to the framework is a textual AST representation of OCQL, which can be generated straightforward from a compiler for JLE + OCQL. The textual description of the AST is represented as nested Prolog terms for which the framework provides a type checker and a Prolog code generator. Figure 6.1 illustrates the compilation steps: A JLE embeds OCQL. Its compiler generates byte code for the core JLE and an OCQL AST in Prolog syntax for the OCQL expressions.

The OCQL AST is type checked, translated to Prolog predicates/queries, and integrated into an OSGi bundle. At runtime the predicates are loaded into the context-management system and the queries are executed from byte code via the IQuery interface. Via the IRequest interface a client describes the necessary context information for the enclosing bundle.

The fact representation for RDF-facts contains a fourth argument referencing its source sensor(s) via bitmasks in an integer value[3]. The mask allows us to keep each triple unique even if several sensors resp. schemas reference it. The sensor bitmask argument allows us to keep just one factbase for all queries, although they may query different sensor subsets at a time. Since there is only one unique triple, pure SLD-resolution can be applied. Otherwise choice points for several definitions would have to be removed, e.g., through a meta-interpreter or by altering the underlying Prolog virtual machine.

6.2.1 *OCQL Parsing Framework*

We specified dynamic and static compilation of OCQL via the ICompile interface in Section 5.1.2. In the implementation we only realized the static variant. The runtime version is mostly an integration of existing components and is left for future work. The main aim of this work is to provide an infrastructure for language extensions. Therefore we provide a compiler for OCQL abstract syntax to Prolog, the extensions must implement the OCQL parser for the concrete syntax.

We coupled the compilation infrastructure with the Eclipse framework, which is also based on the OSGi standard. This allows us to use the same modularization means at compilation time as at runtime. Further, Eclipse supports the creation of OSGi bundles, from which the predicates can be loaded in a standardized way.

To compile files containing pre-parsed OCQL, an Eclipse bundle project must be created that contains a folder `ocql`. The bundle is marked as a cmi bundle in the META-INF/MANIFEST.MF, by adding an entry

```
Bundle-Category: cmi-bundle.
```

An OCQL parser must generate the following predicate specification fact for each predicate:

2 or other languages running on the JVM, e.g., JRuby
3 SWI-Prolog uses the GNU multi precision arithmetic library [79] to support arbitrary large integer values. So the length of the bitmask and thereby the number of sensors is only limited by memory.

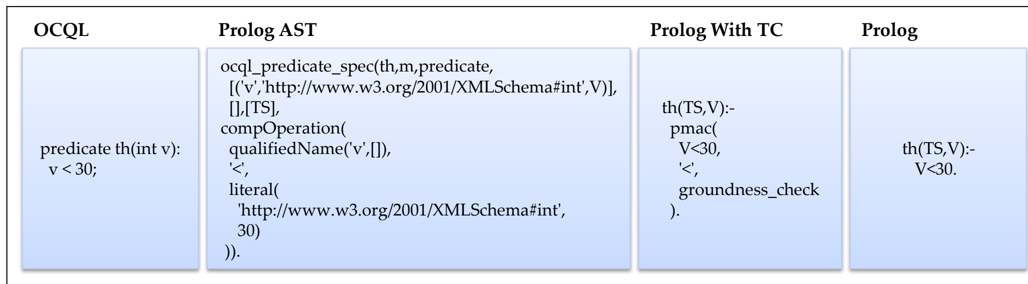


Figure 6.2: Prolog clause generation overview

```
ocql_predicate_spec(
 Name,Module,Kind,TypeParameters,Parameters,
 LocalVars,RdfUrls,PrefixedLVars,EBNF).
```

In the first two arguments the name and the module where the predicate should reside must be given. The other arguments have the following meaning:

Kind gives extensions the means to specialize a predicate. The default kind is `predicate`. Extensions can extend the later described transformation process and thereby read this argument. E.g., the CSLogicAJ aspect introduces the kind `pointcut` and provides a specialized transformation for pointcut predicates.

TypeParameters is the list of type parameters and their bounds.

Parameters is the list of predicate parameters, their types and an associated Prolog variable (*Name*, *Type*, *Var*). The Prolog variable will be necessary for potential extensions, e.g., for CSLogicAJ aspects.

LocalVars is a list of declared local variables following the same schema as the previous argument.

RdfUrls is the list of RDF schema URLs the OCQL code is compiled against.

PrefixedLVars is a list of Prolog variables, which can be prefixed to the final predicate. By convention the first argument is always the variable associate with the associated snapshot's timestamp. Extensions might add arbitrary variables here.

EBNF the term-serialized abstract syntax of OCQL. We will consider this in more detail in the next paragraph.

Let's assume we would like to compile the predicate

```
predicate th(int v):(int tmp):
   tmp = 30 && v < tmp;
```

The parser generates the file in the module `m`:

```
ocql_predicate_spec(th,m,predicate,[],
  [(v,'http://www.w3.org/2001/XMLSchema#int',V)],
  [(v,'http://www.w3.org/2001/XMLSchema#int',Tmp)]),
  [], % No RdfUrls needed
  [Timestamp],
  <EBNF>).
```

EBNF Abstract Syntax

The term-serialization of the EBNF maps the (non-)terminals to Prolog terms. The name of the non-terminals is used as a term name. All sub-expressions are arguments of the non-terminal. The repeated sub-expression (in a * expression) are comma-separated and embraced by brackets. For example, the qualified name `c.name` is serialized to

```
qualifiedName('c',['name'])
```

Terminals are differentiated in their usage context into two different symbol kinds: *separators* and *terminals*. The separators (e.g., comma) are left out in the serialization. The terminals (e.g., the arithmetic operators "+" and "-") are added to the term.

The Figure 6.2 illustrates the mapping of the predicate named `th`, which checks if the argument `v` reached a threshold of 30. In the second column, we see the resulting predicate specification. The OCQL compiler takes the term as an input and applies the pre-mode analysis-type checking from Section 4.3.2 and the transformation to Prolog from Section 4.3.3 in one traversal step. Additionally it wraps all terms, which must be type or mode checked after the mode analysis, with the term `pmac/3`[4].

In the example the variable `V` must be ground, otherwise the comparison would lead to a runtime error. Therefore it is wrapped with the `pmac/3` term, which is tested for groundness after the mode analysis has been applied. The last box shows the final Prolog code for the predicate. The predicate is added to the Prolog module `m` and can access all other predicates in `m`. The module is added to/updated in the folder OCQL in the Eclipse bundle project.

Extension

The transformation and the type checking steps can be extended by application frameworks or language extensions by adding additional clauses to transformation or type checking checks. The transformation is realized by the multifile[5] predicate

```
translate_term(+TermToNormalize,-TranslatedTerm,-SubGoalsList,+Scope)
```

The predicate takes an arbitrary term and the current scope as inputs and bind the translated term and optionally sub-goal specifications to the second and third argument. The next section will show how this extension is applied to add a predicate to OCQL.

6.3 CSLOGICAJ

CSLogicAJ is realized on-top of the CMI. We implemented a *compiler* which generates extended OCQL syntax and a *runtime engine* that intercepts method calls and maintains a runtime state of the bundles and services. The Ditrios framework maintains a runtime model of all registered services, their LDAP properties and all service requests[6] in the CMS Prolog engine. It associated each registered service with an unique identifier, allowing Prolog to refer to the service. For a detailed description of Ditrios see [180].

6.3.1 *Compiler*

The CSLogicAJ compiler uses the OCQL parsing framework to generate Prolog code from OCQL AST. To support pointcuts and advices it defines two new OCQL predicate specifications kinds:

4 Post Mode Analysis Check
5 The Prolog directive *multifile* declares that clauses of a predicate can be defined in different files.
6 Provides as context class ContextRequest with the property *objectclass* and *ldap*, where the later is the complete LDAP query.

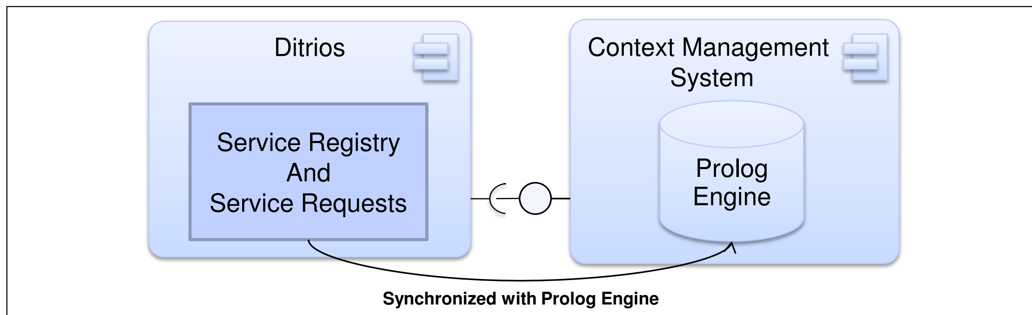


Figure 6.3: Ditrios service runtime model

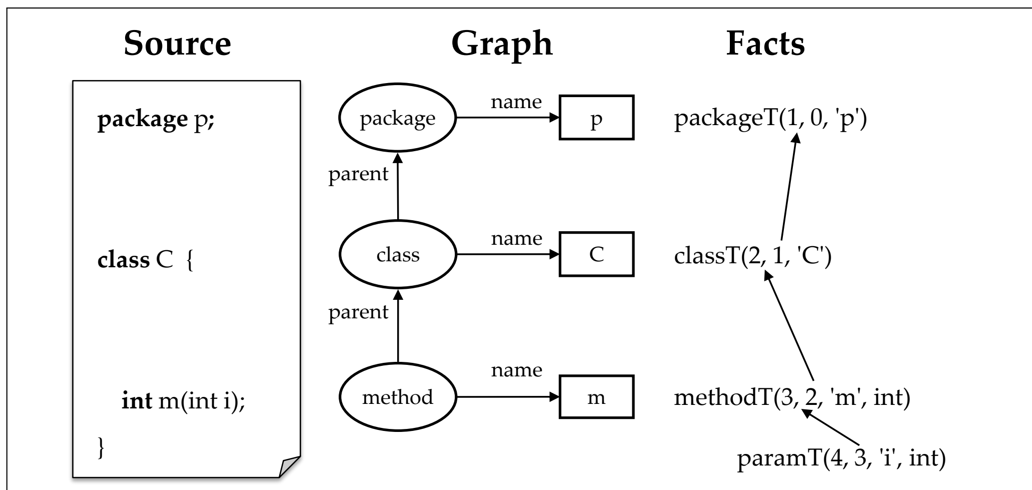


Figure 6.4: JTransformer Prolog AST

`pointcut` and

`advice(before|after|around|onchange)`

They represent the pointcut expression of pointcut and advice declarations. Additionally to the `Timestamp` variable, the aspect constructs prefix the Prolog variable `Jp` to all predicates. `Jp` represents the join points bound by the pointcut expression. Next to the generated context model, the predicates operate on a runtime model of the registered OSGi services. For this purpose the abstract syntax tree (AST) of the interface of each registered service is represented by so called Program Elements Facts (PEF) in the Prolog engine of the CMS. The AST representation was adapted from the Java static analysis and transformation framework JTransformer [190, 122]. For a detailed specification of the PEF representation see [171]. Figure 6.4 gives an example how a class is represented in Prolog. For each AST element a fact[7] is generated. The first argument is always the unique identifier of the element. The other arguments are either references to other nodes or atoms representing names or numbers. We only show exemplarily how the transformation is realized, for details see [180].

Consider the pointcut eventFired pointcut from the LastFMAspect introduced in Figure 5.25:

```
pointcut eventFired(ServiceEvent event) :
    call( * EventManager.serviceEvent(..) )&&
    args(event);
```

7 For brevity we only list PEFs with a reduced number or arguments here.

The pointcut is transformed into a Prolog predicate evaluated on the PEF representation:

```
eventFired(Timestamp, Jp, Formal_event) :-
  fully_qualified_named(Class,'org.cs3.ditrios.facade.EventManager'),
  methodT(Jp,Class,firedServiceEvent,Type,Exceptions,Params),
  match_args(Params,[ ('org.cs3.ditrios.facade.ServiceEvent',Formal_event)]).
```

Pointcut expressions are purely evaluated in Prolog. Since the service pointcut and the state-based pointcuts `target`, `this` and `args` refer to runtime objects, CSLogicAJ must have a mapping between the variable bindings in Prolog and the Java objects. For the state-based pointcuts this mapping is realized by naming patterns. A variable bound to the following atoms results in the binding of advice/pointcut arguments:

target: target service

this: the calling BundleContext

arg:*n*: the *n*-th argument of the call

service:*id*: the service associated with the identifier *id*

6.3.2 *Extensible OCQL/Pointcut Parser*

The compiler is based on the parser of the AspectJ 1.0.6 compiler. It was extended to parse OCQL as part of the pointcut expressions and generate an OCQL Prolog AST. To allow for extensions, the concrete syntax of the OCQL parser implementation is not fixed. It can be easily adapted by modifying a Prolog predicate. The language provides a number of predefined non-terminals: `identifier`, `formal_or_identifier`, `type`, `int`, `float`, `string`, `this`, `pcd`. Further non-terminals are defined in the predicate:

```
non_terminal(Name:atom,Symbols:list,inline:boolean)
```

The `inlined` argument is true if all parsed elements are inlined, and false if the parser should wrap the elements with a term which functor is the non-terminal's name. The production body is a list of non-terminals and EBNF operators. The following operators are defined:

optional expression: opt(EXPR); in case the expr is missing the token '$$' is added to the list of arguments

optional and remove: opt_remove(EXPR); in case the expr is missing no argument is added to the term

alternatives: or(EXPR_1,...)

repetition: star(EXPR_1,...)

terminals: terminal(ATOMIC); the token ATOMIC is parsed and added to the token list

ignored non-terminals: separator(ATOMIC); the token ATOMIC is parsed, but not added to the token list

For example, consider the non-terminal PropertyAccess that represents the access of a property:

```
non_terminal(propertyAccess, [ separator('.'), identifier ], false).
```

The expression *.propertyName* is parsed via this non-terminal to the term `propertyAccess(`*propertyName*`)`.

6.3.3 *Static Analysis*

The compiler framework offers two different means to apply static analysis. First, pre-mode-analysis type checks are possible by the predicate

```
type_check(+Scope,+LhsType,+RhsType,+OnlyRightToLeftAssignment)
```

which either checks the assignment from right-to-left or, if the last argument equals 'fail', it checks that at least one direction is correct ($\lesseqgtr$).

For post-mode-analysis checks the term `wrap_check/4` wraps a term in the resulting Prolog code to be marked for later checks:

```
wrap_check(+CTName,+Check,+Term,-Wrapped)
```

The `Check` attribute may take three different terms as an argument, which are evaluated in the final type checking phase[8]:

assign(LhsType,Operator,RhsType): assignment and findall

call(ArgumentTypes,Parameters): calls

allVarsGround(Vars): arithmetic operations, checks of the first n-i arguments of mapped predicate calls.

Post mode-analysis is applied by the extensible multifile predicate:

```
process_ocql_post_check(+Check,+PtExit,+PtFrom,+MC,+NodeId)
```

The `NodeId` argument can be used to generate error messages. The MC argument contains the modes before and after the transition between the program pointcut PTExit and PTFrom, e.g.:

```
MC=mc([B/2, C/1, D/1],[B/2, C/2, D/2])
```

Using the `history` predicate as an example, we show how the extension of the syntax can be realized. At first, we define the syntax of the predicate:

```
non_terminal(historyExpr,
  [separator('history'),
   separator('('),formal,separator(','),booleanExpr,separator(')')],
   false).
```

The argument starts with history, then an advice argument follows, then a comma and a boolean expression. In the next step the translation and pre-mode-type checking is applied:

```
translate_term(
  historyExpr(formal(_Name,NewTS),Expr),
  (cms_timestamp(NewTS),Normalized),
    SubGoals,
    scope(Module,Predicate,[_TS|UntypedVars],Vars)) :-
  % Retrieve Type of Formal:
  resolve_varname_in_scope(Vars,NewTS,Name,FormalType),
  % Check that long is assignable to the formal type of the time stamp variable NewTS:
  type_check(CTName,FormalType,'http://www.w3.org/2001/XMLSchema#long',fail),
  translate_sub_term(Expr,Normalized,SubGoals,
      scope(Module,Predicate,[NewTS|UntypedVars],Vars)).
```

The mode of the predicates is not relevant in this case. We exemplary show how post-mode-analysis checks are prepared and evaluated. Consider the unification of two typed variables A=B. The `translate_term` uses wrap_check to prepare checks that are run after the mode analysis:

8 For example for the delayed typing rules defined in Figure4.15.

```
public interface DitriosInterceptor {
  Object invokeAdvice(
      InterceptionProcessor processor,
      DitriosFacade ditriosFacade,
      Object targetService,
      Object[] args)
  ...
}
public interface InterceptionProcessor {
  public Object proceed(Object target, Object[] args);
}
```

Figure 6.5: Ditrios interceptor interface

```
wrap_check(Module:Predicate,assign(LhsType,Operator,RhsType),Normalized,Wrapped)
```

After the mode check has been applied the post check is evaluated for all prepared checks. Below is the clause which tests ground variable assignments defined in the rule *unify-g* in Figure 4.15:

```
process_ocql_post_check(assign(_LhsT,'=',_RhsT),PtExit,_PtFrom,MC,_NodeID):-
  % Retrieve the left and right hand side expression from the program point:
  build_in_predicate(PtExit,Lhs=Rhs),

  % Test that both left and right-hand side are ground variables or atoms:
  ground_variable(Rhs,MC),
  ground_variable(Lhs,MC),
  !.
```

6.3.4 *Mapping Advice Constructs to Java Source Code*

For each advice construct a Java class is generated, which is an instance of the `DitriosInterceptor` interface that is defined in Figure 6.5. The interceptor is called with the `DitriosFacade`[9], the target service, the call's arguments and an `InterceptionProcessor`. The interception processor contains a `proceed` method to realize call forwarding to the next advice or the original join point.

The advice body operates on a generic graph of IContext-typed instances instead of the statically typed context classes. An alternative would be the generation of Java classes for each aspect bundle representing the RDF classes imported by the aspect. But, these classes would only be valid in the context of the aspect, potentially leading to the redundant creation of a context instance for different aspects.

For this reason only XML primitive types (see Section A.1) have a static mapping to Java types in CSLogicAJ. Method calls, type casts and type checks are realized by reflection. The IContext interface, defined in Figure 5.10, offers the methods getProperty(uri), castTo(uri), and isSubtypeOf(uri), which realize property access, type cast and runtime type check (instanceof).

Since a property can be of type IContext or a primitive type, the return type of getProperty() is Object[]. If an array containing context classes is handed over to regular Java code, the object array must be converted, since Java arrays have a fixed component type, which is not automatically converted on casts. The array conversion is realized with by the class `TypeUtils` which defines the generic method

```
static <T> T[] copyOf(Object[] original, Class<? extends T[]> newType)
```

9 see Figure 5.14

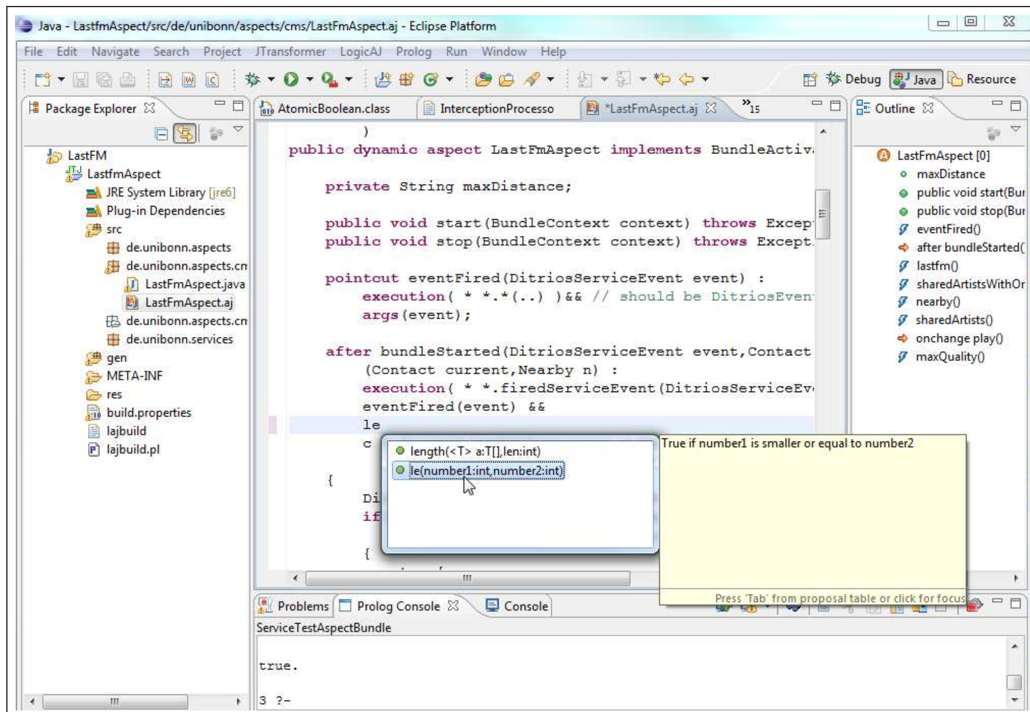


Figure 6.6: CSLogicAJ's integrated development environment

It converts the object array `original` to an array with component type `newType`. The example below illustrates the mapping. Consider that the Java class `LastFMPlayer` plays the URL of a last.fm artist. In CSLogicAJ the code retrieves the first element of the bound artist array, retries the URL property array and looks up the first element:

```
LastFMPlayer.play(artists[0].getUrl()[0]);
```

After OCQL type checking, the generated Java code of the artist array is typed IContext[]. The call of getProperty on the artist object retrieves as an object array, which is converted via TypeUtils to a string array. Finally, we access the first element of the list and pass it to the `play` method of `LastFMPlayer` class:

```
LastFMPlayer.play(TypeUtils.copyOf(artists[0].getProperty(
          "http://ws.audioscrobbler.com/2.0/url"),String[].class)[0]);
```

6.3.5 *Integrated Development Environment*

We provide an integrated development environment for CSLogicAJ based on Eclipse [83], JTransformer [190, 122] and the Prolog Development Tools [172]. It contains an editor with syntax highlighting, predicate/pointcut completion, error/warning markers and an outline, see Figure 6.6. On every save operation the aspects are compiled and on success an aspect bundle is generated, which contains the compiled Prolog code and advice classes. The Eclipse-integrated OSGi launch configuration can be facilitated to start aspects and base components of an application. The IDE can be installed into Eclipse 3.7 via the LogicAJ update site[10].

10 http://sewiki.iai.uni-bonn.de/research/logicaj/installation

6.3.6 *Realization of Query Context Sources*

The connection to the Web service and the mapping to RDF is in the responsibility of the ContextSource. The queries on external context sources are executed via the bidirectional Prolog-Java bridge *JPL*, which is part of SWI-Prolog. The Prolog queries access the remote context services via a Ditrios interface. Since accessing remote data is a time consuming operation, queries are cached by the Prolog engine CMI. All data belonging to the same query and containing the same input parameters is cached until the expiration time is reached. The returned triples are added to the factbase and can be queried by later expressions, following logic update semantics[11] [38]. The retrieved RDF triples are only visible to the querying module although they share the same cache.

6.4 SUMMARY

The CMI and the OCQL compiler rely on a number of projects and frameworks. The mode analysis was implemented based on the LPANE project [129]. The Context Management Infrastructure facilitates the OSGi architecture to expose its API and to encapsulate context queries in OSGi components. The CMS delegates RDF parsing and the query evaluation to the semantic web library of SWI-Prolog. The CSLogicAJ compiler is separated into a Java parser, based on the AspectJ compiler 1.0.6. The type checker and the Prolog code generator are implemented in Prolog. The compiler was implemented with extensibility in mind. Via Prolog predicates the concrete syntax and the type checker can be extended to form new OCQL predicates.

And finally, we provide an IDE for CSLogicAJ, based on the Eclipse framework. It contains a full-fledged editor and an aspect compiler, which generates aspect bundles deployable on OSGi.

---

11 also called *deferred* update semantics, the ISO-Prolog standard update semantics.

7 EVALUATION

This Chapter evaluates the application of the Context Management Infrastructure and the aspect language CSLogicAJ. First, we summarize two publications. The first (Mügge et al. [143]) describes how dynamic adaptation of mobile applications can be achieved with a statically typed object-oriented language and how dynamic context-aware aspect-oriented programming can improve the realization, concerning adaptation anticipation and code quality.

Section 7.2 is a summary of Rho et al. [175] and describes the integration of the CMI framework with the Java language extension JCop [15], a context-oriented programming approach for Java. Here we evaluated the most dynamic variant of OCQL language integration (see Section 5.1.4). We close the Chapter with the results of a micro benchmark on service method calls 7.3 and discuss the consequences for future work.

7.1 PROGRAMMING FOR CONTEXT-BASED ADAPTABILITY - A CASE STUDY

We will first introduce and analyze the requirements for a dynamic mobile adaptation scenario. In our scenario a business user visits a trade fair. He keeps a lot of documents on his mobile device and manages them by different applications, as illustrated in Figure 7.1.

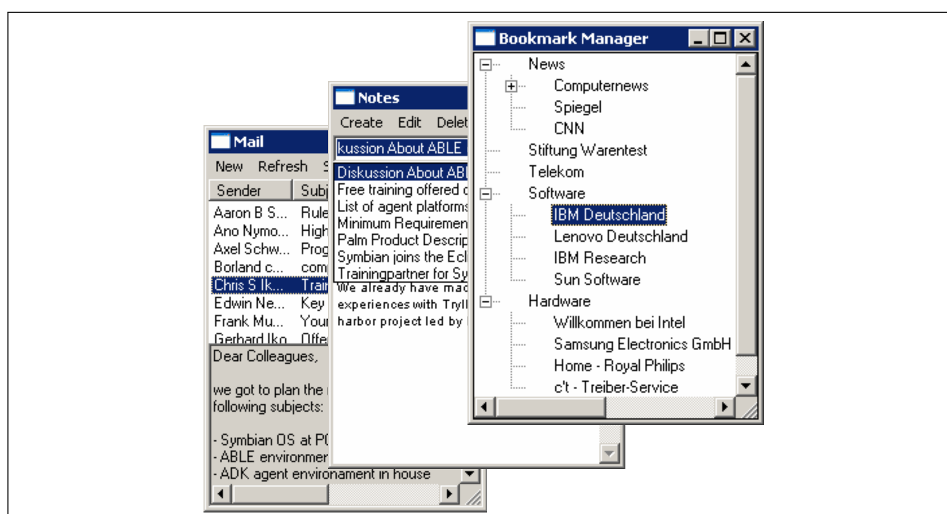


Figure 7.1: Simple document management tools for mail, minutes and bookmarks

When he enters the fair and is about to be engaged in meetings and negotiations, his device recognizes the new situation automatically and scans for adequate adaptations. The fair organization offers some special services for document management support: a *document indexing service*, which creates an index for searching and classifying documents; a *vicinity explorer* calculating the fair stands closest to your current location; an *index matcher*, which determines how similar two index lists are; a *sorter*, a *filter*, and a *tree flattener* for list data manipulation. These services can be combined to offer a real benefit for the user while he is on the fair, as shown in Figure 7.2.

These adaptations work together in the following way: Initially, the indexing service creates a (potentially weighted) index characterizing the content of the document for each locally stored user document. Second, the vicinity explorer calculates the distance

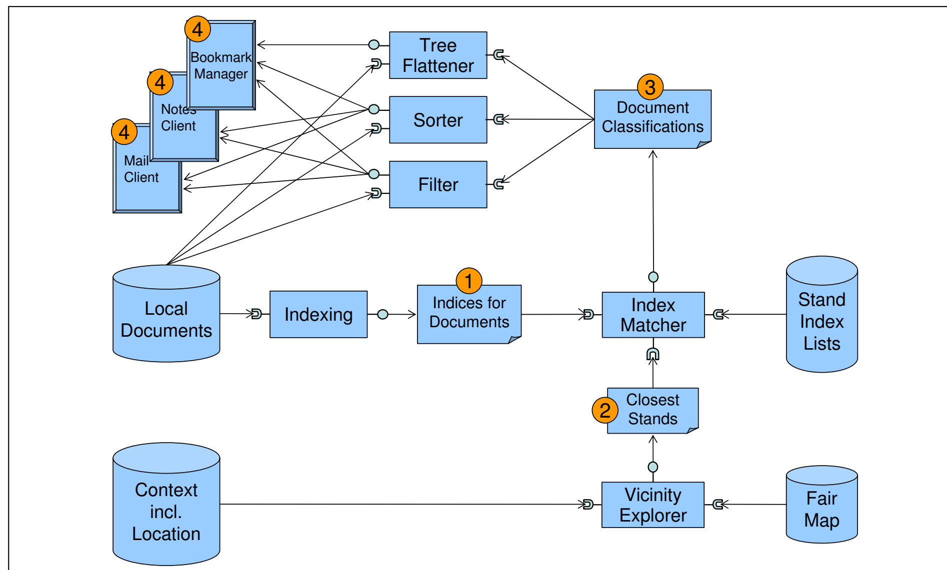


Figure 7.2: Combining services leads to a beneficial adaptation

between the user and all stands on the fair (given by the Fair Map) and determines which stands are close to the user's current position. In the third step the document set gets ordered with respect to their relevance for each of the stands in the user's vicinity. Therefore, each exhibitor provides an index list describing his company (Stand Descriptions). The index matching service estimates the relevance of each document to a stand by comparing both index lists and thus produces a document classification. Finally in step four the relevance classification for the documents are used to display them in a more convenient way. Therefore three services can be used for sorting, filtering and flattening document entries in list- and tree-like structures. Figure 7.3 illustrates the adapted client tools, providing prompt access to those documents relevant for the closest stands. We will call the adapted applications *PimPro*[1].

7.1.1 *Requirements Elicitation*

What requirements can be elicitated from the given scenario? First, the adaptation should be done at runtime, since the user will probably have his tools already started before he enters the fair. Second, the situation should be recognized automatically, since the user will not manually specify each situation change without knowledge about a possible benefit through adaptation. Third, the services popping up at the fair, should be detected automatically, since a busy user won't be able to scan manually for new services every now and then. Fourth, we assume that there will be a large cloud of services offered. The user needs support for discovering appropriate services. This holds for services that are separately useful (e.g., the vicinity explorer service might be suitable to show the closest stands on a map, or simply in a list) when it is tedious to find the service in a long list, but it becomes definitely necessary when it comes to combinations of services as shown in this example. Fifth, the adoption of new services must be tightly integrated into the functionality of the user's applications. In our example service adaptations are spread across three different client applications, and more complex scenarios are easy to think of, i.e., adaptation should allow for cross-cutting changes.

1 Personal Information Manager Pro

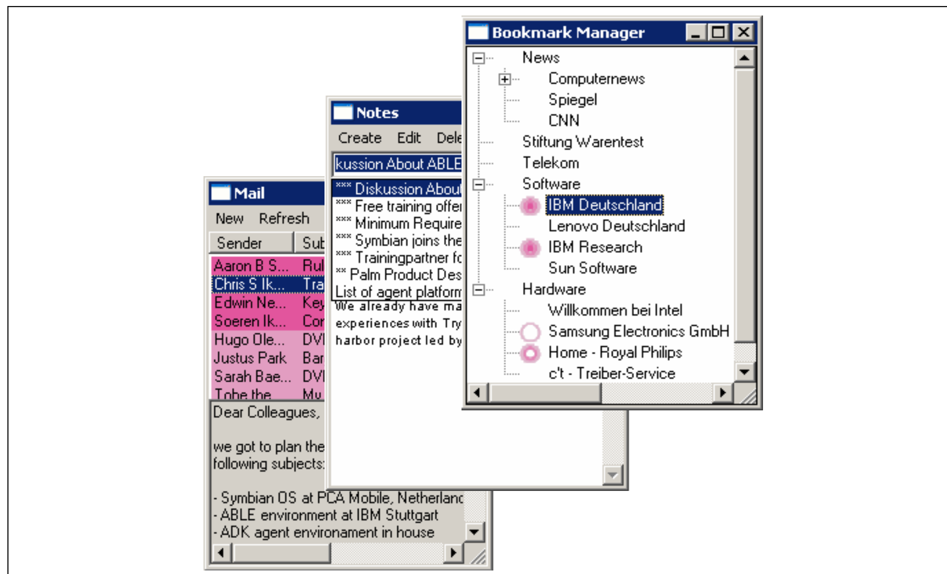


Figure 7.3: Adapted tools provide prompt access to currently relevant documents.

7.1.2 *General Requirements for Context-Sensitive Adaptivity*

Most of the requirements deduced from the scenario in the last Sections, can be abstracted to a general form and occurs frequently in context-sensitive settings. Additionally we found some basic technical requirements regarding the tight integration of the adaptation into the existing applications. I.e., we need to care for replacing existing functionality with more appropriate alternatives, we must allow for extending the set of given functionality by new elements, and lastly we also need to be able to remove functionality, which has been added before. The following list summarizes the requirements and gives a first short comment about how we are going to accomplish each of them. Thus, we tackle

- *replace existing functionality* by enhancing the strategy pattern
- *enhance given functionality* by enhancing the decorator pattern
- *add new functionality* by enhancing the visitor pattern
- *adaptation at runtime* by enhancing design patterns, using SOA, and applying run-time aspect weaving
- *automatic service detection* using a service-oriented architecture
- *cross-cutting adaptations* by aspect quantification
- *minimizing anticipation* by applying aspects and thus introducing details about the variation points not before runtime

These requirements further support the more general requirements 10. - 12. gathered in Section 1.4 that are related to framework-level adaptation support.

7.1.3 *Pure Object-Orientation - Patterns for Adaptivity*

We start with pure object-oriented methods for software adaptability. Design patterns are a well-known means for introducing flexibility to software (c.f. [84]). For the field of product

line engineering Svahnberg discusses in [195] several patterns for introducing systematic variability. Hence, we investigated what could be achieved applying appropriate patterns.

In general, design patterns mostly address statical flexibility, e.g., adaptivity during the software evolution process. Although this is fundamentally different to our setting, were adaptations generally occur at runtime, some selected patterns seem to be a good starting point. In particular we applied the strategy pattern to exchange functionality and the decorator pattern to enhance functionality at runtime.

Originally, the strategy pattern provides an infrastructure for dynamically exchanging a certain functionality at runtime. While exchanging strategies (and thus functionality) at runtime is supported by the pattern, two problems remain: we need to be able to detect strategies and load them at runtime. The latter is relatively easy to achieve by extending the strategy pattern with a dynamic strategy repository, which is able to load new strategies at runtime and offer them to the business logic of the application. Detection of strategies or even discovering appropriate strategies remains a complex problem in its own right, calling for abstractions on the architecture level of the software.

The case for enhancing functionality by decorators is even more complicated. A decorator wraps a given object so that a call to its methods will first be executed by the decorator, potentially specifying additional behavior and then by the original object. The decorator could also specify to enhance the behavior after the original object's functionality has been executed or even replace it completely by a new functionality (which would have the same effect as a strategy).

At least when multiple decorators come into play, the client who configures the decoration of objects, needs detailed knowledge about possible or reasonable combinations. This is feasible in a static setting, since flexibility then means to have the option of convenient re-specification of decorator combinations at development-time. In our setting, we need a dynamic decorator in the sense that we can robustly add or remove decorators to objects at runtime, without explicitly taking care of permitted combinations.

We achieved this by extending the decorator pattern with a configuration logic that automatically cares for reconfiguration of the objects decorations when it is changed. This means basically, that the combination of decorator is self-managed by an "intelligent" decorator manager.

Furthermore we used a variant of the adapter pattern to allow for flexible connections between not quite fitting interfaces and the observer pattern for dynamically recombining functional units.

While design patterns can provide for basic adaptivity, relying on them as the only means cannot reasonably cope with the requirements of context-sensitive adaptivity. For example, cross-cutting concerns will lead to a proliferation of similar structures within the whole software and introduce a high level of maintenance complexity. Detection of available adaptations and discovery of appropriate adaptations will lead to very specific implementations with a high level of complexity. Hence, using separate abstractions for coping with these issues seems appropriate, as we will describe by using services in Section 7.1.4. The general problem of aiming for least anticipation cannot be reached solely by applying patterns, too. Hence, we use Aspect-Orientation as discussed in Section 7.1.5.

7.1.4 *SOA and Object-Orientation - Patterns for Adaptivity*

Service-oriented Architecture Supports Adaptivity

Service-orientation simplifies dynamic adaptation because components are low coupled. The use of components, such as the viewer categorization[2] or the tree decoration, must be completely anticipated in a pure OO solution. A configuration class is necessary to

2 We assume that the applications we adapt support categorization in their views. This allows us to apply document classification in the application.

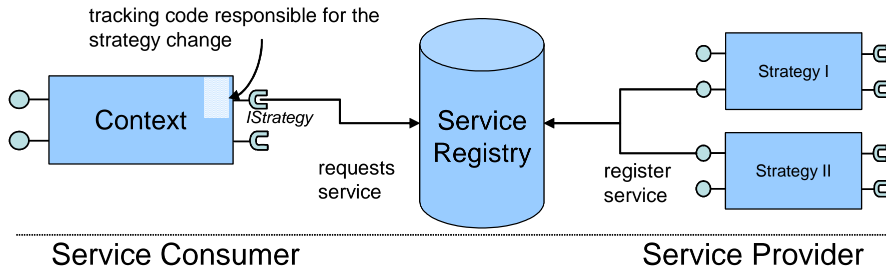


Figure 7.4: Strategy Pattern in a service-oriented architecture

organize the dynamic change of decorations and categorizations. In a service-oriented approach a *service registry* is responsible for organizing the available services. *Service Providers* register services and *Service Consumers* request services. Figure 7.4 depicts how this architecture can be used to implement a Strategy Pattern. Consider the components *Context*, *Strategy I* and *Strategy II*. The *Context* component requests a service *IStrategy*, which is implemented by services registered by the component *Strategy I* and *Strategy II*. We call code that is responsible to select one or several strategies based on the availability and properties *tracking code*.

Service-oriented Strategy

The Strategy Pattern can be applied when a functional part of an application should be dynamically replaceable by an equivalent or different part having the same interface. If no implementation is available a default implementation or a *null object* can be used to avoid null or stale references in the *Context* object that is using the strategy. We used the strategy pattern in the *PimPro* application for adaptation of Notes, Mail and Bookmarks categorization. All three applications were implemented as bundles with a service request for an IDocumentCategorization service. Therefore service tracking code was introduced to all bundles to react to changes concerning the availability of services. The user is responsible for activating and deactivating bundles containing categorization services. Only if no service is available the default service is activated that associates all documents with the same (default) category. Figure 7.5 illustrates this example for the PimPro Notes, Mail and Bookmarks bundles. Here, the user may select a gradual, vicinity or no categorization for his mails, notes or bookmarks. For the default categorization no selection is necessary.

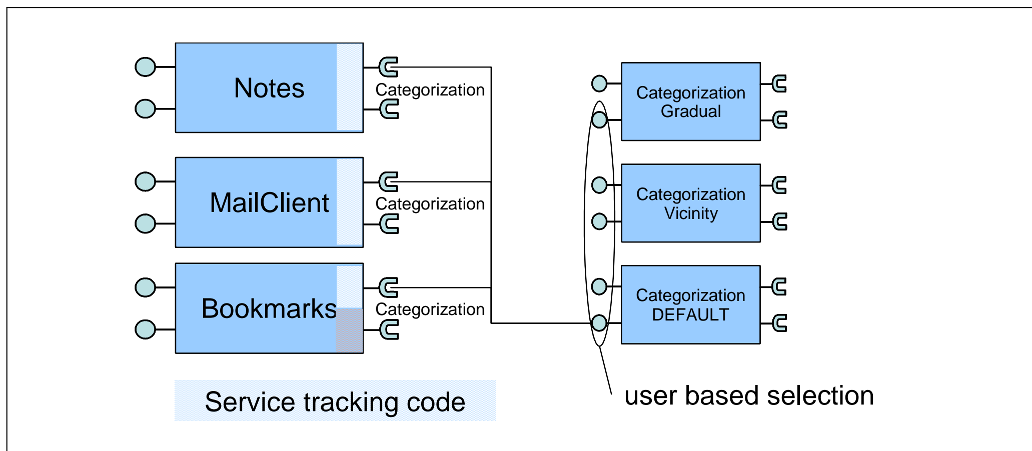


Figure 7.5: Usage of the Strategy Pattern in PimPro for the view categorization

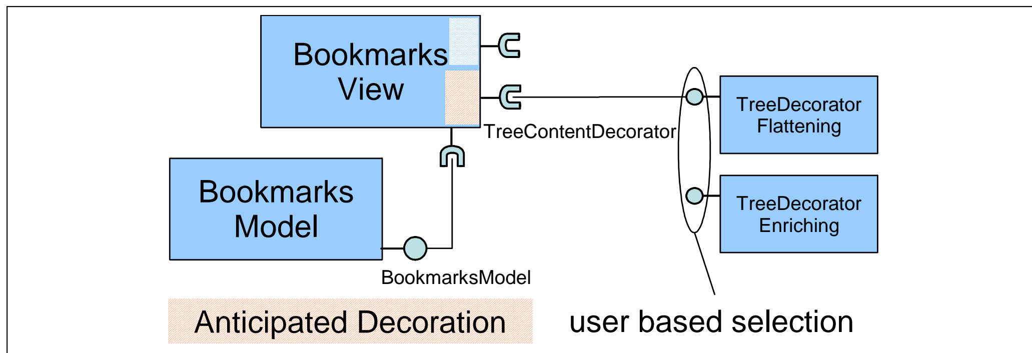


Figure 7.6: The anticipated decorator pattern for the bookmark tree decoration

Service-oriented Decorator

The dynamic tree decoration also takes advantage of SOA. Here, the configuration of the bookmark tree decorator is realized via services. All available tree decorators[3] are tracked and added to the dynamic decorator implementation in the *Bookmarks View* component. The general concept of a decorator pattern as described in Section 7.1.3 is preserved.

Limitations

Several downsides remain. Concerning the strategy solution the necessity of tracking code itself is problematic. The selection criteria of the concrete service must be anticipated in the tracking implementation. For a context dependent selection the bundles must be context-aware. This is a strong restriction since a bundle should be usable in arbitrary settings.

The main limitation of the decorator solution is the anticipation in the *Bookmarks View* component. Every service request that should be decorated must be extended with the relatively complex dynamic decorator pattern structure.

For both patterns an inherent OSGi problem, the uncertainty about the service reference, must be taken into account in the concrete implementation. While using the reference and passing it to the internals of the component, the service may become unavailable. This problem must be anticipated by the programmer. There is no means in OSGi to replace a service reference on the platform level.

7.1.5 *SOA and Aspect-Orientation - Patterns for Adaptivity*

Hannemann et al. [94] and we [173] have shown that AOP, and especially generic aspect languages, simplify and improve the implementation of most of the GOF design patterns [84]. Here, we show how aspect-oriented techniques combined with a context-aware service-oriented architecture can even go further. With the help of CSLogicAJ aspects[4] we are able to remove most of the tracking code - and thereby the anticipation of possible contexts - from the components.

Service aspects enable replacement or decoration of services and can refer to arbitrary context information. By this, service tracking can be moved to the architecture level making dynamic changes of services transparent for the application. The same is true for the dynamic decorator preparation described in Section 7.1.3. Bundles are only

3 implementing the interface TreeContentDecorator
4 The case study applied an early version of CSLogicAJ, where the context pointcut language was based on untyped Prolog predicates instead of OCQL. For this thesis the aspects were translated into latest CSLogicAJ syntax.

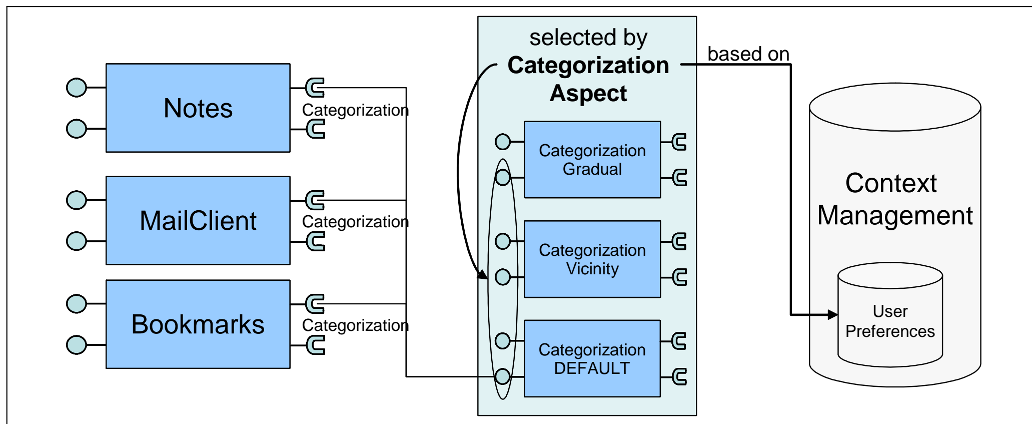


Figure 7.7: Aspect-oriented Strategy Pattern based on user preferences

responsible for specifying their service dependencies via service requests. The service aspects organize the reconfiguration of the composition.

Section 7.1.5 and 7.1.5 illustrate how the language enables dynamic strategy change and decoration.

Dynamic Strategy Change

This example realizes the dynamic change of a categorization service based on user preferences. Let's assume that a context provider is available, which represents the user preferences on a certain device. Since the context management is not in the focus of this case study, we skip the definition and registration of the provider.

Once the context provider is registered, a context class `UserPreference` is available via the context management system. Based on this class it is now possible to define a dynamic strategy change via CSLogicAJ aspects. Figure 7.7 depicts how the `CategorizationAspect` selects one of the available categorization types based on the user preferences. If the preferences or the set of available services change, the aspect selects a different categorization service. If no service is available the default (DEFAULT) service is selected.

The onchange advice `strategyChange` in Figure 7.8 facilitates the *service* pointcut[5] and the predefined `ServiceRequest` context class. The *service* pointcut binds registered services and their attributes, the ServiceRequest context captures all active service requests. The advice is re-executed every time the aspect is activated or one of the advice parameters `service` and `kind` changes. The advice accesses the Ditrios facade via the built-in field `thisJoinPoint`.

The `setActiveService`(requestId, Object) method performs the actual strategy change. Each request for services with interface `IDocumentCategorization` is set to the service bound to the variable *service*. The pointcut *backtracks* over all possible request bindings and the advice is executed for each binding separately. Here the crosscutting application of the aspect on all bundles becomes evident. This change is transparent for the using bundles. They keep their references to requested services.

Dynamic Service Decoration

As an example for service decoration we only consider the TreeFlatteningAspect. In the PimPro example this aspect is responsible for flattening the model of the bookmarks model. This is realized by around advices on all TreeContentProvider services, such as the model of the Bookmarks applications (see Figure 7.9). Every method call to the Bookmarks model is intercepted and modified to flatten the tree structure.

5 see Section 5.4.2

```
aspect CategorizationAspect {
    onchange strategyChange(IDocumentCategorization service,
               String kind, ServiceRequest request)
       changed(ne(service) || ne(kind)):
     UserPreference->one(categorizationKind = kind) &&
     service(service, { categorizationKind = kind } ) &&
     request=ServiceRequest->one(
             objectclass="org.cs3.csi.IDocumentCategorization")
    {
       try {
           thisJoinPoint.getDitriosFacade().
              setActiveService(request.id, service);
       } catch(Exception ex) {
           throw new RuntimeException(ex);
       }
    }
```

Figure 7.8: Dynamic strategy change

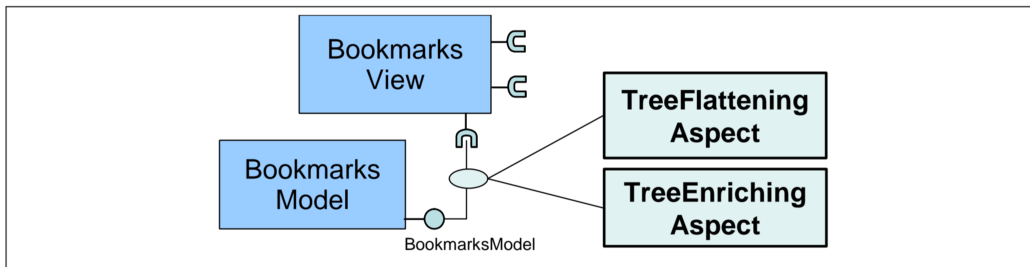


Figure 7.9: Aspect-oriented decorator pattern for the bookmark tree decoration

The aspect implementation is shown in Figure 7.10. We concentrated on the most relevant advice that intercepts the `getElements`(Object) method. First the tree content provider object and its arguments are bound in the pointcut expression.

The helper method `getAllElementsRecursively`[6] retrieves all elements of the provider and returns a list. The list is converted to an array and returned as the flattened element list. The proceed call represents the delegation to the original service method or the next advice that was woven, e.g., by another aspect that decorates the tree.

If more than one decorator aspect is woven the order of weaving is important. For example, an aspect that adds new bookmarks to the tree must be executed before the flattening takes place. Otherwise the result is a mixture of flattened original bookmarks and a new tree of added bookmarks.

A first solution to this problem is explicitly annotating the dependencies between the aspects.

Limitations

The implementation of bundles becomes much easier with AOP but in concrete adaptation scenarios anticipation is still needed in the bundles. The strategy change in Section 7.1.5 is only possible if the service change is possible at any time. The transaction concept presented in Section 5.3.3 overcomes this problem, but results in the necessity to define transaction start and end points in the bundle.

6 This method was implemented straight forward. We left it out only for brevity.

```
aspect TreeFlatteningAspect {
    Object[] around flattenElementTree(
          ITreeContentProvider provider, Object inputElement) :
        execution(* ITreeContentProvider.getElements(Object)) &&
        target(provider) &&
        args(inputElement)
    {
         List result = getAllElementsRecursively(provider,
                           proceed(provider,inputElement));
         return result.toArray();
    }
     // ... around advices that intercept the other
     //     methods of ITreeContentProvider and return
     //     null or false for the other methods
  }
```

Figure 7.10: Bookmarks Flattening Aspect

7.1.6 *Summary*

The case study addressed the question how adaptive software for context-sensitive scenarios could be developed based on a statically typed language like Java or C++. We first outlined a scenario and deduced the most crucial requirements this kind of software faces. Solely relying on object-oriented abstractions cannot reasonably cope with the problems of not fully anticipated runtime adaptation, detecting, discovering and integrating adaptations. These requirements call for more sophisticated abstractions, as provided by a Service-Oriented Architecture. As we illustrated rich adaptations frequently make cross-cutting changes necessary. Aspects can cover this issue to a large extend. They also address the general issue of reducing the level of needed anticipation.

7.2 JCOP QUERY LIBRARY

This Section gives an example of the most dynamic use of the CMI, the first variant of language integration presented in Section 5.1.4. The CMI was integrated with the context-oriented programming language JCop [15, 14]. We developed a library for JCop as an interface to the query system. JCop is a language extension to Java implementing the context-oriented programming approach (Hirschfeld et al. [106]). It provides first-class layers (modules that encapsulate variations of methods) and explicit, implicit, and declarative constructs to control layer activation at run time (controlling which method variation a call is being dispatched to). Offering these constructs, JCop supports means to modularize and control context-depended functionality. Context queries, however, were not explicitly supported in the language core.

Figure 7.11, illustrates the modularization technique with a generic example. A set of base classes is extended by three layers α, β, γ, which redefine different methods in base. At runtime, an arbitrary number of layers can be activated/deactivated in a certain program flow, adding new functionality to the base program.

7.2.1 *Overview*

Our JCop query library supports executing context queries and defining actions - for example, layer activation - to be taken on context change. In the following, we briefly describe the most important API objects and methods[7].

[7] The classes `ContextQuery` and `IContextHandler` belong to the library package `jcop.query`; `Layer` refers to `jcop.lang.Layer`.

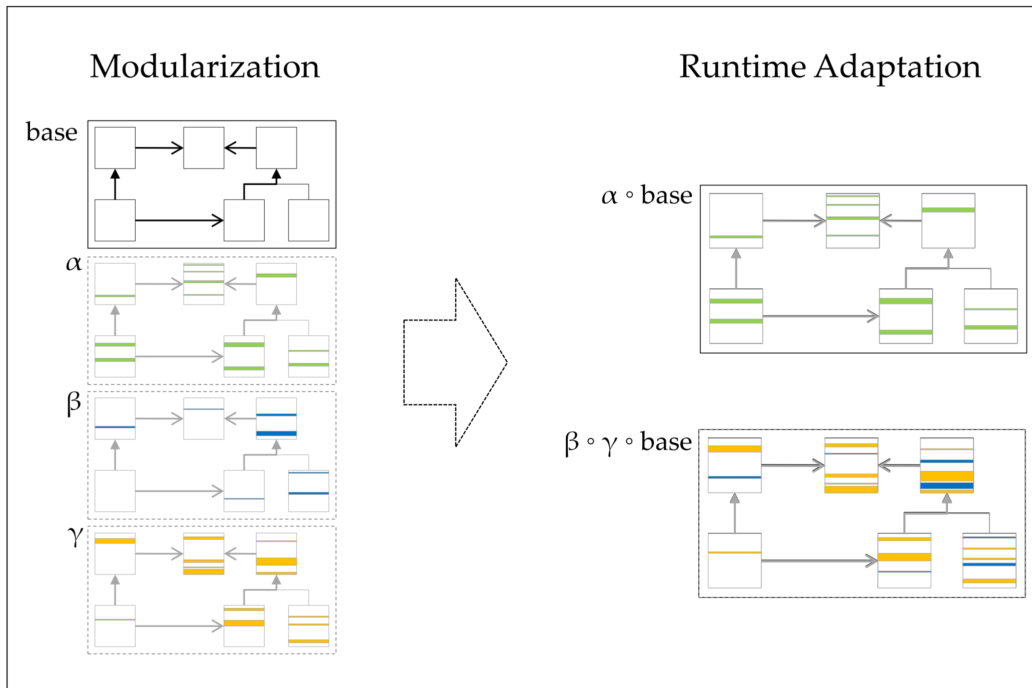


Figure 7.11: Cop modularization approach; Figure adapted from [15]

`ContextQuery(ContextRequest, String, String)` A query object is instantiated with its context type schema, a default namespace, and a string representation of the CQL expression.

`boolean ContextQuery.evaluate()` A query can be executed synchronously and will immediately return a Boolean value whether the context is accessible or not.

`void ContextQuery.evaluate(IContextHandler)` Queries can also be executed asynchronously. In that case, an `IContextHandler` object is passed to the query's evaluation method that is called by the CMS on context entry and exit.

`ResultSet ContextQuery.getResultSet()` The variable bindings of the last executed query are represented by a `ResultSet` that holds a list (for each solution found via backtracking) containing maps of key-value pairs. In addition, `ResultSet` provides some auxiliary methods such as `boolean isEmpty()`.

`void ContextQuery.addLayers(Layer[])` Layers can be associated with a query, for example, to make them accessible to the `IContextHandler` callback methods.

`void IContextHandler.onContextEntry / Exit(Layer[])` The callback methods are activated on context change and can be used for implementing any kind of reaction to the new state. They are parametrized with `Layer` objects if they have been associated with the query.

7.2.2 *Example*

In the following, we sketch a decomposition of the ToDo application example using our context query library. The example in Figure 7.12 presents three main activities: *modularized definition of adaptation code* with layers, *context reasoning* with the query language, and *activation of the adaptation* by a context object .

The layer `NearbyContactsMsg` implements the nearby contacts message display at the start of the calendar application (Lines 6–15). The layer and its partial methods require

context data in form of references to the nearby contact objects. This information is passed to the arguments of the layer constructor[8]. The layer activation is controlled by a `NearbyContacts` context object that is created and activated during generation of the corresponding `ToDoItem` (Lines 20–21). The context query is created on instantiation of `NearbyContacts`. First, we request a context schema from the `IContextAggregator` that defines the context types to be used in our query (Lines 35–40) and declare the default RDF namespace (Line 43). With that, we are able to specify the actual query as a string, using `Contact` and `NearbyService` and their properties. The query first selects all `Contact` entities that fulfill the condition of the `select` predicate, i.e. where first and last name match a `Contact` object (associated with the `ToDoItem`). These entries are then filtered by a `forAll` predicate. It only selects those entries for which the `NearbyService` context type finds a match within a range of 100 meters (Lines 44–47). We use JCop's `on` predicate to evaluate the context query at any *layered method*, that is, any method potentially affected by the respective layer activation (Lines 28–29). In this example we use the synchronous query evaluation that returns a Boolean value indicating if the context is active.

7.2.3 *Summary*

The integration of JCop with the CMI allows full runtime flexibility by parsing OCQL expressions before each call. The downsides of the approach are missing type safety for the returned context classes and a OCQL compilation overhead of > 100 ms for each call. Although the compilation step could likely be optimized further[9], an considerable overhead will remain compared to the pre-compiled variant that we benchmarked in the next section.

7.3 Intercepted Service Call Benchmark

The service method interception in Ditrios is realized by facilitating Java Proxies and reflection [136]. We have created a micro benchmark to evaluate the overhead of service method calls compared to regular Java calls. Every call kind is repeated for 5 second twice. First, to give the Java Hotspot just-in-time compiler time to optimize the calls, the second phase measures the optimized call values. Figure 7.13 shows the results. The analysis was carried out on an Intel Core i7-2600K, with 8-GB and the Java JDK 1.7 32-bit client VM[10] and the 32-bit SWI-Prolog 5.11.28, which is queried via the native Java Prolog Library (JPL).

We call a dummy service method which increments a field and is intercepted by the context-aware advice below:

```
pointcut benchmarkPointcut(Person l) :
   execution(* IQueryTestService.executeStringTest()) &&
   bind(l = Person->one( firstName="Mina"));

before benchmarkAdvice(Person p, QueryTestService service) :
       service(service) && benchmarkPointcut(p)
{
   BenchmarkTest.setAspectCalled(true);
}
```

The most-left result shows the evaluation of an un-cached service call, which evaluates the pointcut above on an RDF factbase with 500 contact classes. The whole call and

8 Layer instantiation has been recently introduced as a new feature to the JCop language.
9 In the current implementation parsing, static analysis, and generation of Prolog predicates write and read intermediate files stored in the file system.
10 The server VM improves the reflective call values by a factor of 4, but since we are targeting clients with the approach we use these more realistic values.

pointcut evaluation takes about 0.5 ms. As long as the context factbase is not changing, the subsequent advice calls are cached. In this case the advice call is in the order of magnitude of a reflective proxy call. For the cached advice call we measured 4500 calls/ms and for the reflective proxy call about 16000 calls/ms. The rest of the Figure shows how proxies and reflective calls compare to the direct invocation of a dummy method[11].

Since we only intercept service method calls, the performance is typically sufficient for most scenarios, without high-performance demands. In the benchmark example the method is empty except for a field increment. In a real-world example the calls on service will likely evaluate much more complex code and therefore the overhead of the call is neglectable. The main bottleneck are the non-cached advice methods, which can reduce the performance significantly when the context is changed in small time ranges, making a reevaluation necessary. To overcome this limitation the weaving could be separated into a primitive-pointcut evaluation part and a context query residue, where the later is only evaluated if the join point is matched by the primitive pointcuts. The first part can then be woven by optimized dynamic aspect approaches, restricting the evaluation of context queries to where necessary. A number of approaches can be used for the pre-weaving:

For arbitrary Java virtual machines JAC[160] and Handiwrap [23] weavers are applicable, which add advice hooks at every join point to a class at load- or compile-time. If the application is deployed on x86/amd64 Linux also PROSE 2 [164][12] and Steamloom [37] aspect weavers are an option which extend the Jikes RVM [115] and realize weaving by Java byte code manipulation.

7.4 SUMMARY

In this Chapter we have presented a case study on context-aware mobile applications, which compared different implementation variants based on pure object-orientation, service oriented-architecture and context-aware aspect-oriented programming. The study has shown that aspect-orientation successfully encapsulated crosscutting concerns in the analyzed application and reduced the need for adaptation anticipation.

The CMI offers three different variants to integrate OCQL into a language[13]. CSLogicAJ represents the most sophisticated approach concerning static type safety. Aspects are only type checked against their imported RDF schemata. In case of several deployed aspects, this leads to potentially different, aspect specific type hierarchies in an application.

In Section 7.2 we have presented the most dynamic use of OCQL, which does not rely on static typing at all. We embedded OCQL into the context-oriented programming language JCop, and enabled layer activation based on the state of the context management system. The realization demands a runtime compilation of the OCQL predicates, and therefore presents a very different use-case for the CMI as CSLogicAJ. The third OCQL integration variant, a global RDFS type hierarchy for the whole application, is left for future work, since it is mostly a specialization of the CSLogicAJ case.

We closed the Chapter with performance measurements of LogicAJ aspect weaving, which has shown potential for further optimization. We have discussed a possible optimization approach, which combines the underlying Ditrios architecture with a common dynamic weaving framework for aspect-languages.

11 The method was called on *volatile* field to avoid method in-lining.
12 The first version of PROSE facilitated the Java low level debugging interface, which resulted in an overall slow-down of the virtual machine execution.
13 see Section 5.1.4

```
1  public class CalendarApp {
2    public void initialize() {...}
3    ...
4  }
5
6  public layer NearbyContactsMsg {
7    private void ResultSet contacts;
8
9    public NearbyContactsMsg(ResultSet rs) {
10     thislayer.contacts = rs;
11   }
12   before public void CalendarApp.initialize() {
13     //show message which contacts are nearby
14   }
15 }
16
17 public class ToDoApp {
18   public addContactsToItem(Contact[] cts) {
19     ... //create a new ToDoItem
20     ctx = new NearbyContacts(cts);
21     ctx.activate();
22   }
23 }
24
25 public context NearbyContacts {
26   private ContextQuery nearby;
27
28   on(nearby.evaluate()) :
29    with(new NearbyContactsMsg(nearby.getResultSet()));
30
31   public NearbyContacts(Contact[] cts) {
32     this.nearby = createQuery(cts);
33   }
34   private ContextQuery createQuery(Contact[] cts) {
35     ContextRequest request =
36       CMS.getContextAggregator().requestSchema(
37         null,
38         "http://www.example.org/nearby.rdf",
39         "max(precision)",
40         Cardinality.single);
41     return new ContextQuery(
42       request,
43       "http://www.example.com",
44       "Contact->select(" + createCond(cts) + ")
45          ->forAll(c | NearbyService
46            ->nearby({maxDistance=100})
47              ->exists(c.email=email))");
48   }
49   private String createCond(Contact[] cts) {
50    // for each contact c in this.cts, generate:
51    // ( firstName=c1.getFirstName() &&
52    //   lastName=c1.getLastName() ) || ...
53   }
54 }
```

Figure 7.12: Implementation of the ToDo application using JCop's query library.

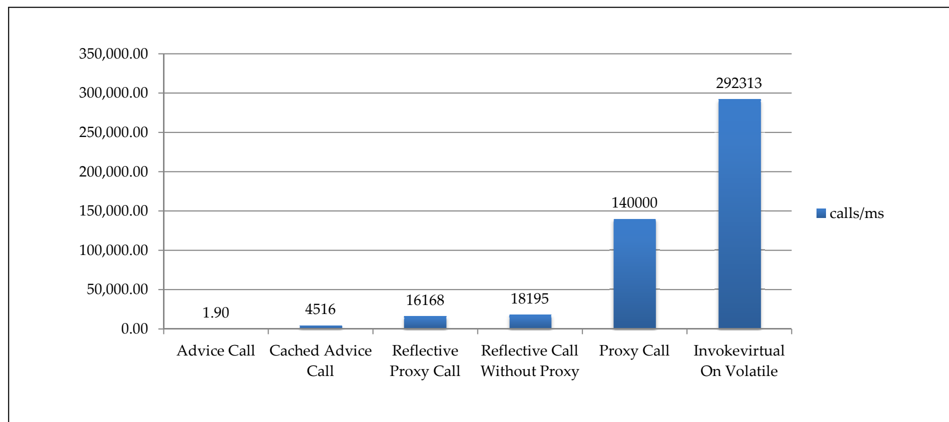


Figure 7.13: Cached service call compared with normal calls

Part IV

RELATED WORK, CONCLUSIONS, AND FUTURE WORK

8 RELATED WORK

This thesis has related work in several research areas. The OCQL language shares characteristics with other context query languages and the aspect-language with other (context-aware) dynamic aspect languages, and the context management system is similar to a range of approaches supporting context-awareness. The related work on RDF-Java mappings has already been discussed in Section 4.1 and will not be repeated here.

8.1 CONTEXT QUERY LANGUAGES

8.1.1 *SPARQL*

SPARQL is the standard query language for querying RDF graphs and is inspired by SQL. It uses triple patterns, which are matched with the given RDF graphs. Queries are evaluated on the RDF simple entailment [97, Section 2], but SPARQL endpoints[1] implementation may offer RDF Schema or OWL entailment, e.g., Pellet [184].

SPARQL and its precursors have been used as query languages for a number of context management systems [201, 193], but we did not base our approach on SPARQL for several reasons. Especially the current version, SPARQL 1.0, is limited in its expressiveness to relational algebra [13], and therefore recursive expressions, such as closures[2], are not expressible. And there are number of further limitations:

- It does not support meta-calls such as the findall statement in OCQL/Prolog.
- Variables cannot be typed, only explicit runtime tests can be added to ensure that the variable is bound to instances of a certain type.
- Does not support aggregation functions.
- Variable assignments are not possible, e.g., for string processing such as concatenation.
- Sub-queries in general and especially to remote databases (endpoints) are not supported.
- Negation-as failure (NaF) cannot be expressed directly.

As motivated in Section 5.5 we consider NaF essential for testing sensor reads. In SPARQL 1.0 NaF is only implicitly expressible via a combination of the *optional* and *filter* expression, see [167]. The expression first optionally binds an expression to a variable and later tests if it was bound, here is the original example from [167], which binds all persons which do not have the dc:date property:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
SELECT ?name
 WHERE { ?x foaf:givenName  ?name .
         OPTIONAL { ?x dc:date ?date } .
         FILTER (!bound(?date)) }
```

1 SPARQL endpoints are services, which provide access to an RDF knowledge base via the SPROT protocol [50].
2 For example, the transitive friends relationship.

In SPARQL 1.1 [181], which is at time of this thesis still a working draft, many limitations will be removed. For example, support for queries on the RDFS and OWL entailment [86] have been added, NaF is directly supported, path expressions allow the definition closures, variable assignment is possible, sub-queries enable alternating quantifiers on sub-queries, remote queries (federate expressions [166]) and aggregations will be added to the core of the language.

Still, SPARQL does not support variable typing and meta-calls and is limited in its general expressiveness. Further, universal quantification is only indirectly expressible. This makes queries like "a contact that knows all other contacts" more complex than in OCQL. Let's assume the following triples for a class `:Contact` are defined:

```
@prefix :  <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
:Contact rdf:type rdfs:Class.
:knows
    rdfs:range   :Contact ;
    rdfs:domain   :Contact.

:contact1
    rdf:type    :Contact ;
    :knows :contact1 ;
    :knows :contact2.

:contact2
    rdf:type    :Contact;
:knows :contact2.
```

The given query can be defined straight forward in OCQL:

```
Contact->select(contact|
   Contacts->forall(other| other = contact ||
                           contact.knows->one() = other)
```

In contrast SPARQL expression must simulate the universal quantification via an OPTIONAL clause and a later not-bound test:

```
PREFIX ex: <http://example.org/>
  SELECT ?contact
  WHERE {
    ?contact a ex:Contact .

    OPTIONAL {
      ?other a ex:Contact.
      NOT EXISTS {?contact ex:knows ?other }
      FILTER ( ?other != ?contact).
    } .
    FILTER ( !bound(?other) ) .
  }
```

In general SPARQL's SQL-like approach does not fit well with a pointcut language, which resembles more to logic than a relational language.

8.1.2 *SWRL*

The Semantic Web Rule Language (SWRL) is a W3C proposal for an OWL rule language, based on Rule Markup Language [111]. The antecedent of the rule is conjunction of positive literals. Negation is not allowed to avoid non-monotonic reasoning. All variables in the consequent must occur in the antecedent and are universally quantified. Literals

in SWRL are either OWL concepts *C*(x), properties *P*(x,y) or the comparison predicates sameAs(x,y) and differentFrom(x,y), where x and y are variables. Further SWRL specifies a number of math, comparison, string, date/time/duration and URI handling built-ins.

Its concrete syntax is defined in XML, but exemplarily we write a query with common formal logic operators. Let's assume we want to define the friends closure in SWRL:

```
friend(P1, P2) ∧ friend(P2, P3) ⇒ friend(P1, P3)
```

The rule is an axiom on the OWL ontology. In an implementation it is successively applied until a fixpoint is reached (e.g., by forward chaining [148]). The limitations compared to OCQL are quite obvious. Negation-as-failure is not possible in SWRL, no meta-predicate support, no variable typing. SWRL was not developed as a query language, but to enhance the expressiveness of OWL ontologies with Horn-like rules. With SQWRL [149], a query language was proposed, and implemented in Semantic Web framework Protégé [146] and has been used to build context-aware applications [150].

8.1.3 *MUSIC CQL*

As a critique of SPARQL Reichle et al. [169] presented a context query language, which was developed as part of the EU project MUSIC[3]. The first nine requirements[4] for our approach are based on requirements elicited for MUSIC project and therefore also fulfilled by their CQL approach. The MUSIC CQL is a XML-based query language based on an OWL context model. Queries are defined as constraints on the ontology and can explicitly specify reasoning of relations and apply time range restrictions. Compared to OCQL it does support variable typing, meta-predicates and, similar to SPARQL, it follows a relational approach.

8.1.4 *F-Logic / Flora-2*

Frame Logic (F-Logic) [119] is a general purpose object-oriented logic language that features the common object-oriented properties object identity, inheritance, polymorphic types, query methods and encapsulation. A number of languages have extended or adapted the concept [63, 183, 211, 132], where Flora-2 [211] is the most expressive approach, which also supports higher-order logic (Hi-Log) [49], and transaction logic [39].

Flora-2 was implemented based on the deductive database engine [176], and takes advantage of XSB's built-in tabling [176]. Attempts have been made to standardize the Flora-2 / F-Logic concepts as a Semantic Web language, the Semantic Web Services Language (SWSL) [27], but the competing SWRL approach has found more community and tool support [99].

Flora-2 takes a much more general and expressive approach than OCQL, but does not consider type checking. Via meta-programming runtime type checking is possible [210], but results in runtime overhead, and should only be used for debugging purposes [210, 26.2].

8.1.5 *Prova 2*

The knowledge reasoning system Prova 2 [158] is a typed logic language for the Semantic Web[5]. Instead of a static typing approach chosen for OCQL Prova2 applies a runtime type check on unifications. More precise, it uses *description logic-typed unification*, by extending the normal unification algorithm with the description logic reasoner Pellet [184], resulting

3 Self-adapting applications for Mobile Users In ubiquitous Computing environments
4 see Section 1.4
5 See also Section 2.4.3.

in an EXPTIME worst case complexity in case of the description logic OWL lite on unifications.

Prova2 is more an extension of Prolog, allowing the use of arbitrary complex terms in literal arguments that only allows tuples and lists. But the main differences between Prova2 and OCQL are static vs dynamic typing and OCQL's integration with the Java type system.

8.2 CONTEXT-MANAGEMENT SYSTEMS

Most context-aware platforms concentrate on context management, providing a well-defined interface for an application for context data handling and querying. The application anticipates the interfaces to the context management and is therefore itself context-aware. One example is the SOCAM approach [89] by Gu et al., an approach built on top of the OSGi framework. The context model is based on the Web Ontology Language OWL Lite [187] and RDF schema. Context reasoning is carried out with a rule system based on first-order predicate calculus. The rules can be evaluated with forward- or backward chaining, and both evaluation techniques can be combined. Based on triggered rule actions, which are not further specified, a client can react to context changes.

Semantic Space [204] is another related framework, which focuses on providing

- an explicit representation of raw context data,
- the means to acquire contexts via expressive queries, and
- high-level contexts through reasoning.

The context itself is modeled as an OWL ontology, which is queried by RDQL [140] an precursor of SPARQL. *Context wrappers* provide these contexts, they are discovered and handled by a *context aggregator*. Four basic classes can be used to characterize smart spaces; user, location and computing entity represent real world objects and activity a conceptual object. Though conceptually similar, it does not support AOP and context analysis is limited by SPARQL's expressiveness.

Oh et al. [151] developed a "context management architecture for large scale smart environments". Instead of a full-fledged ontology language they base on the context modeling language 5W1H [108], which represents a user centered context representation describing *Who* did *What*, *Where*, *When*, *Why*, and *How*. In contrast to most approaches community management is considered already on the architectural level. Communities are explicitly modeled and managed. Once community members are located in the same environment and have a similar goal context information between them is shared. Compared to our approach, the 5W1H language has a very limited expressiveness, it does not go beyond propositional logic [108].

The Ditrios architecture provides an exception handling mechanism by means of aspects. Architecture level exception handling for distributed services has also been proposed by [9]. The *DeEvolve* platform provides an exception handler concept with user interaction to look up service alternatives on failure. The handlers are defined in an XML description, which is also used to describe the service composition. Although the means are different the possibilities are similar. The aspects can be used for automatic reconfiguration or to incorporate the user into the selection process.

8.2.1 *Context-aware Aspect-oriented Programming*

AspectJ [118] offers a generic `if`(*BooleanExpression*) pointcut that executes an arbitrary boolean expression on static members of an aspect. For example consider the following

aspect that facilitates the boolean method contextActive(..) to check if a certain context is active at call-time:

```
public aspect ContextAwareAspect {
    static boolean contextActive(JoinPoint jp){ ... }
    static Context getContext() { ... }
    before() : if(contextActive(thisJoinPoint)) {
        getContext();
    }
}
```

But the transfer from the if pointcut expression to an advice is not possible with this construct, the advice must explicitly gather the context information in the advice. The AspectBench compiler [19] introduced the let pointcut [21] to bind objects to advice/-pointcut parameters. Avgustinov et al. used the let pointcut to bind the current join point object to compare it in trace matches, but did not combine this approach with a context model or context management framework.

Tanter et al. [197] presented a framework approach for *context-aware aspects* (CAA), which took this further step. CAAs is based on the Reflex AOP kernel [7], which offers a Java based-context definition framework and an API for defining context-related aspect constructs on metaobject provided by the Reflex framework. Tanter et al. used the framework approach instead of extending an existing aspect language for fast prototyping reasons [197].

CAAs can adapt the advice execution to the runtime state of the program. They proposed a Java-based context model and provide a pointcut-like API to bind and evaluate the current context. The CAA approach also supports context snapshots and pointcuts may refer to past events. But here the snapshot approach is targeted for storing only small parts of the global context, and relate it to single objects. For example, they define a promotional context for an online shop and associate its snapshot with creation time of a shopping cart. Later all join points on methods of this shopping card instance can refer to this snapshot although the promotional context might have vanished or changed in the meanwhile. The context classes are in responsibility for snapshot creation themselves, since the snapshot creation can become expensive in the presence of deep context object graphs. This approach differs from our realization of snapshots and in the representation of context information. No explicit context query model or language is defined. Since it is a framework approach a context management system could be combined with CAAs, but this step was left for future work.

Fuentes et al. [82] proposed an ambient intelligence domain-specific language for architecture level adaptations. Adaption strategies can be written, which change the application at runtime. The analysis of the context is carried out with SWRL [110]. Next to the limitation inherited from SWRL[6] the system does not offer asynchronous advice and does not offer a concept like query context sources.

8.2.2 *Dynamic Component-based Aspect-oriented Programming*

Several dynamic aspect languages for component systems have been proposed for Java [80, 160, 145, 194]. The performance for first and cached method calls is much better (orders of magnitudes for the first call) in these systems, see also Section 7.3. Our current approach is a pure on-demand weaving approach, which evaluates all pointcuts at every unvisited join point[7]. This is similar to the technique used by *Morphing Aspects* [93], which weaves aspects on-demand by meta-programming in Smalltalk. But CSLogicAJ considers, next to the pure program join point, also context data.

6 See Section 8.1.2.
7 The visit marker is resetted after every context change

Our dynamic weaving approach was not optimized for runtime performance, and we consider a combination of optimized dynamic weaving approaches with our context framework as future work (see Section 9.2). This can be achieved by separating aspect weaving and context analysis, which is by now realized in one step.

9 CONCLUSIONS AND FUTURE WORK

9.1 CONCLUSIONS

The main contribution of this work is the *Object-oriented logic Context Query Language* (OCQL), a query language that bridges logic programming, a Semantic Web-based context model, and the Java type system. It's type system allows for a tight integration with programming languages based on the Java type system. OCQL is complemented by a flexible context management infrastructure (CMI). Together, they support the creation of *statically typed context-aware adaptation languages*. We demonstrated this by means of an integration with an aspect-oriented and a context-oriented programming language.

We started this work with the identification of requirements for context-aware adaptation frameworks and languages. As a basis we took the requirements for context-query languages gathered by the MUSIC EU project [169] and extended the list to cover context-aware adaptation languages. Based on these requirements, we developed a CMI for context-aware adaptation languages, which relies on RDF Schema for context modeling. The CMI manages context sensors, either local or remote, and aggregates them in a context management system. The context information is queried via OCQL. As building blocks for this language, we developed

- an RDF Schema to Java mapping that considers multiple `rdfs:range` definitions;
- an extension of Lu's mode analysis algorithm for normal logic programs [130] and its formal semantics to support the higher-order predicate `findall`;
- a concept for the integration of a subset of Java Generics with a typed logic language, resulting in a type system with parametric polymorphism and subtyping, and support for a selected set of *all-solution* predicates;
- a type checking algorithm for OCQL that employs the mode analysis for dataflow analysis; and
- the specification of OCQL's semantics by a translation to a subset of Prolog.

We have build the CMI on top of the OSGi-based component framework Ditrios [180, 174], which offers a service interception mechanism. Combining both, service interception and context analysis, offers the means for context-aware service adaptation. The CMI and the OCQL language compiler were designed with extensibility in mind and allow for simple integration into programming languages based on the Java type system. The CMI offers three different levels of language integration, with tradeoffs concerning flexibility and static type safety (see Section 5.1.4).

We evaluated two different variants on two different languages. First, OCQL was combined with the context-oriented programming language JCop [175]. This approach demonstrated how OCQL can be applied in a dynamic setting, where OCQL is parsed and evaluated at runtime, at the expense of untyped query results and potential runtime errors.

The aspect language CSLogicAJ [174] demonstrated the full integration into a host language. Each aspect is type checked only against the RDFS classes it imports, building up its own context class taxonomy. This allows for the separate compilation of CSLogicAJ aspects. Further, the class hierarchy and properties used by an aspect cannot be influenced by RDF schemas referenced by other aspects. To our knowledge, CSLogicAJ is the first aspect language that fulfills all requirements we elicited for context-aware adaptation

systems. OCQL was integrated into CSLogicAJ's pointcut language, resulting in a *hybrid context analysis and pointcut language*.

Additionally, CLogicAJ offers an *asynchronous advice* construct. Asynchronous advice defines program-flow independent actuators, which react to context change and is a unique mechanism for aspect languages. This concept unifies common Event-Condition-Action rules [67] with aspect-orientated programming.

In a case study we compared the implementation a context-aware adaptation scenario, with plain object-oriented and service-oriented solutions. The comparison showed how context-aware aspect-oriented programming, combined with a service oriented architecture can

- reduce the anticipation of adaptations significantly,
- encapsulate crosscutting concerns on the architecture level, and
- dynamically apply adaptations on running applications.

9.2 FUTURE WORK

The work presented in this thesis can be evolved in several directions:

9.2.1 *Weaving Optimizations*

The dynamic aspect weaving approach used in the thesis is not optimized for runtime performance, and we consider the combination of other dynamic weaving approaches with Ditrios as a natural step. To achieve this we plan to separate the weaving process into a pure AspectJ primitive pointcut evaluation part and a context query residue, which is only evaluated if the join point is matched by the primitive pointcuts. The first part can be woven by an optimized dynamic aspect approach, restricting the evaluation of context queries to where necessary. A number of approaches can be used for the pre-weaving part:

For common Java virtual machines JAC [160] and Handiwrap [23] weavers are applicable, which add advice hooks at every join point to a class at load- or compile-time. If the application is deployed on x86/amd64 Linux also PROSE 2 [164][1] and Steamloom [37] aspect weavers are an option, which extend the Jikes RVM [115] and implement weaving by Java byte code manipulation.

9.2.2 *OWL Support*

The context model developed in this thesis is based on RDF Schema. All other Semantic Web languages can be serialized into RDF Schema and therefore a large number or resources are available for the CMI. However, RDF Schema is very limited its expressiveness. For example, we cannot restrict the cardinality of property relationships and cannot declare two individuals to be the same. The missing cardinality constraints lead to the general assumption in this work that a property of a context class is related to 0-N resources, although for a large number of properties, e.g., surname or the birthday of a person, the cardinality restrictions would avoid unnecessary checks and indirections.

The OWL languages, see Section 2.2.3, extend the vocabulary of RDFS resulting in more concise class definitions, including constructs to define cardinalities, mark two classes or objects as equivalent and more. Further, the variants OWL lite and OWL DL, distinguish

1 The first version of PROSE facilitated the Java low level debugging interface,which resulted in an overall slow-down of the virtual machine execution.

classes from objects (individuals), which is not the case in RDFS. This leads to a clear separation of type declarations and objects.

To accommodate OWL in OCQL's type system, we plan to extend the RDF-Java mapping[2] gradually, starting with a mapping of the most restricted variant OWL lite.

9.2.3 *Typed Java Library*

We defined three different levels of OCQL language integrations[3]. The second variant has the perquisite that all RDF schemas used in an application are statically known. This leads to a fixed context type taxonomy, enabling the generation of concrete Java interfaces for RDFS classes. Java clients can be compiled against these interfaces. In case that no runtime loading of querying clients will happen this is a suitable solution and does not involve further language extensions. A good candidate for such a controlled environment is the Android platform [1]. Android applications represent fixed deployments without any static or runtime dependencies, except for the Android SDK classes.

We will develop a context-management system for Android, which manages a central RDF-based context repository. A standalone variant of OCQL generates Java interfaces for the local context model for an Android application and an Java API[4] to represent the predicates as queries.

2 see Section 4.1
3 See Section 5.1.4.
4 Application Programming Interface

Part V

APPENDIX

APPENDIX A

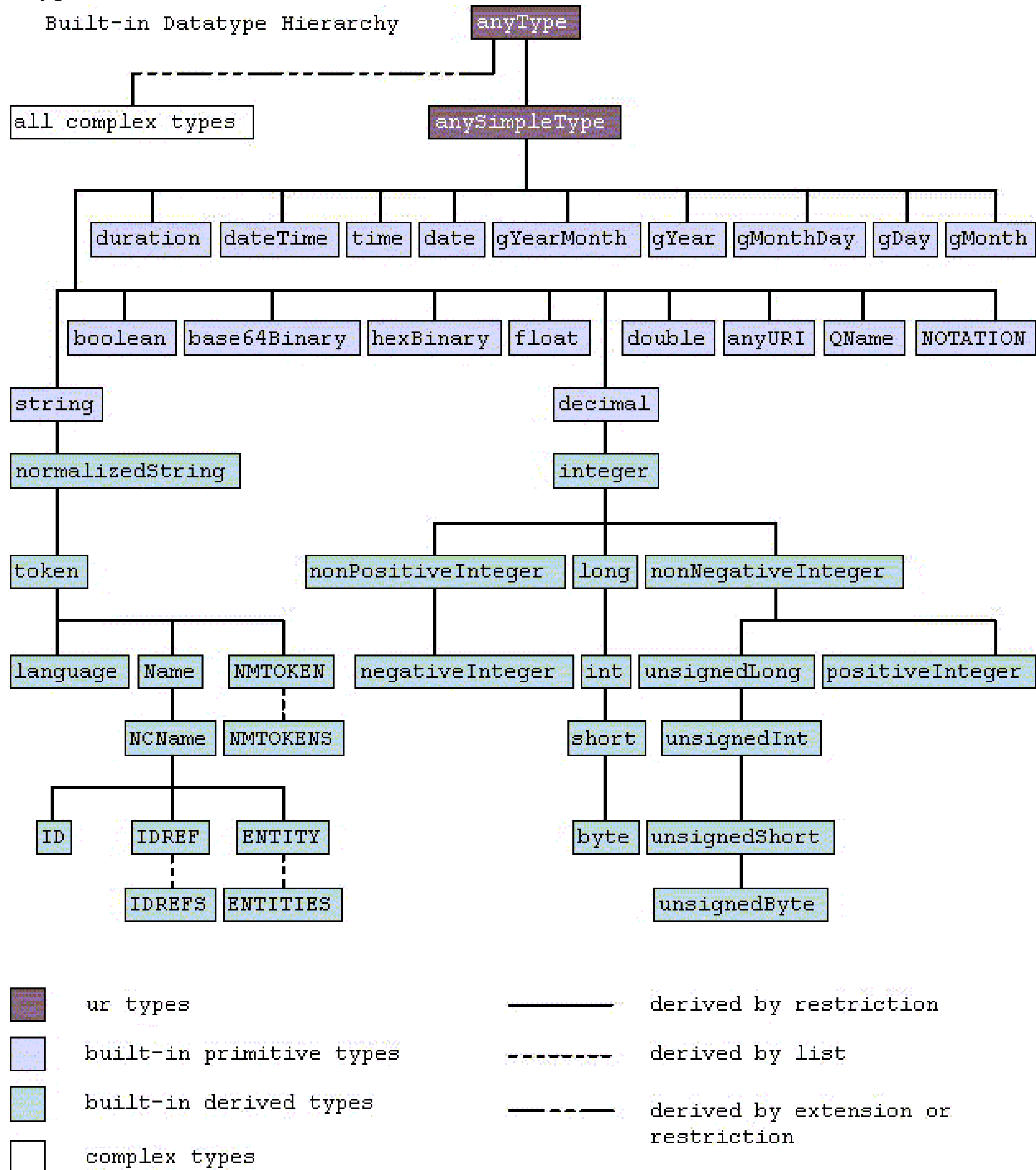
A.1 XML PRIMITIVE TYPE MAPPING

The table below lists the mapping of XML basic types to Java:

XML Primitive Type	Java Types
string	java.lang.String
boolean	Boolean
decimal	Integer
int	Integer
long	Long
short	Short
byte	Byte
float	Float
double	Double
duration	javax.xml.datatype.Duration
dateTime	java.util.Date
time	java.util.Date
date	java.util.Date

A.2 XSD BUILT-IN TYPE HIERARCHY

The following diagram shows the hierarchy of built-in xml schema primitive and derived datatypes.



Copyright © 2004 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231

A.2.1 *Lunjin Lu - Abstract Unification Algorithm*

The following generic abstract unification algorithm is an extract from [130], Chapter 5:

Algorithm 5.3.1 Let $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and A, B be atoms such that $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$. Define $share(X, t, \mathcal{S}) \stackrel{def}{=} \exists Y \in vars(t).(X, Y) \in \mathcal{S}$ where X is a variable, t is a term and $\mathcal{S}$ is a pair sharing relation.

$$unify^\flat(A, \theta^\flat, B, \sigma^\flat) \stackrel{def}{=} \begin{cases} let & \Psi \text{ be a renaming substitution such that} \\ & \mathcal{U}\Psi \cap \mathcal{V} = \emptyset, \\ & E_0 = eq(mgu(A\Psi, B)), \\ & \mathcal{V} = vars(\sigma^\flat \downarrow Alias) \\ in & \\ & \begin{cases} restrict(solve(E_0, \theta^\flat\Psi \uplus \sigma^\flat), \mathcal{V}) & if\ E_0 \neq fail \\ \bot[\![\mathcal{V}]\!] & otherwise \end{cases} \end{cases} \tag{5.1}$$

$$<\mathcal{I}, \mathcal{S}, \mathcal{A}> \uplus <\mathcal{I}', \mathcal{S}', \mathcal{A}'> \stackrel{def}{=} <\mathcal{I} \cup \mathcal{I}', \mathcal{S} \cup \mathcal{S}', \mathcal{A} \cup \mathcal{A}'> \tag{5.2}$$

$$solve(E, \theta^\flat) \stackrel{def}{=} \begin{cases} let & E_1 = \{(X = Y) \in E \mid Y \in \mathcal{VAR}\}, \\ & E_2 = E \setminus E_1, \\ & \zeta^\flat = solve0(E_1, \theta^\flat) \\ in & \\ & \begin{cases} \bot[\![vars(\theta^\flat \downarrow Alias)]\!] & if\ failure(\zeta^\flat \downarrow Alias, E_2) \\ solve0(E_2, \zeta^\flat) & otherwise \end{cases} \end{cases} \tag{5.3}$$

$$solve0(E, \theta^\flat) \stackrel{def}{=} \begin{cases} \theta^\flat & if\ E = \emptyset \\ solve0(E', soln(e, \theta^\flat)) & if\ E = \{e\} \cup E' \end{cases} \tag{5.4}$$

$$restrict(<\mathcal{I}, \mathcal{S}, \mathcal{A}>, \mathcal{V}) \stackrel{def}{=} <\{X/\mathcal{I}(X) \mid X \in \mathcal{V}\}, \mathcal{S} \cap \mathcal{V}^2, \mathcal{A} \cap \mathcal{V}^2> \tag{5.5}$$

$$soln(e, \sigma^\flat) \stackrel{def}{=} reduce(compose(\sigma^\flat, unieq(e, \sigma^\flat), vars(e))) \tag{5.6}$$

$$unieq(X = t, <\mathcal{I}, \mathcal{S}, \mathcal{A}>) \stackrel{def}{=} <<\mathcal{I}', \mathcal{S}', \mathcal{A}'>, \mathcal{Z}'> \tag{5.7}$$

where

$$\mathcal{I}' = \begin{cases} let\ \mathcal{I}_0 = \mathbf{est}(X = t, \mathcal{I}) \\ in \\ \lambda U. \begin{cases} \mathcal{I}_0(Y) & if\ \exists Y \in vars(t) \cup \{X\}.((U, Y) \in \mathcal{A}) \\ \mathcal{I}(U) & otherwise \end{cases} \end{cases}$$

$$
\mathcal{S}' = \begin{cases} let\ \mathcal{V}_t = vars(t) \setminus \{V \mid \mathbf{grd}(\mathcal{I}(V))\}, \\ \quad \kappa^1 = \chi^\flat(X, <\mathcal{I}, \mathcal{S}, \mathcal{A}>), \kappa^2 = \chi^\flat(t, <\mathcal{I}, \mathcal{S}, \mathcal{A}>), \kappa = (\kappa^1, \kappa^2) \\ in \\ \begin{cases} \emptyset & if\ \kappa^1 = 0 \vee \kappa^2 = 0 \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} & if\ \kappa = (1,1) \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} \cup \{(X,X)\} & if\ \kappa = (1,2) \wedge \neg share(X,t,\mathcal{S}) \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} \cup \mathcal{V}_t^2 & if\ \kappa = (2,1) \wedge \neg share(X,t,\mathcal{S}) \\ \{(X,Y),(Y,X) | Y \in \mathcal{V}_t\} \cup \{(X,X)\} \cup \mathcal{V}_t^2 & otherwise \end{cases} \end{cases}
$$

$$
\mathcal{A}' = \begin{cases} (\{(X,t),(t,X)\} \cup \mathcal{A})^* & if\ t \in \mathcal{VAR} \\ \mathcal{A} & otherwise \end{cases}
$$

$$
\mathcal{Z}' = \begin{cases} \emptyset & if \quad \mathbf{grd}(\mathcal{I}(X)) \wedge \forall Y \in vars(t).\mathbf{grd}(\mathcal{I}(Y)) \\ & \quad \vee\ \mathbf{var}(\mathcal{I}(X)) \wedge t \in \mathcal{VAR} \wedge \mathbf{var}(\mathcal{I}(t)) \\ \{X\} & if \quad \mathbf{var}(\mathcal{I}(X)) \\ vars(t) & if \quad t \in \mathcal{VAR} \wedge \mathbf{var}(\mathcal{I}(t)) \\ \{X\} \cup vars(t) & otherwise \end{cases}
$$

$$
compose(<\mathcal{I}, \mathcal{S}, \mathcal{A}>, <<\mathcal{I}', \mathcal{S}', \mathcal{A}'>, \mathcal{Z}'>, \mathcal{Z}_0) \stackrel{def}{=} <\mathcal{I}'', \mathcal{S}'', \mathcal{A}''>
$$

where

$$
\begin{aligned}
\mathcal{I}'' &= \lambda U. \begin{cases} \mathcal{I}'(U) & if\ \exists Y \in \mathcal{Z}_0.(U,Y) \in \mathcal{A} \\ \mathbf{ins}(\mathcal{I}'(U)) & if\ \exists Z \in \mathcal{Z}'.(U,Z) \in \mathcal{S} \\ \mathcal{I}'(U) & otherwise \end{cases} \\
\mathcal{S}'' &= \mathcal{S} \cup \mathcal{S}' \cup \\
&\quad \{(U,Z) | \exists V.((U,V) \in \mathcal{S}' \wedge (V,Z) \in \mathcal{S})\} \cup \\
&\quad \{(W,V) | \exists U.((W,U) \in \mathcal{S} \wedge (U,V) \in \mathcal{S}')\} \cup \\
&\quad \{(W,Z) | \exists U,V.((W,U) \in \mathcal{S} \wedge (U,V) \in \mathcal{S}' \wedge (V,Z) \in \mathcal{S})\} \\
\mathcal{A}'' &= \mathcal{A}'
\end{aligned}
$$

$$
reduce(<\mathcal{I}, \mathcal{S}, \mathcal{A}>) \stackrel{def}{=} <\mathcal{I}, \left\{ (X,Y) \in \mathcal{S} \;\middle|\; \begin{array}{ll} & \neg\mathbf{grd}(\mathcal{I}(X)) \wedge \neg\mathbf{grd}(\mathcal{I}(Y))\} \\ \wedge & (X \equiv Y \rightarrow \neg\mathbf{var}(\mathcal{I}(X))) \end{array} \right\}, \mathcal{A}>
$$

A.2.2 *Implementation of Array Index Access*

Below the predicate arrayAccess/3 is defined, which is used in the realization of the OCQL predicate index/3:

```
arrayAccess(0, [Head|_], Head) :- !.
arrayAccess(N, [_|Tail], Elem) :-
   nonvar(N),
   M is N-1,
   arrayAccess(M, Tail, Elem).
arrayAccess(N,[_|T],Item) :-
   arrayAccess(M,T,Item), N is M + 1.
```

A.2.3 *Predefined Predicates and Arithmetic Expressions*

This sections contains a number of predefined CSLogicAJ/OCQL predicates and arithmetic expressions. The built-in OCQL predicates can easily be extended with arbitrary Prolog predicates, see also Section 6.2.1.

CSLogicAJ Aspect Pointcuts

call(*MethodPat*) the AspectJ call pointcut, e.g., call(void Foo.m(int)) selects all calls to the method void Foo.m(int)

target(*Type*|*Var*) binds all join points whose runtime receiver object is of type `Type`. In case the argument is the name of an advice parameter (`Var`) the type comparison is applied on the parameters type and the advice parameter is bound to the receiver object.

this(*Type*|*Var*) similar to target, but here the enclosing runtime instance of the join point's call site is checked and bound.

args(*Type*|*Var*,...) binds all join points, whose arguments are compatible with the list of type names or advices variables. Arguments are either method call arguments or the values assigned to a field. For method calls the wildcard '..' can be used for an arbitrary number of parameters.

currentAspect(String aspectName) Bind `aspectName` to the enclosing concrete aspect's class.

Service Predicates

service(advicdArg) binds *adviceArg* to services implementing the *adviceArg*'s interface. The type of adviceArg must be an interface.

service(advicdArg[, TupleInit]) binds *adviceArg* to the first service implementing *adviceArg*'s interface and that the constraint on the service's LDAP properties defined by the *TupleInit* expression, for instance the predicate p(MapView map) : service(map,{centered="true"}). Binds a *map* to service that has a property *centered* with the value "true".

String Predicates

concat(String arg1, String arg2, String arg3) The third argument forms the concatenation of the first and the second argument

camelCase(String identifier, String camelCaseIdentifier) converts to an identifier to camel case syntax, for example: `camelCase("myField", "MyField")`

lowerCase(String anyCase, String lowerCase) Converts the characters of `anyCase` into lower case and binds the String to `lowerCase`.

regex(String arg, String regEx, String[] groups) Checks if the pattern `regEx` matches the first argument.

replace(String input,char search,char replace,String result) replaces in the string `input` the character `search` with the character `replace` and binds the resulting string to `result`, e.g., `replace("asdf","s","f","afdf")`

stringToCharacters(String s, String[] characters) Converts between a string `s` and an array of `characters`.

subString(String string, int start, int length, int after, String sub) `subString` maintains the following relation: `sub` is a sub-string of `str` that starts at `before`, has `len` characters and `str` contains `after` characters after the match.

upperCase(String anyCase, String upperCase) Converts the characters of `anyCase` into upper case and binds the String to `upperCase`.

Array Predicates

<T> member(T[] member, T[] array) Checks if the first argument is a member of the `array` bound to the second argument; the second argument must already be ground

length(Object[] a, int len) Succeeds if `len` represents the number of elements of the array `a`.

<T> union(T[] array1, T[] array2,T[] union) the last argument is a set and the union of `array1` and `array2`

<T> removeElements(T[] remove, T[] fromArray, T[] resultArray) removes the elements in array `remove` from the array `fromArray` and binds the resulting array to `resultArray`.

Meta Predicates

findall(*varOrTuple*,*BooleanExpr*,*Bag*) backtracks over all solutions of *BooleanExpr* and aggregates the results as an array of the instantiations of the variable or tuple *Template* and unifies the array with *Bag*.

bagof(*varOrTuple*,*BooleanExpr*,*Bag*) is similar to the `bagof/3` predicate in Prolog, but without existential variables[1]. The semantics is similar to findall, but bagof backtracks over the alternatives of free variables in *BooleanExpr* which are not in *varOrTuple*.

setof(*varOrTuple*,*BooleanExpr*,*Set*) is similar to the `bagof/3`, but the last argument will be bound to a *set* instead of a *bag* of bindings.

Variables

var(varname) tests if the variable `varname` has not been bound, yet.

ground(varname) tests if the variable `varname` is ground, meaning it is either bound to an class instance/literal or a tuple/array that does not contain non-bound variables.

1 Which are syntactic sugar an can be simulated by a forward predicate

Arithmetic Expressions

OCQL supports all arithmetic expressions and bitwise functions defined in the ISO Prolog standard ISO [112, 9.3, 9.4]. These include the expressions power, sine, cosine, arctangent, exponential, logarithm, and square root. And the bitwise functions left shift, right shift, bitwise and, bitwise or, and bitwise complement.

BIBLIOGRAPHY

[1] Android Software Development Kit. URL `http://developer.android.com/sdk/index.html`. (Cited on page 149.)

[2] Cliopatria: the SWI-Prolog rdf toolkit. http://cliopatria.swi-prolog.org. (Cited on page 82.)

[3] Java platform standard edition 6, api specification. URL `http://download.oracle.com/javase/6/docs/api/`. (Cited on page 56.)

[4] last.fm. URL `http://www.last.fm/about`. (Cited on page 4.)

[5] Google latitude. URL `http://www.google.com/latitude`. (Cited on page 4.)

[6] Quintus Prolog Modules. URL `http://www.sics.se/isl/quintus/html/quintus/ref-mod.html`. (Cited on page 21.)

[7] A versatile kernel for multi-language aop. (Cited on page 145.)

[8] Joseph Albahari and Ben Albahari. C# 4.0 in a nutshell: The definitive reference. 2010. (Cited on page 89.)

[9] Sascha Alda and Armin B. Cremers. Towards composition management for component-based peer-to-peer architectures. In *Proceedings of the Workshop Software Composition (SC 2004)*, pages 42 – 58, April 2004. (Cited on page 144.)

[10] OSGi Alliance. *Listeners Considered Harmful: The "Whiteboard" Pattern - Revision 2*, August 2004. (Cited on page 100.)

[11] OSGi Alliance. *OSGi Service Platform Service Compendium - Release 4.2*, Juni 2009. (Cited on page 13.)

[12] OSGi Alliance. *OSGi Service Platform Core Specification - Release 4.2*, Juni 2009. (Cited on pages 13 and 15.)

[13] Renzo Angles and Claudio Gutierrez. The expressive power of sparql. *The Semantic Web-ISWC 2008*, pages 114–129, 2008. (Cited on page 141.)

[14] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. Dedicated programming support for context-aware ubiquitous applications. In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM '08. The Second International Conference on*, pages 38–43, oct 2008. doi: 10.1109/UBICOMM.2008.56. (Cited on page 133.)

[15] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, pages 50–65. Springer, 2010. (Cited on pages xi, 125, 133, and 134.)

[16] Krzysztof R. Apt and Maarten H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982. URL `http://doi.acm.org/10.1145/322326.322339`. (Cited on page 20.)

[17] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006. (Cited on page 30.)

[18] Brian Arkills. *LDAP Directories Explained: An Introduction and Analysis*. Addison-Wesley Professional, 2003. ISBN 020178792X. (Cited on page 15.)

[19] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible aspectj compiler. *Transactions on Aspect-Oriented Software Development*, October 2005. (Cited on page 145.)

[20] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising aspectj. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 117–128, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065026. URL http://doi.acm.org/10.1145/1065010.1065026. (Cited on page 31.)

[21] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie Hendren, Oege De Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. Efficient trace monitoring. Technical report, Formal Approaches to Testing Systems and Runtime Verification (FATES/RV), Lecture Notes in Computer Science, 2006. (Cited on page 145.)

[22] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003. (Cited on page 18.)

[23] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 86–95. ACM, 2002. (Cited on pages 136 and 148.)

[24] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007. (Cited on pages 4 and 6.)

[25] Roberto Barbuti, Roberto Giacobazzi, and Giorgio Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):133–181, 1993. (Cited on pages 28 and 37.)

[26] Jon Barwise and John Etchemendy. *Language, Proof, and Logic*. CSLI Publications, Stanford, California, 1999. (Cited on page 74.)

[27] Steve Battle, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger, Richard Hull, Michael Kifer, David Martin, Sheila McIlraith, Deborah McGuinness, Jianwen Su, and Said Tabet. Swsl-rules: A rule language for the semantic web. Technical report, Semantic Web Services Language Committee of the Semantic Web Services Initiative, 2005. (Cited on page 143.)

[28] C. Beierle. Logic programming with typed unification and its realization on an abstract machine. *IBM Journal of Research and Development*, 36(3):375–390, 1992. (Cited on page 74.)

[29] Christoph Beierle. Concepts, implementation, and applications of a typed logic programming language. In *Logic Programming: Formal Methods and Practical Applications*, pages 139–167. 1995. (Cited on page 25.)

[30] Christoph Beierle, Gregor Meyer, and Heiner Semle. A brief description of the protos-l system. pages 402–404, 1991. URL http://dl.acm.org/citation.cfm?id=648111.748585. (Cited on page 25.)

[31] Besmir Beqiri. Xtreme media player project. URL http://xtrememp.sourceforge.net/. (Cited on page 107.)

[32] Sebastian Bergmann and Günter Kniesel. Gap: Generic aspects for php. *Proc. 3rd European Workshop on Aspects in Software*, 2006. (Cited on page 30.)

[33] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001. (Cited on page 30.)

[34] Tim Berners-Lee. Www past & future. URL http://www.w3.org/2003/Talks/0922-rsoc-tbl/. http://www.w3.org/2003/Talks/0922-rsoc-tbl/. (Cited on page 4.)

[35] Tim Berners-Lee and Dan Connolly. Primer: Getting into rdf & semantic web using n3. *World Wide Web Consortium*, 2000. (Cited on page 16.)

[36] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001. (Cited on page 16.)

[37] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-842-3. doi: http://doi.acm.org/10.1145/976270.976282. (Cited on pages 136 and 148.)

[38] Egon Boerger and Bart Demoen. A framework to specify database update views for Prolog. *Lecture Notes in Computer Science*, 528, 1991. ISSN 0302-9743. (Cited on page 124.)

[39] Anthony J. Bonner and Michael Kifer. A logic for programming database transactions. *Logics for Databases and Information Systems*, pages 117–166, 1998. (Cited on page 143.)

[40] Gilad Bracha. Generics in the java programming language. *Sun Microsystems, java. sun. com*, 2004. (Cited on page 9.)

[41] Gilad Bracha, Peter von der Ahe, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In Theo D'Hondt, editor, *European Conference on Object-Oriented Programming 2010*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer Berlin, Heidelberg, 2010. ISBN 978-3-642-14106-5. (Cited on page 8.)

[42] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. *Proceedings of the 1st ACM SIGSOFT workshop on Selfmanaged systems WOSS 04*, 04 (November):28–33, 2004. URL http://oro.open.ac.uk/23229/. (Cited on page 2.)

[43] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. World Wide Web Consortium, Recommendation REC-xml-names-19990114, January 1999. (Cited on pages 57 and 58.)

[44] Dan Brickley. Rdf vocabulary description language 1.0: Rdf schema, 2004. URL http://www.w3.org/tr/rdf-schema/. (Cited on page 55.)

[45] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF schema. World Wide Web Consortium, Recommendation REC-rdf-schema-20040210, February 2004. (Cited on pages 4, 17, and 59.)

[46] Maurice Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, 1991. revised version of K.U.L. technical report CW 62, 1987. (Cited on page 28.)

[47] Avi Bryant and Robert Feldt. AspectR - simple aspect-oriented programming in ruby, 2004. (Cited on page 30.)

[48] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83. ACM, 2004. (Cited on page 18.)

[49] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993. (Cited on page 143.)

[50] Kendall Grant Clark, Elias Torres, and Lee Feigenbaum. SPARQL protocol for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/. (Cited on page 141.)

[51] Tony Clark and Jos Warmer. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263. Springer Verlag, 2002. ISBN 3540431691. (Cited on page 64.)

[52] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer Verlag, July 2003. (Cited on pages 19 and 64.)

[53] Michael Codish, Dennis Dams, Gilberto Filé, and Maurice Bruynooghe. Freeness analysis for logic programs-and correctness? In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 116–131, Budapest, Hungary, 1993. The MIT Press. ISBN 0-262-73105-3. (Cited on page 28.)

[54] Alain Colmerauer. Prolog and infinite trees. *Logic Programming*, 16:231–251, 1982. (Cited on page 19.)

[55] Agostino Cortesi and Gilbert Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. *ACM SIGPLAN Notices*, 26(9):52–61, September 1991. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). (Cited on page 28.)

[56] Pascal Costanza. Dynamic replacement of active objects in the gilgul programming language. *Component Deployment*, pages 391–411, 2002. (Cited on page 8.)

[57] Pascal Costanza. A short overview of AspectL. In *European Interactive Workshop on Aspects in Software (EIWAS'04), Berlin, Germany*, 2004. (Cited on page 8.)

[58] Patrick Cousot. Methods and Logics for Proving Programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 814–993. MIT Press/Elsevier, 1990. (Cited on page 26.)

[59] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, January 1977. (Cited on page 26.)

[60] Patrick Cousot and Radhia Cousot. Abstract Interpretation and applications to logic programs. *J of Logic Programming*, 13(2-3):103–180, July 1992. (Cited on pages 26 and 27.)

[61] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM, 2005. (Cited on page 4.)

[62] Saumya K. Debray and David S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming*, 5(3):207–230, September 1988. (Cited on page 28.)

[63] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Ontobroker: Ontology based access to distributed and semi-structured information. *Database Semantics: Semantic Issues in Multimedia Systems*, 351, 1999. (Cited on page 143.)

[64] Brecht Desmet, Jorge Vallejos, and Pascal Costanza. Introducing mixin layers to support the development of context-aware systems. In *3rd European Workshop on Aspects in Sofware (EWAS 2006)*, University of Twente, Enschede, The Netherlands, August 2006. URL `http://p-cos.net/documents/mixin-layers.pdf`. (Cited on page 6.)

[65] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001. ISSN 1617-4909. doi: http://dx.doi.org/10.1007/s007790170019. (Cited on pages 3 and 4.)

[66] Roland Dietrich and Frank Hagl. A polymorphic type system with subtypes for Prolog. In *ESOP'88*, pages 79–93. Springer, 1988. (Cited on page 25.)

[67] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto: A rulebase of adbms features. *Rules in Database Systems*, pages 1–17, 1995. (Cited on page 148.)

[68] Margaret H. Dunham and Abdelsalam Helal. Mobile computing and databases: Anything new? *Acm Sigmod Record*, 24(4):5–9, 1995. (Cited on page 7.)

[69] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1994. ISBN 0-8053-1753-8. (Cited on page 23.)

[70] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR, 2005. ISBN 0-131-85858-0. (Cited on page 3.)

[71] Thomas Erl. *SOA design patterns*. Prentice-Hall PTR, 2009. ISBN 0-136-13516-1. (Cited on page 3.)

[72] Patel-Schneider Peter F. and Boris Motik. Owl 2 web ontology language mapping to rdf graphs - w3c recommendation. 2009. (Cited on page 91.)

[73] Francois Fages and Emmanuel Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, 2001. URL `citeseer.ist.psu.edu/496783.html`. (Cited on page 24.)

[74] David C. Fallside and Priscilla Walmsley. Xml schema part 0: primer second edition. *W3C recommendation*, 2004. (Cited on page 16.)

[75] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns*, volume 2000, 2000. (Cited on pages 3 and 30.)

[76] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Akşit. Aspect-oriented software development. 2004. (Cited on page 3.)

[77] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7. (Cited on pages 7 and 30.)

[78] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem* 1. *Artificial intelligence*, 19(1):17–37, 1982. (Cited on page 18.)

[79] Free Software Foundation. The GNU multiple precision arithmetic library, 2011. URL `http://gmplib.org/`. (Cited on page 116.)

[80] Andreas Frei. *Jadabs - An Adaptive Pervasive Middlerware Architecture*. No. 16273, ETH, October 2005. (Cited on pages 3, 8, and 145.)

[81] Thom Frühwirth. *Constraint handling rules*. Cambridge University Press, 2009. ISBN 0-521-87776-8. (Cited on page 24.)

[82] Lidia Fuentes and Daniel Jimenez. An ambient intelligent language for dynamic adaptation. ECOOP Workshop OT4AmI, 2005. (Cited on pages 3, 6, 8, and 145.)

[83] Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison Wesley, 2003. ISBN 0-321-20575-8. (Cited on page 123.)

[84] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994. (Cited on pages 127 and 130.)

[85] Sabine Glesner and Karl Stroetmann. Combining inclusion polymorphism and parametric polymorphism. Technical report, Jan 1999. (Cited on page 71.)

[86] Birte Glimm and Markus Krötzsch. Sparql beyond subgraph matching. In Peter Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web - ISWC 2010*, volume 6496 of *Lecture Notes in Computer Science*, pages 241–256. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-17745-3. (Cited on page 142.)

[87] Sebastian Gonzales, Wolgang Demeuter, Pascal Costanza, Stéphane Ducasse, Richard Gabriel, and Theo D'hondt. Report of the ECOOP'03 workshop on object-oriented language engineering in post-java era, 2004. URL `http://www.iam.unibe.ch/~scg/Archive/Papers/Gonz04aoolepje04-report.pdf`. (Cited on page 6.)

[88] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *Java (TM) Language Specification, The Third Edition*. Addison-Wesley Professional, third edition edition, May 2005. ISBN 0-321-24678-0. (Cited on pages x, 56, 76, 79, and 80.)

[89] Tao Gu, Hung Keng Pung, and Da Qing Zhang. Toward an osgi-based infrastructure for context-aware applications. In *IEEE Pervasive Computing, vol. 03, no. 4*, Oct-Dec 2004. (Cited on page 144.)

[90] S. Gudmundson and G. Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. *Advanced Separation of Concerns*, 168, 2001. (Cited on page 8.)

[91] Kris Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *In Second Workshop on Aspect-Oriented Software Development*, pages 21–22, 2002. (Cited on page 30.)

[92] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM, 2003. (Cited on page 103.)

[93] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 46–55. ACM, 2004. (Cited on page 145.)

[94] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-orientedprogramming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-471-1. doi: http://doi.acm.org/10.1145/582419.582436. (Cited on page 130.)

[95] Robert Harper. Programming in standard ml. *Online tutorial notes available from http://www. cs. cmu. edu/rwh/introsml/index. html*, 1998. (Cited on page 9.)

[96] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *ACM Sigplan Notices*, volume 28, pages 411–428. ACM, 1993. (Cited on page 30.)

[97] Patrick Hayes. RDF semantics. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/. (Cited on page 141.)

[98] Patrick Hayes. RDF semantics. World Wide Web Consortium, Recommendation REC-rdf-mt-20040210, February 2004. (Cited on pages 17, 56, and 58.)

[99] John Hebeler, Matthew Fisher, Ryan Blace, Andrew Perez-Lopez, and Mike Dean. *Semantic web programming*. Wiley, 2009. (Cited on page 143.)

[100] Nevin Heintze and Joxan Jafar. A finite presentation theorem for approximating logic programs. In *Conference record of the 17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 197–209, 1990. (Cited on page 22.)

[101] Andres Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003. (Cited on page 9.)

[102] Patricia M. Hill and Jloyd Lloyd. *The Gödel programming language*. The MIT Press, 1994. (Cited on pages 9, 22, and 25.)

[103] Patricia M. Hill and Rodney W. Topor. A semantics for typed logic programs. *Types in Logic Programming*, pages 1–62, 1992. (Cited on pages 23 and 24.)

[104] Robert Hirschfeld. Aspects - Aspect-Oriented Programming with Squeak. *Objects Components Architectures Services and Applications for a Networked World*, 1(2591):216–232, 2003. (Cited on page 8.)

[105] Robert Hirschfeld. Aspects-aspect-oriented programming with squeak. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, 2003. (Cited on page 30.)

[106] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), march 2008. (Cited on page 133.)

[107] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008. (Cited on pages 3, 4, and 6.)

[108] Dongpyo Hong, Hedda R. Schmidtke, and Woontack Woo. Linking context modelling and contextual reasoning. In *4th International Workshop on Modeling and Reasoning in Context (MRC)*, pages 37–48, 2007. (Cited on page 144.)

[109] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16:14–21, 1951. (Cited on page 19.)

[110] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004. (Cited on page 145.)

[111] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosofand, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission, May 2004. URL http://www.w3.org/Submission/SWRL/. Last access on Dez 2008 at: http://www.w3.org/Submission/SWRL/. (Cited on page 142.)

[112] ISO. *ISO/IEC ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core*. ISO/IEC, 1995. (Cited on page 159.)

[113] ISO. *ISO/IEC 29341-3-2:2008: Information technology — UPnP Device Architecture – Part 3-2: Audio Video Device Control Protocol - Media Renderer Device*. ISO/IEC, 2008. (Cited on page 4.)

[114] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987. (Cited on page 24.)

[115] Jikes. The Jikes research virtual machine. URL http://jikesrvm.org. http://jikesrvm.org. (Cited on pages 136 and 148.)

[116] Aditya Kalyanpur, Daniel Pastor, Steve Battle, and Julian Padget. Automatic mapping of OWL ontologies into Java. In *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2004. (Cited on page 56.)

[117] 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, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag. (Cited on page 30.)

[118] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001. (Cited on pages 7, 30, 31, and 144.)

[119] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM (JACM)*, 42(4):741–843, 1995. (Cited on page 143.)

[120] Feliks Kluźniak. Type synthesis for ground Prolog. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pages 788–816. MIT Press, 1987. (Cited on page 22.)

[121] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. *Changes*, 2004. URL http://www.w3.org/TR/rdf-concepts/. (Cited on pages 4 and 16.)

[122] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. Workshop on Linking Aspect Technology and Evolution (LATE'07), in conjunction with Sixth International Conference on Aspect-Oriented Software Development (AOSD.07), March 12-16, 2007, Vancouver, British Columbia, Mar 2007. URL http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/knieselHannemannRho-late07.pdf. (Cited on pages 119 and 123.)

[123] Alexander Kozlenkov and Michael Schroeder. PROVA: Rule-based Java-scripting for a Bioinformatics Semantic Web. *Lecture Notes in Computer Science*, 2994:17–30, 2004. ISSN 0302-9743. (Cited on pages 26 and 74.)

[124] Lee W. Lacy. *OWL: Representing information using the web ontology language*. Trafford Publishing, 2005. ISBN 1-4120-3448-5. (Cited on page 18.)

[125] T. K. Lakshman and Unday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp*, pages 202–217, 1991. (Cited on page 22.)

[126] Stephan Kenji Lerche. Kontextverwaltung und die erweiterung einer pointcut-sprache um eine objektorientierte kontextanfrage basierend auf rdf modelliertem kontext. Diploma thesis, University of Bonn, August 2009. (Cited on page 55.)

[127] Karl J. Lieberherr. *Adaptative Object-Oriented Software: The Demeter Method*. PWS Publishing, 1996. ISBN 053494602-X. (Cited on page 30.)

[128] John W. Lloyd. *Foundations of Logic Programming*. Second Edition, Springer-Verlag, 1987. (Cited on pages 19, 20, and 37.)

[129] Lunjin Lu. Logic program analysis engine download page. URL `http://www.secs.oakland.edu/~l2lu/software.html`. (Cited on pages 115 and 124.)

[130] Lunjin Lu. Abstract interpretation, bug detection and bug diagnosis in normal logic programs. PhD thesis, University of Birmingham, 1994. URL `http://www.cs.waikato.ac.nz/~lunjin/PHD.ps.gz`. (Cited on pages x, 9, 10, 28, 39, 40, 43, 47, 49, 51, 53, 115, 147, and 155.)

[131] Lunjin Lu. A mode analysis of logic programs by abstract interpretation. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Ershov Memorial Conference*, volume 1181 of *Lecture Notes in Computer Science*, pages 362–373. Springer, 1996. ISBN 3-540-62064-8. URL `http://dx.doi.org/10.1007/3-540-62064-8_30`. (Cited on pages 28, 29, and 37.)

[132] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with FLORID: a deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998. (Cited on page 143.)

[133] Frank Manola, Eric Miller, and Brian McBride. Rdf primer. w3c recommendation. *World Wide Web Consortium*, 2004. (Cited on page 63.)

[134] Kim Marriott and Harald Søndergaard. Bottom-up abstract interpretation of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 733–748. MIT Press, 1988. (Cited on page 28.)

[135] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4:258–282, April 1982. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357162.357169. URL `http://doi.acm.org/10.1145/357162.357169`. (Cited on page 20.)

[136] Glen McCluskey. Using java reflection. URL `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`. (Cited on page 135.)

[137] Ted McFadden, Karen Henricksen, and Jadwiga Indulska. Automating context-aware application development. In *UbiComp 1st International Workshop on Advanced Context Modelling, Reasoning and Management, Nottingham*, pages 90–95, 2004. (Cited on page 96.)

[138] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000. ISSN 0098-5589. doi: 10.1109/32.825767. URL `http://portal.acm.org/citation.cfm?id=331520.331527`. (Cited on page 2.)

[139] Gregor Meyer. *On Types and Type Consistency in Logic Programming*. PhD thesis, FernUniversität Hagen, 1999. Published as volume 235 of DISKI, Akademische Verlagsgesellschaft Aka, Berlin, 2000. http://www.fernuni-hagen.de/pi8/typical. (Cited on pages 9 and 23.)

[140] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of squishql, a simple rdf query language. *The Semantic Web-ISWC 2002*, pages 423–435, 2002. (Cited on page 144.)

[141] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. (Cited on page 23.)

[142] Holger Mügge, Tobias Rho, and Armin B. Cremers. Integrating aspect-orientation and structural annotations to support adaptive middleware, workshop on middleware-application interaction (mai'07), in conjunction with the second european conference on computer systems (eurosys.07). Lisbon, Portugal, March 2007. (Cited on page 8.)

[143] Holger Mügge, Tobias Rho, Daniel Speicher, Pascal Bihler, and ArminB. Cremers. Programming for context-based adaptability — lessons learned about oop,soa, and aop. SAKS Workshop in conjunction with GI/ITG-Tagung Kommunikation in verteiltenSystemen, March 2007. (Cited on page 125.)

[144] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artif. Intell.*, 23(3):295–307, 1984. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(84)90017-1. (Cited on pages 22 and 23.)

[145] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th Int. ACM Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, March 2006. (Cited on page 145.)

[146] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen. Creating semantic web contents with Protégé-2000. *Intelligent Systems, IEEE*, 16(2):60–71, 2001. (Cited on page 143.)

[147] John O'Conner. Using java db in desktop applications. URL `http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javadb/`. http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javadb/. (Cited on page 107.)

[148] M. O´Connor, H. Knublauch, S. Tu, B. Grosof, M. Dean, W. Grosso, and M. Musen. Supporting rule system interoperability on the semantic web with swrl. *The Semantic Web–ISWC 2005*, pages 974–986, 2005. (Cited on page 143.)

[149] Martin O'Connor and Amar Das. Sqwrl: a query language for owl. In *OWL: Experiences and Directions (OWLED), Fifth International Workshop*, 2009. (Cited on page 143.)

[150] Martin O'Connor, Ravi Shankar, Csongor Nyulas, Samson Tu, and Amar Das. Developing a web-based application using owl and swrl. In *AAAI Spring*, 2008. (Cited on page 143.)

[151] Yoosoo Oh, Jonghyun Han, and Woontack Woo. A context management architecture for large-scale smart environments. *Communications Magazine, IEEE*, 48(3):118–126, 2010. (Cited on page 144.)

[152] Richard A. O'Keefe. Finite fixed-point problems. In *ICLP*, pages 729–743, 1987. (Cited on page 52.)

[153] Richard A. O'Keefe. The Craft of Prolog, 1990. (Cited on pages 21 and 52.)

[154] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3):54–62, 1999. (Cited on page 2.)

[155] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re) shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001. (Cited on page 30.)

[156] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972. URL `http://www.acm.org/classics/may96/`. (Cited on page 8.)

[157] Bijan Parsia and Evren Sirin. Pellet: An owl dl reasoner. In *Third International Semantic Web Conference-Poster*, 2004. (Cited on page 18.)

[158] Adrian Paschke. A typed hybrid description logic programming language with polymorphic order-sorted DL-typed unification for semantic web type systems. *CoRR*, abs/cs/0610006, 2006. URL `http://arxiv.org/abs/cs/0610006`. informal publication. (Cited on pages 26 and 143.)

[159] Jason Pascoe. Adding generic contextual capabilities to wearable computers. In *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, pages 92–99. IEEE, 1998. (Cited on page 4.)

[160] Renaud Pawlak, Laurence Duchien, Gerard Florin, and Lionel Seinturier. Jac: A flexible solution for aspect-oriented programming in Java. *MetaLevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, 2001. (Cited on pages 8, 136, 145, and 148.)

[161] Filip Perich, Anupam Joshi, Timothy Finin, and Yelena Yesha. On data management in pervasive computing environments. *Knowledge and Data Engineering, IEEE Transactions on*, 16(5):621–634, 2004. (Cited on page 7.)

[162] Frank Pfenning. *Types in logic programming*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-16131-1. (Cited on pages 9, 22, 23, 25, and 26.)

[163] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002. ISBN 1-58113-469-X. doi: http://doi.acm.org/10.1145/508386.508404. (Cited on page 8.)

[164] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109. ACM Press, 2003. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643614. (Cited on pages 136 and 148.)

[165] Georg N. Prezerakos, Nikolaos D. Tselikas, and G. Cortese. Model-driven composition of context-aware web services using contextuml and aspects. 2007. (Cited on page 3.)

[166] Eric Prud hommeaux. SPARQL 1.1 Federation Extensions, June 2010. URL `http://www.w3.org/TR/sparql11-federated-query/`. (Cited on page 142.)

[167] Eric Prud'Hommeaux and Andy Seaborne. Sparql query language for RDF. *W3C working draft*, 4(January), 2008. (Cited on page 141.)

[168] Mario Pukall, Grebhahn, Christian Kästner, Walter Cazzola, and Sebastian Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–3, 2011. (Cited on page 8.)

[169] Roland Reichle, Michael Wagner, Mohammad Ullah Khan, Kurt Geihs, Massimo Valla, Cristina Fra, Nearchos Paspallis, and George A. Papadopoulos. A context query language for pervasive computing environments. In *5th IEEE Workshop on Context Modeling and Reasoning (CoMoRea) in conjunction with the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom)*. IEEE Computer Society Press, 2008. URL http://www.vs.uni-kassel.de/publications/2008/RWKGVFPP08. (Cited on pages 4, 6, 7, 91, 96, 143, and 147.)

[170] Raymond Reiter. *On closed world data bases*. Plenum Press, New York, 1978. (Cited on page 18.)

[171] Tobias Rho and Guenter Kniesel. Jtransformer 2.9 java pef specification, . URL https://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/2.9/java_pef_overview. (Cited on page 119.)

[172] Tobias Rho and Guenter Kniesel. Prolog development tools, . URL https://sewiki.iai.uni-bonn.de/research/pdt. (Cited on page 123.)

[173] Tobias Rho and Guenter Kniesel. Uniform genericity for aspect languages, technical report iai-tr-2004-4, computer science department iii, university of bonn. In *Uniform Genericity for Aspect Languages, Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn*. Dec 2004. URL http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/RhoKniesel-IAI-TR-2004-4.pdf. (Cited on page 130.)

[174] Tobias Rho, Mark Schmatz, and Armin Cremers. Towards context-sensitive service aspects. In *Workshop on Object Technology for Ambient Intelligence and Pervasive Computing, in conjunction with 20th European Conference on Object Oriented Programming (ECOOP 06), July 3-7, Nantes, France*. July 2006. URL http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/RhoSchmatz-OT4AmI06.pdf. (Cited on pages 3, 99, and 147.)

[175] Tobias Rho, Malte Appeltauer, Stephan Lerche, Armin B. Cremers, and Robert Hirschfeld. A context management infrastructure with language integration support. In *In Proceedings of the Workshop on Context-oriented Programming (COP) 2011, co-located with ECOOP 2011*. ACM DL, Lancaster, UK, July 2011. (Cited on pages 97, 125, and 147.)

[176] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Record*, volume 23, pages 442–453. ACM, 1994. (Cited on pages 18 and 143.)

[177] Neal Sample, Dorothea Beringer, Laurence Melloul, and Gio Wiederhold. CLAM: Composition language for autonomous megamodules. In Paolo Ciancarini and Alexander L. Wolf, editors, *Proceedings of Coordination '99*, volume 1594 of *LNCS*, pages 291–306, 1999. (Cited on page 6.)

[178] Marco L. Sbodio and Wolfgang Thronicke. Ontology-based context management components for service oriented architectures on wearable devices. In *Industrial Informatics, 2005. INDIN'05. 2005 3rd IEEE International Conference on*, pages 129–133. IEEE, 2005. (Cited on page 55.)

[179] Bill N. Schilit and Marvin M. Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994. (Cited on page 4.)

[180] Mark Schmatz. Generische kontext-sensitive Aspekte für Service-orientierte Architekturen. Diploma thesis, University of Bonn, 2007. (Cited on pages 99, 100, 118, 119, and 147.)

[181] Andy Seaborne and Steve Harris. SPARQL 1.1 query. W3C working draft, W3C, oct 2009. http://www.w3.org/TR/2009/WD-sparql11-query-20091022/. (Cited on page 142.)

[182] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96–101, 2006. (Cited on page 16.)

[183] Michael Sintek and Stefan Decker. TRIPLE - A query, inference, and transformation language for the Semantic Web. *The Semantic Web-ISWC 2002*, pages 364–378, 2002. (Cited on page 143.)

[184] 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, 2007. doi: doi:10.1016/j.websem.2007.03.004. (Cited on pages 26, 141, and 143.)

[185] Jan G. Smaus, Francois Fages, and Pierre Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, pages 214–226, 2000. (Cited on page 25.)

[186] Jan-Georg Smaus. *Modes and Types in Logic Programming*. PhD thesis, University of Kent at Canterbury, Germany, 1999. (Cited on page 25.)

[187] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. Owl web ontology language reference. http://www.w3.org/TR/owl-ref, Feb. 2004. World Wide Web Consortium (W3C) recommendation. (Cited on page 144.)

[188] Gert Smolka. *Logic programming over polymorphically order-sorted types*. PhD thesis, Universität Kaiserslautern, Germany, 1989. (Cited on page 23.)

[189] Zoltan Somogy, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996. (Cited on pages 9, 22, 24, 25, and 26.)

[190] Daniel Speicher, Tobias Rho, and Günter Kniesel. Jtransformer - eine logikbasierte infrastruktur zur codeanalyse. In *Workshop Software-Reengineering, WSR*, pages 2–4, 2007. (Cited on pages 119 and 123.)

[191] Olaf Spinczyk, Andread Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60. Australian Computer Society, Inc., 2002. (Cited on page 30.)

[192] Renganathan Sundararajan and John S. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In R. Shyamasundar, editor, *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'92 (New Delhi, India, December 18-20, 1992)*, volume 652 of *LNCS*, pages 203–216. Springer-Verlag, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong-Barcelona-Budapest, 1992. URL `http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=652&spage=203`. (Cited on page 28.)

[193] Michael Sutterer, Olaf Droegehorn, and Klaus David. User profile selection by means of ontology reasoning. In *Telecommunications, 2008. AICT'08. Fourth Advanced International Conference on*, pages 299–304. IEEE, 2008. (Cited on page 141.)

[194] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM, 2003. (Cited on pages 8 and 145.)

[195] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.v35:8. (Cited on page 128.)

[196] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002. (Cited on page 7.)

[197] Eric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-Aware Aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006) LNCS, Springer-Verlag.*, March 2006. (Cited on pages 6 and 145.)

[198] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. (Cited on page 28.)

[199] Hong-Linh Truong and Schahram Dustdar. A survey on context-aware web service systems. *International Journal of Web Information Systems*, 5:5–31, 2009. (Cited on pages 3 and 6.)

[200] W.T. Tsai, Chun Fan, Yinong Chen, Raymond Paul, and Jen-Yao Chung. Architecture classification for soa-based applications. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, page 8 pp., april 2006. doi: 10.1109/ISORC.2006.18. (Cited on page 3.)

[201] Herma van Kranenburg, Mortaza S. Bargh, Sorin Iacob, and Arjan Peddemors. A context management framework for supporting context-aware distributed applications. *Communications Magazine, IEEE*, 44(8):67–74, 2006. (Cited on page 141.)

[202] Max Völkel. RDFReactor - From ontologies to programmatic data access. In *Poster Proceedings of the Fourth International Semantic Web Conference*, 2005. (Cited on page 55.)

[203] w3c. XML Schema Part 2: Datatypes Second Edition, 2001. URL `http://www.w3.org/TR/xmlschema-2/`. (Cited on pages 16 and 56.)

[204] Xiaohang Wang, Jin Song Dong, ChungYau Chin, SankaRavipriya Hettiarachchi, and Daqing Zhang. Semantic space: An infrastructure for smart spaces. *IEEE Pervasive Computing*, 3:32–39, 2004. ISSN 1536-1268. doi: http://doi.ieeecomputersociety.org/10.1109/MPRV.2004.1321026. (Cited on pages 4, 91, 96, and 144.)

[205] Xiaohang Wang, Daqing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using OWL. In *PerCom Workshops*, pages 18–22. IEEE Computer Society, 2004. URL `http://csdl.computer.org/comp/proceedings/percomw/2004/2106/00/21060018abs.htm`. (Cited on pages 91 and 96.)

[206] David H. D. Warren. Implementing Prolog — compiling logic programs. D.A.I. Research Report 39, 40, University of Edinburgh, 1977. (Cited on page 24.)

[207] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991. (Cited on page 3.)

[208] Jan Wielemaker. An optimised semantic web query language implementation in Prolog. *Logic Programming*, pages 128–142, 2005. (Cited on page 18.)

[209] Jan Wielemaker, Michiel Hildebrand, and Jacco van Ossenbruggen. Using Prolog as the fundament for applications on the semantic web. In S. Heymans, A. Polleres, E. Ruckhaus, D. Pearse, and G. Gupta, editors, *Proceedings of the 2nd Workshop on Applications of Logic Programming and to the web, Semantic Web and Semantic Web Services*, volume 287 of *CEUR Workshop Proceedings*, pages 84–98. CEUR-WS.org, 2007. (Cited on page 115.)

[210] Guizhen Yang, Michael Kifer, Hui Wan, and Chang Zhao. Flora-2: User's manual. *Department of Computer Science, Stony Brook University, Stony Brook*, page 49, 2002. (Cited on page 143.)

[211] Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *CoopIS/DOA/ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 671–688. Springer, 2003. ISBN 3-540-20498-9. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2888&spage=671. (Cited on pages 18 and 143.)

[212] Eyal Yardeni, Thom W. Frühwirth, and Ehud Y. Shapiro. Polymorphically typed logic programs. In *Types in Logic Programming*, pages 63–90. 1992. (Cited on page 25.)

[213] Andreas Zimmermann, Andreas Lorenz, and Marcus Specht. Applications of a context-management system. In *CONTEXT 2005*, pages 556–569, 2005. (Cited on page 91.)

NOMENCLATURE

Context Management Infrastructure

AOP Aspect-oriented Programming

CMI Context Management Infrastructure

CMS Context Management System

$CSLogicAJ$ Context-sensitive Service-oriented Logic Aspects for Java

$OCQL$ Object-oriented logic Context Query Language

$OSGi$ Open Services Gateway initiative

QCS Query Context Sources

Prolog

$\mathcal{G}$ Goal

$\mathcal{P}$ Program

ϕ Formula

A Atom

B Positive Literal

Cl Clause

L Literal

$SLDNF$ Selective Linear Definite clause resolution with Negation as Failure

t Term

V Logic Variable

RDFS Java Mapping

Id Legal Java identifiers

JI Java Interfaces representing classes from $RNVT$

JI_b Java interfaces with a one to one mapping to RDFS classes

JI_e Powerset-extended set of mapped Java types

$JPTW$ Java primitive type wrappers

JT Java Types

LVT RDFS literal value types

RC All subclasses of rdfs:Resource

$RNVT$ RDF non-literal classes

RS RDFS schemas

XDT XML Schema datatypes

INDEX

Abstract Domain, **27**
abstract domain, 28
Abstract Semantics, 27, 43
Abstract Substitution, **28**, **37**, 37
Actuator, 4
Aliasing, 28, **37**, 37
All-Solution Predicate, 21
Aspect, 33
Aspect-Oriented Programming, 30
Aspect-Oriented Software Development, 30
Asynchronous Advice, 103
Atom, 19

Backward Chaining, 18
Base Class, 30
Blank Node, **16**
Bundle, 13

ClientService, 100
Closed World Assumption, 18
Collecting Semantics, 26, 42
Component Types, 74
Compound Term, 19
Concrete Semantics, 26, 41
Context Class, 57
Context Management Infrastructure, 91
Context Management System, 91
Context Pointcut Language, 103
Context Predicate, 64
Context Sources, 4
Context-Oriented Programming, 3
Context-Query Languages, 6
COP, 3
CQL, 7
Crosscutting Concerns, 7, 30
CSLogicAJ, 103
CWA, 18

Declarative Services, 14
Ditrios, 99
Dynamic Residue, 31
dynamism, 2

EBNF, 64
Entailment, 17

First-Order Logic, 19
FOL, 19
Forward Chaining, 18
Functor, 19

Importing Context, **57**

Join Point, 30
Join Point Model, 7, **30**
JTransformer, 123

last.fm, 4
Literal, 16

Named Pointcut, 32
Negation a Failure, 20
Negation as Failure, 18
Normal Logic Programs, 19, 37

Object-oriented logic Context Query Language, 64
OCL, 64
Open World Assumption, **18**
OSGi, 13
OWA, 18
OWL, 18

Pointcut, 9
Predicate Mapping, 68
Program Eements Facts, 119
Program Graph, **40**, 43
Program Point, 37
Program points, 40
Prolog, 19
Proper Subtype Relationship, 72
Proxy indirection, 100

Query Context Source, 70

RDF, 16
RDF Schema, 17
RDFS, 17
Resource Description Framework, **16**

Semantic Web, 4, **16**
Sharing, 28, **37**, 37
SLDNF, 20
Synchronous Advice, 103

Tabling, 18
Term, 19
Turtle, 16
Type Compatible, 75

Unified Resource Identifier, **16**
URI, **16**

VSLDNF, 39