Logic Metaprogramming Framework for Java

Doktori (Ph.D.) értekezés

Espák Miklós

Témavezető: Dr. Fazekas Gábor

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika és Számítástudományok Doktori Iskola

Debrecen, 2009

Logic Metaprogramming Framework for Java

Doktori (Ph.D.) értekezés

Espák Miklós

Témavezető: Dr. Fazekas Gábor

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika és Számítástudományok Doktori Iskola

Debrecen, 2009

Ezen értekezést a Debreceni Egyetem Természettudományi Doktori Tanács Matematika és Számítástudományok Tudományok Doktori Iskola *Informatika* programja keretében készítettem a Debreceni Egyetem természettudományi doktori (PhD) fokozatának elnyerése céljából.

Debrecen, 2009. április 18.

Espák Miklós
doktorjelölt

Tanúsítom, hogy Espák Miklós doktorjelölt a fent megnevezett Doktori Iskola *Informatika* programjának keretében irányításommal végezte munkáját. Az értekezésben foglalt eredményekhez a jelölt önálló alkotó tevékenységével meghatározóan hozzájárult. Az értekezés elfogadását javasolom.

Debrecen, 2009. április 18.

Dr. Fazekas Gábor
témavezető

Logic Metaprogramming Framework for Java

Értekezés a doktori (Ph.D.) fokozat megszerzése érdekében
az informatika tudományágban

Írta: Espák Miklós okleveles programtervező matematikus

Készült a Debreceni Egyetem
Matematika és Számítástudományok Doktori Iskolája
(Informatika programja) keretében

Témavezető: Dr. Fazekas Gábor

A doktori szigorlati bizottság:

elnök: Dr. Sztrik János
tagok: Dr. Kozma László
Dr. Várterész Magda

A doktori szigorlat időpontja: 2008. szeptember 18.

Az értekezés bírálói:

Dr. .....
Dr. .....
Dr. .....

A bírálóbizottság:

elnök: Dr. .....
tagok: Dr. .....
Dr. .....
Dr. .....
Dr. .....

Az értekezes védesének időpontja: 200.....

Contents

Chapter 1

Introduction

Software systems are extensively used in many fields of life today. These systems are usually extremely complex, and developing them needs a project team with high expertise in both software technology and the application domain. During the development, (among others) the business logic (concepts, processes, etc.) of the system have to be mapped to software artifacts. The mapping is not easy, since the systems of notions of the two fields are very different. On one hand, you have to extract the relevant concepts of the domain and reveal their relationships, which needs deep understanding of the field. On the other hand, you have to represent these concepts and their interrelationships with the means of the software model.

During a software development process, many models are developed from the high-level conceptual models to the low-level implementation models, describing numerous parts of the world being modelled. The main artifact produced in the course of the project is a computer program. Computer programs themselves can also be regarded as models, being a (hopefully valid) abstraction of the application domain.

Modelling a problem in a programming language requires you to think in the concepts of the language primarily, and to translate all the elements and relationships of the domain into the language. This requires much intuition, and we cannot expect that the process can be automatized ever. However, a programming language determines how one can think of a problem, and this is usually more constraining than the way you would express things in common sense. Moreover, the difficulty of modelling a particular problem in miscellaneous languages may be very different.

During the last half century, a myriad of *high-level* programming languages have been developed, and this is still an active field of research. The languages can be categorized into *programming paradigms* based on the target of the abstraction. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components [1]. They determine which aspects of the system appear in the program as individual program elements. The different programming paradigms require very different ways of thinking about the problem. None of them can be regarded as superior to the other definitely. Actually, it depends on the type and complexity of the problem, which approach is the most suitable for it. There exist also *multiparadigm programming languages* that allow you to code various parts of the program along different paradigms.

Another obstacle of turning one's thoughts into a computer program easily is that programming languages differ very much from native languages both in their semantics and their syntax. (The semantics is closely related to the programming paradigm.) So, you have to learn to "speak" one or more programming languages "natively" so that you can write programs which are correct, concise, evolvable and reusable as well. In turn, there are *controlled natural languages* that are easy to understand and use even without a long learning curve. However, these are not used in general purpose programming languages now.

1.1 Goals

In this dissertation I present a multiparadigm programming system. The system has four levels. The base level contains the integration of Prolog and Java. These languages are probably the most widespread representatives of *logic programming* and *object-oriented programming* languages, respectively. Two such systems will be proposed. The *Japlo* language is a lingual symbiosis of these languages that extends the Java language with new syntactic elements for supporting logic programming [21]. The *Prolog4J* framework, however, provides a convenient interface over the tuProlog system [14] without the extension of Java. The remaining parts of the dissertation have been built upon Prolog4J.

Prolog is not a typed language, while Java is strongly typed. Moreover, as Java is an object-oriented language, complex type hierarchies can be constructed in it. This makes it more difficult to integrate the languages because the conversion of data between Prolog and Java is not obvious. For this reason, I present a type system for Prolog that allows programming in an object-oriented style.

I implemented the system as a tuProlog library and it is called OOLibrary. It is part of the Prolog4J framework.

Based on Prolog4J and OOLibrary, I introduce a *declarative metaprogramming* (DMP) framework that stores the metainformation of a program in a Prolog database, and provides an easy-to-use object-oriented interface for Java programs to examine it. To retrieve the required *metainformation*, only a query has to be formulated, which specifies the properties of the required data and logical relationships between them. Within the framework the body of methods can be achieved in the form of a Prolog term so that they can be transformed with the means of Prolog. Moreover, the metainformation can be extracted from the bytecode, not the source, which allows to use the framework in a dynamic way. This DMP framework is the second level of the system.

The next goal of the dissertation was to place an *aspect-oriented programming* (AOP) framework upon the aforementioned DMP framework. The AOP framework is the third level of the system. It makes it possible for the programmer to refer to events of the execution of a program and makes certain subprograms run automatically when these events occur (*implicit invocation*). The system relies on the Metadata Facility [5] provided by Java 5 [40] to denote metadata by affixing special annotations to the program elements. The descriptions that pick out the points of the program flow (called *pointcuts*) can be formulated as Prolog queries on the code. Besides of the predicates exploring the program code, spatial and temporal relations between events can also be expressed.

As this AOP framework relies on bytecode manipulation, aspects can be inserted in the program at load-time or even run-time. Although run-time *weaving* provides high flexibility, it does not decrease the performance of the program necessarily but in some cases it may also improve it [20].

Finally, a special feature of this AOP framework is a controlled natural language interface, which hides the embedded logic programming framework completely, and allows the programmer to refer on the events using simple English phrases. I will show that this technique does not only make writing pointcuts more intuitive, but the natural language phrases are easier to read and write, and they are often more concise than the equivalent variants in current aspect-oriented systems. Moreover, it hides the Prolog background definitively, saving the programmer from getting familiar with a different kind of programming language.

The structure of the thesis is the following. In Chapter 2 I introduce several programming paradigms, and discuss various research projects that are either directly used in my system or they reflect the current state of the research in

this field. Chapter 3 and 4 are about the integration of logic programming facilities into Java. At first, the Prolog4J framework is presented in Chapter 3 that contains the specification of OOLibrary in Section 3.1. The introduction of the Japlo language is in Chapter 4. In Chapter 5 and 6 I discuss the declarative metaprogramming library and the aspect-oriented programming framework for Prolog4J, respectively. The presentation of the controlled natural language interface can be found in Section 6.3. Finally, I summarize the results of my dissertation.

Chapter 2

Programming paradigms

In the first part of this chapter I introduce some programming paradigms and techniques briefly. After that I present some related works that served as my motivation to develop a multiparadigm programming framework and build a new logic metaprogramming framework upon it so that I can construct an expressive and easy-to-use aspect-oriented framework for Java.

2.1 Overview

Programming paradigms reflect different ways of thinking about a system to be modelled. It depends on the problem, which approach provides a better solution. Moreover, none of these approaches can be regarded as superior to the others. Also, it is hard even to define when a program can be considered as *better* than another one.[1] There are many aspects: conciseness, readability, flexibility, reusability, efficiency and so on. In this section I will discuss several programming paradigms and techniques. (This selection is far from being complete, and not all of its elements can be regarded as paradigms, rather they are programming techniques, styles or approaches. I choose only those which appear in my multiparadigm programming system.)

[1] It is easy to define when a program is good: if it produces the expected output for any input allowed. For hard real-time programs it is also critical that the program should respond within a predefined time interval.

2.1.1 Imperative programming

The oldest paradigm of high-level programming languages is called *imperative programming*. An imperative program is made up of statements that must be performed in a certain order: sequential execution is the default, but there are special statements, which influence the control flow (*jumps*). Imperative programs use *variables* for storing the results of computations. You can assign values to variables and read them arbitrary times. Imperative programming is closely connected with the Neumann-architecture [43].

Imperative programming is a general category, and incorporates several paradigms, like *structured programming*, *procedural programming* and *modular programming*. Structured programming states that both the data and the code should have a hierarchical structure. A few control statements (blocks, branches and loops mainly) are defined, which can be nested in each other arbitrarily, and other kind of jumps (especially backward GOTO) are considered harmful.

Procedural programming allows to define *subprograms* or *subroutines*, which certain consecutive parts of the program can be extracted into. These subprograms represent computations (*functions*) or activities (*procedures*), and can be applied by *calling* or *invoking* them.

The idea of *modular* programming is the grouping of programming constructs that represent the same concern of the system into distinct *modules*. The coherence between the program elements within a module has to be high, while the modules have to be loosely coupled. It is important to note that this is a basic, general principle of programming that can be applied not exclusively for imperative programs.

Of course, structured, procedural and modular programming are not exclusive categories: current imperative languages support each of them. Using imperative programming, a problem can be solved by specifying every single step of the solution. This does not require any *"intelligence"* from the computer (run-time environment), it only has to execute the statements in the order defined by the program and the semantics of the language. This is a primarily computer-centric way of thinking. In spite of this, imperative languages are very popular, the vast majority of computer programs are coded according to this paradigm.

2.1.2 Declarative programming

Declarative programming is the other pole of programming paradigms vis-á-vis imperative programming. In contrast with the latter one, declarative program-

ming focuses on *what* the program should accomplish, rather than specifying *how* to go about accomplishing it. This approach is a more human-centric way of thinking, therefore it needs some intelligence to be integrated into the run-time environment of programs. Declarative programming languages are categorized into the following classes: *functional programming languages*, *logic programming languages* and *domain-specific languages*.

In functional programming the computations (functions) are in the centre of the abstraction. Although this seems to be an algorithmic approach, it differs from imperative programming in several ways. At first, pure functional languages do not allow *side-effect*, which means that functions can only use their formal arguments and the return value of other functions to carry out their computation, and they cannot modify their environment. There are no global variables or output formal arguments through which functions could communicate. At second, many functional languages (mostly those based on ML) use rewrite rules to expand function applications, to construct the code of the program this way.

Functional programming is ideal for describing transformation of data. Another advantage is that several functional languages have mathematical foundation (λ-calculus, π-calculus and other calculi), defining their formal syntax and semantics, and sophisticated ways of proving the correctness of functional programs have been elaborated. A disadvantage is that –because of the lack of side-effects– the change of the state of objects cannot be coded directly: the object has to be passed to a function, and it has to return a new one with the new state. This is rather circuitous on certain fields, like I/O or GUI programming.

Another branch of declarative programming is *logic programming*. Instead of describing the algorithm for solving a problem, logic programs describe only logic relationships of the problem, and it is the task of the runtime environment to find solutions for a query based on this high-level logic representation and the embedded theorem prover.

The best known declarative programming language is Prolog. Prolog programs operate on a single *uniform database*. This database stores *facts*, which represent relationships between objects, and *rules*, which are used to state contingent facts [45]. Based on this simple relational database sophisticated queries can be composed in a very flexible way, and –thanks to the algorithms embedded– the Prolog engine finds every solution very efficiently. Based on this inherent way of finding solutions for an adequately declared problem, Prolog turned out a success on several fields, among others the following ones:

- *Intelligent systems* perform useful tasks by utilizing artificial intelligence

techniques.

- *Expert systems* are intelligent systems which reproduce decision-making at the level of a human expert.
- *Natural language systems* can analyse and respond to statements made in ordinary language as opposed to approved keywords or menu selections.
- *Relational database management systems* may store huge amount of data, structured in the quite simple way of mathematical relations. Prolog can be used to obtain data specifying their properties (internal relationships) instead of specifying the path (navigation) they can be accessed.

Domain-specific languages (DSL) are *not* general purpose programming languages, rather they provide means to describe problems of a specific application domain. Domain-specific languages has been used for almost as long as computing has been done. DSLs are very common: examples include regular expressions, cascading style sheets (CSS), makefiles, SQL, HQL or markup languages. [22]

DSLs are typically declarative. For example, markup languages serve for describing documents, without determining the algorithm how they have to be displayed. (That is why markup languages are simple to use, but at the same time they cause many compatibility problems for different types and versions of web browsers.)

2.1.3 Object-oriented programming

The idea of *object-oriented programming* (OOP) originates from simulation environment. In OOP, the task of a program is carried out by the communication of *objects*. Objects are individual, they have a unique identity called *object identifier* (OID). Objects also have a *state*, which describes the value of their *attributes* or *fields* at a given moment of the execution, and they have *behaviour*, which describes how they react on messages sent to them by other objects. The messages are realized by *methods*, which are subprograms essentially, and they represent the behaviour of objects. This way OOP combines the data model (describing the structure of the state) and the procedural model (describing the behaviour), which are inseparable.

Another key concept of OOP is that objects are categorized into a taxonomy called *class hierarchy*. A *class* describes the *type* of similar objects, having the same set of fields and the same behaviour. Moreover, there is an inheritance

relationship between classes, which makes it simpler to define new classes based on existing ones by *extending* them with new fields or new methods. Although these new classes (*subclasses*) inherit the behaviour (as well as the fields) of the extended class (*superclass*), they can even redefine some of its methods, changing this default behaviour.

The first object-oriented languages (SIMULA-67, SmallTalk) were based on structured languages, but the concepts described above are general enough to allow declarative object-oriented languages to be implemented. Some examples are CLOS (the Common Lisp Object System), Objective Caml, Haskell and LogTalk.

OOP spread widely in a relatively short period in the second half of the 80's (the most influential languages were C++ and SmallTalk), and then in the 90's the object-oriented approach appeared in software modelling as well. Nowadays, the object-oriented paradigm is the most important paradigm in software development, and it is applied at almost every phase of the software life-cycle. It is extensively used in enterprise applications and provide solution for describing the business logic, the user interfaces, persistence and so on. Some of the most significant languages of these fields are Java, C♯ and C++.

The success of OOP is due partly to the high-level of reusability of classes, partly to the fact that it grasps certain important aspects of the we think about the world that structured programming cannot: especially the integrity of data and code, and the possibility of defining generalization or specialization relationships between concepts.

2.1.4 Metaprogramming

Metaprogramming is not a general-purpose programming paradigm, rather it is a technique which allows the programmers to handle (i.e. examine, manipulate or construct) a program as such (called *base level* program or *base program*) from a new, higher dimension, called *meta-level* program or *metaprogram*). This way the elements of the base program can be referred to from the metaprogram as *metaobjects*. In object-oriented environment the type of metaobjects are called *metaclasses*. The term *reflection* means the construction of metaobjects, representing (reflecting) certain elements of the base program. Depending on whether only static metainformation are available about the data structures of the program, or there are metainformation about its code, we talk about *structural* or *behavioural* reflection, respectively. On the contrary, *reification* means the process of realizing the effect of manipulation of metaobjects on the base level.

The base program and the metaprogram can be written in the same language, or even they can be parts of the same program. A *metaobject protocol* (MOP) of a language specifies what kind of metainformation can be accessed, and how it can be used to analyse or modify the base level program. In other words, MOPs are interfaces to the language that give users the ability to incrementally modify the language's behaviour and implementation, as well as the ability to write programs within the language. In this way a MOP allows users to adjust the language to better suit their needs. [33]

The Java language has limited support for reflection. Only structural reflection is supported, and the structure of classes can only be introspected but not modified. Since the late 90's several MOPs have been elaborated for Java. I discussed them in my master's thesis [18], in which I also presented a new metaobject protocol for Java called Soul.

Metaprogramming is usually not intended for *"end user programmers"*. Typical examples of metaprograms are compilers, interpreters, debuggers but metaprogramming is extensively used in component-oriented technologies as well (e.g. application servers, graphical user interface builders, UML editors, etc.), since it makes it possible to manage software components automatically based on the metainformation extracted from them.

Programming languages which allow representing programs as their first-class entities (like LISP or Prolog) are of particular interest with respect to metaprogramming, since it is quite easy to create programs in them, which execute another program of the same language. Such programs are called *meta-circular interpreters* or *metainterpreters* [27, 44].

2.1.5 Declarative metaprogramming

Declarative metaprogramming is defined as the use of a declarative programming language for writing metaprograms. Declarative languages are very suitable for writing metaprograms because they allow the programmer to focus on what to achieve rather than how to achieve it. One typical task of metaprograms is to look up parts of the program being processed. Using the imperative approach this "search" could be done by navigating through metaobjects representing the elements of the program, examining one by one whether the search criteria applies for them. However, this path can change as the program evolve, which can result in the metaprogram having to be adopted to it. The quest would be easier if you could define *"what"* you are looking for separately from *"how to"*, which is a basic idea of logic programming.

Another typical task of metaprograms is to transform the processed program,

which is also easier when it is not necessary to specify the exact way of applying the transformation. Consequently, functional programming is a good choice to achieve this.

The project underlying this thesis was mostly inspired by declarative metaprogramming (DMP) [37]. DMP states that declarative languages are the most suitable for reasoning about the metainformation of programs. Registering metainformation into a Prolog database makes it easier to find difficult relationships within a program [36, 49]. Typically, the most natural way to organize *metaclasses* of a language (classes of *metaobjects* storing metainformation) is a tree. Although metaobjects can be represented as Prolog terms, but –because of the lack of support for inheritance– there is no object polymorphism between terms representing metaobjects. A class hierarchy definition would allow that rules which accept terms (objects) of a class, also accept objects of any subclass of the class automatically.

2.1.6 Aspect-oriented programming

One of the key issues of designing computer programs is called *separation of concerns* (SoC). It expresses that the program should be modularized in such a way that program elements representing the same *concern* of the problem should be placed in the same module[2], and one module should represent only one concern in an ideal case. Otherwise, the code implementing a concern will be *scattered* through several modules, and the code of various concerns will be *tangled* within a single one.

The aim of *aspect-oriented programming* is to separate such *crosscutting concerns*. To achieve this it suggests to extract the scattered code representing a single concern into a distinct module. This special module is usually called *aspect*. Additionally, you have to specify the places where this separated code and the original one (cleaned up from the tangled concerns) should be joined together. The moments of execution when the control is passed over between the code of concerns are called *join points*. The set of join points can be defined by *pointcuts*. Aspects may have special subprograms called *advices*, which cannot be invoked directly, but are activated at the join point (the join point *fires*). This is called *implicit call*. In contrast with ordinary subprograms, the places of the invocation of advices are specified by the advices themselves by *pointcut designators*.

[2]Here the term module is used in a broad sense. Now "module" can be any distinct part of the program, e.g. even a class or a method.

From a technical point of view AOP languages use a special technique called *weaving* to assemble the final program from the separated modules. Weaving can happen at almost any phase of the life cycle of programs. According to that there is compile-time, load-time or run-time weaving and so on.

There are several, conceptually different AOP languages and frameworks. In some of them crosscutting concerns can be separated into co-ordinate modules [48]. In others, like AspectJ –and this is the usual– there is a distinction between the base program and the aspects. This approach is very similar to that of MOPs. Indeed, aspects can be simulated by metaobjects that redefine the semantics of method invocations so that they can perform some additional activity when they are invoked. I discussed the relationship between AOP and MOPs in [19]. Another work about this topic can be found in [52].

It is a very critical issue of AOP systems that how precise the set of the join points can be described. As you have to designate the points in the program execution, it is obviously an issue of metaprogramming. It has been showed that simply enumerating places in the code causes that the pointcut will be *fragile*, which means that refactoring the base code involves that the pointcut designator has to be modified as well [31]. To enhance the reusability of aspects, pointcuts have to be described in a more robust way, by the properties of the join points. Logic metaprogramming –as discussed above– provides powerful techniques for this.

Another critical issue of aspect-oriented systems is when weaving is performed. Compile-time AOP systems require the source code of both the aspects and the base program to compose them by weaving producing either source files (precompilation) or object files.

Weaving can also be performed by modifying the object code, turning implicit calls into explicit ones. The transformation of the object code is called *instrumentation*. Weaving through instrumentation makes is possible at later phases of the program, like postcompilation, load or execution. In general, weaving at a later time makes the system more dynamic and flexible. Load-time or run-time weaving allows you to weave (or even *unweave*) aspects for classes which were not available at the time of launching the application (possibly loaded through the network), or just whose source code is not available. Run-time AOP systems support e.g. *fix-and-continue* debugging, and make it possible to modify the behaviour of the program without restarting it. Although load-time and run-time weaving causes some time overhead since the object code has to be instrumented during the execution, weaving is usually done only once during the life-cycle of a class. Moreover, it has been shown that run-time weaving does not decrease the efficiency of the composed program inevitably but it

can even improve that [20].

AOP has been popular within the object-oriented paradigm, although the idea behind it is general enough to make it possible to apply it for other kind of languages as well. AOP is typically used for separating codes responsible for non-functional requirements like logging, persistence, security and so on.

2.1.7 Multiparadigm programming

As it was emphasized previously, the strength of the individual programming paradigms and techniques appear in different fields. None of the general purpose paradigms can provide ideal solution for any class of problems. Hence a lot of research effort has been invested in combining them by creating programming environments which do not constrain programmers to solve every part of the problem using a single paradigm. The goal of *multiparadigm programming* is to allow this.

The simplest form of multiparadigm programming is the use of a framework, which allows programs written in different programming languages to communicate with each other. However, there exist several multiparadigm programming languages, which incorporate programming constructs supporting several paradigms. A good example is C++, which was originally designed to be an object-oriented extension of the structured language C, but supports generic programming and metaprogramming through templates as well [2]. Although C++ is definitely a multiparadigm language, it is an imperative language at the same time and does not have support for declarative programming at all.

As already discussed, a declarative programming language is ideal for metaprogramming, however, it may not always be the most suited language for other parts of the program. For example, it is better to write graphical user interfaces in an object-oriented language. The concept of "linguistic symbiosis" means a mechanism to allow easy integration of declarative and non-declarative parts of a program. This way the metaprogramming facilities of a (possibly non-declarative) language could be accessed through a declarative interface.

The aim of this work is to introduce a multiparadigm programming framework, which combines the object-oriented language Java with logic programming (Prolog). Based on this framework a declarative metaprogramming library is introduced, which serves as the foundation of an aspect-oriented extension of Java.

2.1.8 Controlled natural languages

Controlled natural languages (CNL) are not a programming paradigm, but only a class of languages. They are subsets of natural languages with a restricted vocabulary and a narrow set of grammar rules. Controlled language texts can be read just as easy as natural languages, but they are unambigous, which is an important factor for processing them by computers.

Controlled natural languages are used successfully and actively in industry for specific application domains [50]. Also, there are standards of controlled subsets of English for decades [47, 24]. Another excellent example for CNLs, is the translation of the OMG's Semantics of Business Vocabulary and Business Rules into English [28]. Although the "official" definition is in XML, the appendices contain it in a controlled language form, which is much easier to read for humans.

There are attempts to use controlled natural languages in general purpose programming languages as well. There is a preliminary work about *naturalistic programming* written by Lopes et al. [35]. The paper emphasizes that it is only a suggestion for moving beyond aspect-orientation. The most important issue of naturalistic programming is to blur the barriers between the way we think about things and how they can be modelled within the current programming paradigms. It says that the natural way of thinking can be approximated by examining natural languages for finding lingual constructs which allow one to express their thoughts in a more concise way.

Anaphoras are such things. In current programming languages every element of a program must be denominated if they need to be referred to from another part of the program. In natural languages, however, we do not give names for every single thing we are talking about. Rather, we use anaphoras for referring to things just mentioned before.

Lopes et al. suggest in [35] to use a controlled natural language for writing programs. Here, "controlled" means that the language follows strict grammar rules for avoiding ambiguity. The language they suggest is a minimal subset of English. This solution raises some concerns, though. Rewriting an algorithm into this language, preserving the original structure of the code, results in a very verbose text, which would rather decrease the readability of the program instead of improving it. However, using anaphoras, the local variables can be eliminated, resulting a code being concise and quite easy to read, especially for domain experts not familiar with programming languages.

Regarding our field, metaprogramming could be considered as such an application domain. As already discussed, metaprogramming can be done best in

a declarative way, which is an inherent property of natural languages. Moreover, in aspect-oriented programming there is a special, very restricted use of metaprogramming for specifying pointcuts. Even, the structure of pointcut designators is quite simple, which would make it relatively easy to define a controlled natural language interface upon them.

2.2 Integration of Prolog and Java

Because of the success of Prolog on several areas (shown in 2.1.2), innumerable attempts have been made to combine it with other languages. The most simple tools are the foreign language interfaces (FLI) [51]. FLIs make it possible to evaluate a Prolog goal from another language, and sometimes also allow to run foreign language subprograms from rules. However, these interfaces are very austere, enforcing the programmer to do a lot of boiler plate coding to initialize the Prolog engine, to create the terms which can be processed by it, to get it produce the solutions of a query, and to transform the solutions back to the objects of his/her language. FLIs are not ideal at all for multiparadigm programming, as they provide only a pure, raw interface towards the guest language. Moreover, they are usually implemented using native methods, which decreases portability [51].

In this section I introduce some current multiparadigm frameworks that support interaction between Java and Prolog. Since a partial goal of the dissertation is to support declarative metaprogramming for Java, I have selected those which would be suitable for such a system. The most important issues for selecting one of them were the followings:

- Bidirectionality: the Prolog code should be able to call Java and vice versa.
- Pure Java: the code should not depend on platform specific libraries.
- Prolog terms can be created programmatically.
- Project has to be actively developed.
- It should be open source and free, at least for research purposes.

There are several projects which fulfil some of these requirements, but only a few that satisfies each of them. I excluded the following projects:

- JPL: It is the Java interface of SWI-Prolog [54]. Although SWI-Prolog is one of the most widespread Prolog environments, there is not much activity on the development of its Java interface. It is not pure Java, Prolog code can be invoked through a foreign language interface (FLI), so native libraries have to be carried with the application. [51]
- jProlog: Sources of Java classes have to be generated from a Prolog source if you want to use their predicates from Java. It is a very static approach. [13]
- Jinni: It is the Java interface of BinProlog. Jinni is not simply a Java programming interface but also allows object-oriented programming in Prolog. Unfortunately, Jinni is not a free software: it is neither free of charge, nor is its source code open. [9]

After investigating a couple of such projects, I found two that fulfil these requirements: tuProlog [14] and JLog [29]. In the following sections I will give a short introduction to both.

2.2.1 tuProlog

tuProlog [14] is a lightweight Prolog implementation written in Java, which provides bidirectional Prolog/Java integration. Since tuProlog represents terms as Java objects, goals can be constructed at run-time, and then they can be get solved by the Prolog engine. tuProlog makes it also possible to write Prolog libraries in Java. A library can define a theory (a set of static facts and rules as a string) and primitive predicates, functors and directives as well. These primitives can be defined by Java methods following some simple conventions. One of the predefined libraries (`alice.tuprolog.JavaLibrary`) allows to access Java objects from Prolog.

If you would like to solve a goal which depends on values computed at run-time then you have to construct the term representing the goal using the provided classes: Int, Long, Float, Double, Struct and Var. (Atoms are regarded as compounds with zero argument, and are also represented as `Struct` objects.) Of course, the results of the goal will be represented also by these types. Fig. 2.1 illustrates the use of tuProlog. The `main` method creates a Prolog engine and inserts the facts and rules of the classical Prolog sample problem into the its database. The `isMortal` method takes a name as its argument and returns whether he is mortal or not. The `getMortals` method collects all mortal beings into a `List` and returns it.

```
package test;
import java.util.*;
import alice.tuprolog.*;

public class MortalTest {
  private static Prolog prolog = new Prolog();
  public static boolean isMortal(String somebody) {
    Term goal = new Struct("mortal", new Struct(somebody));
    SolveInfo sol = prolog.solve(goal);
    if (sol.isSuccess())
      return true;
    prolog.solveHalt();
    return false;
  }
  public static List<String> getMortals() throws PrologException{
    Term goal = new Struct("mortal", new Var("Somebody"));
    List<String> mortals = new ArrayList<String>();
    SolveInfo sol = prolog.solve(goal);
    while (sol.isSuccess()) {
      Term sb = sol.getTerm("Somebody");
      mortals.add(((Struct) sb).getName());
      if (!sol.hasOpenAlternatives())
        break;
      sol = prolog.solveNext();
    }
    return mortals;
  }
  public static void main(String[] args) throws PrologException {
    prolog.addTheory(new Theory(
        "mortal(X) :- human(X).\n" +
        "human(socrates).\n" +
        "human(plato).\n"));
    System.out.println(isMortal("socrates"));
    System.out.println(getMortals());
  }
}
```

Figure 2.1: Using tuProlog

As it can be seen, facts and rules can be inserted dynamically, and also a query can be constructed and solved at run-time. However, the use of this API is not concise at all: you must handle or declare checked exceptions, and transform the objects representing terms back to the type of your "application domain" (maybe an exaggerating attribute now) by hand for further processing.

It is an additional inconvenience that tuProlog does not support generics, for-each loops, auto-boxing or varargs (deliberately, for compatibility with Java 2). These language features introduced in Java 5 would allow additional static type checking, and they would make the transformation and traversing the solutions a bit more convenient. These issues are discussed by Cimadamore and Viroli [8], introducing the *P@J* framework, which will be discussed in Section 2.2.3. Among other improvements, they suggest a redesigned, generic class hierarchy of terms to allow static type checking.

The use of the system I suggest is intended to be even more clear: by creating a mapping between Prolog and Java types, the inner class hierarchy of terms may be hidden from the programmer completely. The user of the framework can pass plain old Java objects (POJOs) to Prolog queries, and also gets back POJOs as a result (if any). Conversions happen automatically in the background in both directions. This framework is called *Prolog4J* and will be discussed in Section 3.2.

2.2.2 JLog

JLog [29] is another implementation of a Prolog interpreter, written in Java. It is suitable for developers who need an embedded Prolog engine in Java. In JLog it is easy to consult, construct queries, and evaluate query results. It also includes translation facilities to map between Prolog terms and standard Java objects.

Fig. 2.2 shows how the example of Fig. 2.1 can be programmed using JLog. As it can be seen, the values of input variables of a goal can be passed to the query as a `Hashtable` object, in which the keys are the name of the variables. Similarly, the bindings of the individual solutions of a query can also be obtained as `Hashtable` objects.

Note that you need not convert the result, this task is taken by the `jTermTranslation` class. This class aggregates `iObjectToTerm` and `iTermToObject` converters into a single conversion class which chooses the correct converter object for the for desired conversion. The fundamental idea behind the converter mappings is that typically each Prolog term object has a single corresponding Java object, and each Java object has a single corresponding Prolog term. For

each conversion direction (i.e., object to term and term to object) there is a default converter, used if the not matching converters are found in the look-up table. [29]

```
public class JLogTest {
  private static final jPrologAPI prolog = new jPrologAPI(
      "mortal(X) :- human(X).\n" +
      "human(socrates).\n" +
      "human(plato).\n");

  public static boolean isMortal(String somebody) {
    Hashtable bindings = new Hashtable();
    bindings.put("X", somebody);
    Hashtable varz = prolog.queryOnce("mortal(X).", bindings);
    prolog.stop();
    return varz != null;
  }
  public static List<String> getMortals() {
    List<String> mortals = new ArrayList<String>();
    Hashtable varz = prolog.query("mortal(X).");
    while (varz != null) {
      mortals.add((String)varz.get("X"));
      varz = prolog.retry();
    }
    prolog.stop();
    return mortals;
  }
  public static void main(String[] args) {
    System.out.println(isMortal("socrates"));
    System.out.println(getMortals());
  }
}
```

Figure 2.2: Using JLog

2.2.3 P@J

The *P@J* framework [8] is intended to provide an easy to use and type-safe Java framework for enhancing interoperability with Prolog. P@J is based on tuProlog 2.0. The idea behind it is very similar: to evaluate a Prolog goal, an annotated method has to be declared, which will serve as an interface be-

tween the languages. P@J defines its own type hierarchy over tuProlog's one (Fig. 2.3). The main advantage of this hierarchy is that it is generic, i.e. you can specify typed Prolog lists and variables. This allows to make queries in a type-safe way, eliminating the need for type casts. The conversion between the P@J and tuProlog representation is done automatically, and is called *marshalling/unmarshalling*. There are methods also for converting P@J terms to plain Java objects and back.

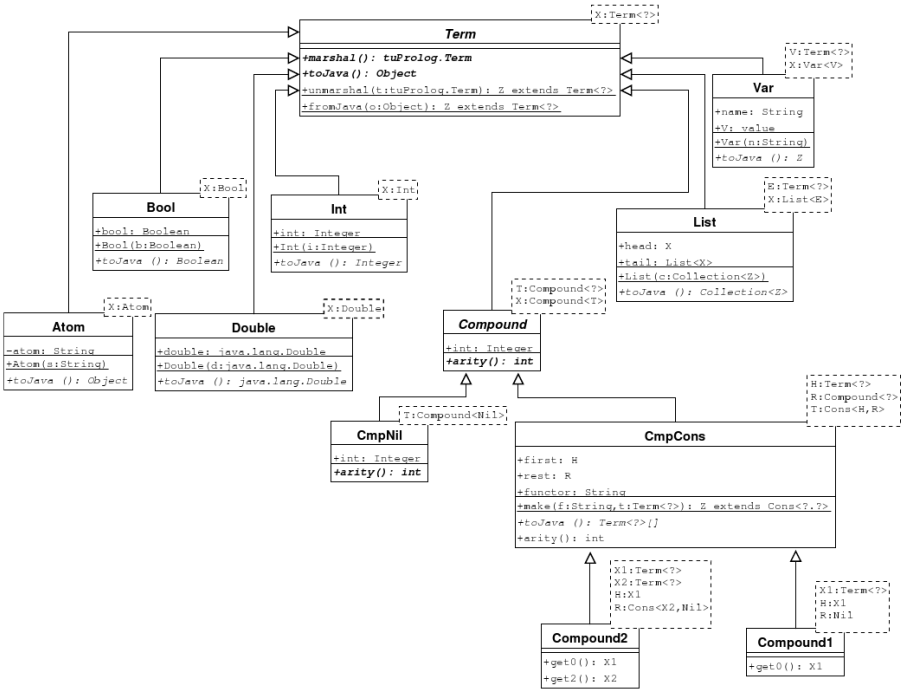

Figure 2.3: P@J type hierarchy

Although the type hierarchy of P@J allows type-safe programming, it is more complicated compared to tuProlog. While tuProlog 2.0 has only six types for representing terms (including abstract classes), P@J has twelve. Additionally these types represent a new layer between the plain Java types and the tuProlog types that causes twice as many conversions.

P@J uses annotations for specifying rules. The rules themselves can be defined using the `@PrologClass` class annotation, while the queries on them

can be given using `@PrologMethod` annotated methods of such classes. Fig. 2.4 shows how the `isMortal` and `getMortals` methods of Fig 2.2 can be written using P@J. The numbers of the formal arguments ($1, $2, ...) can be used as variables in the goal. If a Prolog method has more output arguments and/or just you would like to get them together with the input arguments then the solutions are returned as compounds, where the arguments of the compounds are the elements of the solution. Annotating a formal argument with `@HIDE` causes that the values bound for that variable will not be wrapped into the compound.

```
@PrologClass(clauses={
    "mortal(X) :- human(X).",
    "human(socrates).",
    "human(plato)."})
public abstract class MortalUtility {
  @PrologMethod (link="mortal($1)",
                 multipleOutput=false,
                 style=PrologInvocationKind.BOOLEAN)
  abstract @GROUND boolean isMortal(@INPUT @GROUND Atom c1);
  @PrologMethod (link="mortal($0)",
                 multipleOutput=true,
                 style=PrologInvocationKind.FUNCTIONAL)
  abstract @GROUND Iterable<List<Int>> getMortals();

  public static void main(String[] args) throws Exception {
    MortalUtility mu = Java2Prolog.newInstance(MortalUtility.class);
    System.out.println(isMortal(new Atom("socrates")));
    System.out.println(getMortals());
    Var<List<Atom>> x = new Var<List<Int>>("X");
    List<Int> list = Term.fromJava(c);
    for (List<Int> compound: mu.getMortals()) {
      Collection<Integer> ci = compound.get0().toJava();
      System.out.println(ci);
    }
  }
}
```

Figure 2.4: Using the P@J framework

P@J also supports defining term classes by the `@Termifiable` annotation. The objects of such classes are automatically converted to terms and vice versa through JavaBean introspection. Regarding the example of

Fig. 3.12, the object `new Human("homeros")` would be converted to the term `'Human'(property('name', 'homeros'))`. In contrast with the solution I proposed in the current paper, this term cannot be substituted by the term equivalent of an instance of a subclass of `Human`, because the type hierarchy is not available on the Prolog side. As it can be seen, similarly to the current work the OID is also not reserved.

An advantage of P@J is that its implementation does not rely for a special compiler. It does not generate any code. Instead, `@PrologClass` classes and `@PrologMethod` methods are abstract, and an instance for such classes can be achieved by a special factory class.

2.3 Declarative metaprogramming frameworks for Java

In the current section I introduce some projects from the field of declarative metaprogramming. This does not mean that only these are relevant for the goals of this dissertation. Rather, DMP (or metaprogramming in general) is usually a foundation of frameworks or tools, for example in aspect-oriented programming. So other projects using DMP will be discussed in Section 2.4. The relation between AOP and DMP is discussed there.

2.3.1 TyRuBa

TyRuBa is a logic metaprogramming system for generating Java source code. Its name comes from "Type Rule Base" that implies that it supports type-oriented logic programming.

The syntax of TyRuBa is very similar to the syntax of Prolog, but there are some differences as well, for better support for code generation. For example, there are quoted terms, for specifying Java source code, and the name of TyRuBa variables starts with '`?`' and they can be used in quoted terms as well. Compound terms can be written using '`<`' and '`>`' signs instead of the usual parentheses.

Fig. 2.5 summarizes the types of terms. In contrast with Prolog, there is a new kind of term, called *tuple*[3]. As it can be seen, compound terms can have different types. Note that there is a general type called `Object`, which is a

[3]Indeed, tuples are just as the same as compound terms where only the arguments are of interest and the name of the functor is indifferent.

Kind of term	*Term*	*Type of term*

constant	`abc`
	`100`
	`123.123`
	`abc::Foo`
		`String`
		`Integer`
		`Float`
		`Foo`

compound terms	`person<John, Smith>`	`person(String, String)`

lists and pairs	`[]`
	`[1, 2]`
	`[1, foo]`
	`[[],[1],[a]]`
	`[foo<abc>]`
		[*any type*]
		`[Integer]`
		`[Object]`
		`[[Object]]`
		`[foo<String>]`

tuples unbound variable	`<John, Smith>` `?x`	`<String, String>` `?x`

Figure 2.5: Built-in types of TyRuBa

supertype of all other types. You can also define new types based on existing ones.

In TyRuBa rules and facts have to be declared. The declaration specifies the type of the formal arguments. Additionally, the *modes* of the rule or fact has also to be declared, which describe whether the actual arguments have to be bound or not (B means 'bound' and F means 'free'), and the possible number of the results in each case (`DET` mode allows exactly one result, `SEMIDET` 0 or 1, `NONDET` 0 or more and `MULTI` 1 or more). A simple example of a declaration is shown on Fig. 2.6. It is the declaration of the `livesIn` facts, which have two arguments storing who lives where. The definition of the facts follows the Prolog way.

```
livesIn :: String, String
MODES
  (B,F) IS SEMIDET
  (F,B) IS NONDET
END
```

Figure 2.6: An example for declaration of facts

2.3.2 JQuery

JQuery is a source code browser based on TyRuBa, for Java projects developed with Eclipse. Using it you can create queries on the source of a Java project, and display the matches in a tree view.

JQuery stores the abstract syntax tree (AST) of the project source code as TyRuBa facts. The AST is retrieved from Eclipse JDT (Java Development Tool). Newer versions of JQuery also processes the bytecode of classes for generating facts. JQuery defines a type hierarchy of metaclasses using the TyRuBa type system. The type hierarchy is illustrated in Fig. 2.7. The type hierarchy is decomposed into facts and stored in the database in this way.

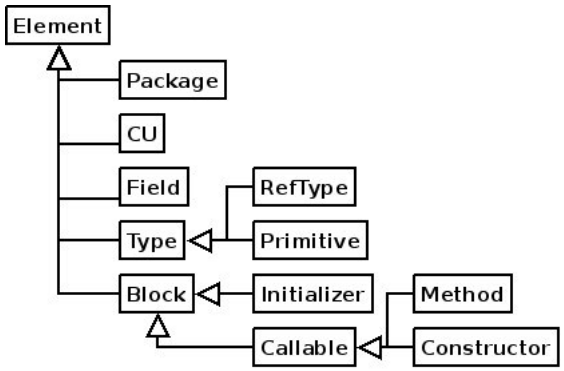

Figure 2.7: JQuery metaclass hierarchy

JQuery provides predicates for accessing the program elements. Predicates are classified into two categories: *core fact predicates* stores the structure of the program directly, while *derived predicates* are defined by rules representing queries on the facts. Thanks to the type system of TyRuBa, the type of formal arguments of both core facts and derived predicates is declared.

Unfortunately, JQuery does not support the features of Java 5, like annotations, generics or enumerations.

2.4 Aspect-oriented programming languages

2.4.1 AspectJ

AspectJ [32] is one of the first AOP languages, and it is still popular today. Although AspectJ's approach is not the one and only for AOP, many aspect-oriented concepts appeared in this language first. The language is regarded as a reference point for other AOP languages and frameworks, which usually characterize themselves against AspectJ.

AspectJ defines the notion of join points as well-defined points in the program flow. A set of join points is called *pointcuts* that pick out certain join points and can also denominate elements of the context at the join point. Pointcuts can be specified using *pointcut designators* (PCD) that are Java-like logical expressions made up of simpler pointcut designators. The whole list of *primitive pointcut designators* (PPCD) is summarized in [10].

Special subprograms called *advice* can be defined that are assigned to a pointcut, and will be invoked when the execution reaches a join point that fits on the pointcut. There are five types of advices that specify when the advice should be run related to the join point: *before*, *after*, *after returning*, *after throwing* and *around*. "After" advices run after that the activity at the join point has been performed, either it has returned properly or was interrupted because of an exception. "Around" advices run at the place of the join point (instead of it), and the join point itself can be executed by a `proceed()` invocation from within the advice.

Pointcuts can also be used for describing the place where an aspect can introduce new fields or methods to the base program. This technique is called *intertype declaration*.

The following example stems from the AspectJ Programming Guide [12]. The example and its variants are frequently used by other AOP implementations as well, to illustrate their use or to demonstrate some of their advantages against AspectJ.

The features are presented using a simple figure editor system. A Figure consists of a number of FigureElements, which can be either Points or Lines. The Figure class provides factory services. There is also a Display that is responsible for displaying figureelements.

A critical issue of aspect-oriented systems is how sensible the aspects are on the evolution of the base program. The evolution of the base program may require the modification of an aspect in two cases: at first when the aspect calls a method of the base program whose signature changes, at second when the change

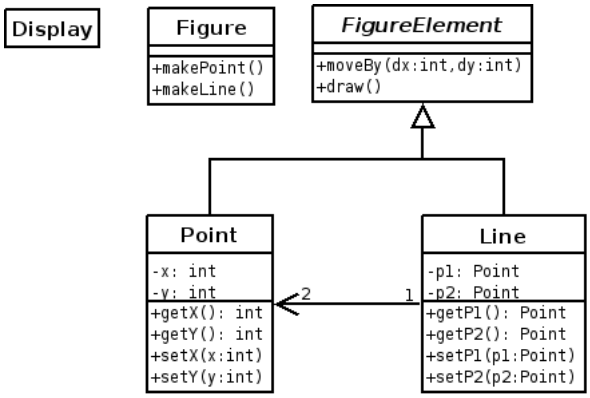

Figure 2.8: UML class diagram for the figure editor example

affects the pointcuts designating the interface between the base program and the aspects' code. The first case is not an issue of aspect-oriented programming, and can be handled by performing refactorings of the code just as by normal object-oriented programs [23].

The second case, however, is an important problem of AOP systems. A pointcut is said to be *fragile* when the change of the base program causes that the pointcut does not determine the proper set of join points any more. This is in contrast with the concept of abstraction and reusability which concepts are about making a code more compact and encapsulating the individual concerns of a program into distinct, reusable modules, which are formed in such a way that internal change of a module should not effect other modules at all.

Fig. 2.9 shows an example for an aspect which is responsible for refreshing the display whenever the position of a figure element changes. As it can be seen, the pointcut designator refer on specific fields of the base code. These fields are private fields of their class, not being parts of the public programming interface of the class. This harms the concept of information hiding, and makes the pointcut fragile, since renaming these fields results that the pointcut becomes invalid. (AspectJ gives compilation error, fortunately.) Instead of referring to the private fields directly, you could refer to `moveBy` and the setter methods. However, the pointcut remains still fragile in this way as well.

So, specifying pointcuts by enumerating elements of the program is not suitable for keeping aspects flexible against changes. AspectJ also allows to specify

```
aspect DisplayUpdating {
  pointcut move():
    set(int Point.x) ||
    set(int Point.y) ||
    set(Point Line.p1) ||
    set(Point Line.p2);

  after() returning: move() {
    Display.update();
  }
}
```

Figure 2.9: The DisplayUpdating aspect with enumeration pointcut

program elements by a pattern containing wildcards or (from AspectJ 5) annotations. The problem with wildcards is that they refer to program elements by their name that is a syntactical issue and has nothing to do with the semantics. An example for the use of wildcards in pointcuts is illustrated in Fig. 2.10.

```
aspect DisplayUpdating {
  pointcut move():
    call(void FigureElement.moveBy(..)) ||
    call(void FigureElement+.set*(..));

  after() returning: move() {
    Display.update();
  }
}
```

Figure 2.10: The DisplayUpdating aspect with wildcard pointcut

Annotations are intended to allow you to assign metadata to program elements, which metadata describe some semantic details of the element (e.g. their role). The use of annotations in PCDs can also arise several problems. At first, advice pointcuts refer to events of the program flow, which happens inside of method bodies. Statements or expressions, however, cannot be annotated. You can, though, annotate the element which is accessed at the event (e.g. "reading of a field annotated by A").

This is a perfect solution if the aspect represent a non-functional requirement. (This is a the most typical use of AOP now.) For functional requirements, like updating the display in this example, using annotations is rather harmful,

since the program elements could be selected based on their use, what is already hard-wired in the code. Considering this example, it is obvious that the changes of those fields of figure elements which never read during `Display.update()` are totally irrelevant regarding the refreshment of the display, so it is not necessary to inform the aspect about their changes. This information, however, could be extracted from method bodies, so annotating the fields would be redundant and could lead to inconsistency. Moreover, it would require additional administration from the programmer.

We can state that the means of AspectJ to express pointcuts for functional requirements are not flexible with respect to the evolution of programs. More flexibility could be achieved when instead of denominating the elements of the program, they could be selected based on their use. This would need that they should be able to be referred to from PCDs by variables so that their relationships can be expressed.

As already mentioned, another critical issue of aspect-oriented systems is when the weaving of aspects is performed. First versions of AspectJ supported compile-time weaving, the current versions weave aspects at load-time through bytecode instrumentation. Run-time weaving is not supported for now.

2.4.2 LogicAJ

LogicAJ is an extension of AspectJ, in which meta-variables can be used within the pointcut designators to refer the elements of the base program [55]. There are two kinds of meta-variables: *logic meta-variables* match exactly one element (their name starts with a single question mark), while *logic list meta-variables* can match several elements of the base program (their name starts with two question marks). As discussed in the previous section in details, the ability to referring to the elements of the base program increases the expressive power of the pointcuts, and makes the aspect code less vulnerable to the structural changes of the program.

LogicAJ is based on JTransformer [4], which project has similar goals as JQuery. JTransformer relies on SWI-Prolog as its Prolog engine. JTransformer processes the source code of a Java program to collect all the (static) metainformation of the program for storing it in the Prolog database. Fig. 2.11 illustrates the structure of the database through an example. As it can be seen, similarly to JQuery, the AST is "flattened", each element of the program are stored as a single fact, whose arguments are atoms. Every metaelement has also a unique identifier. These numeric IDs are used as logical references between the tree nodes. (Prolog does not support physical references like pointers.) In con-

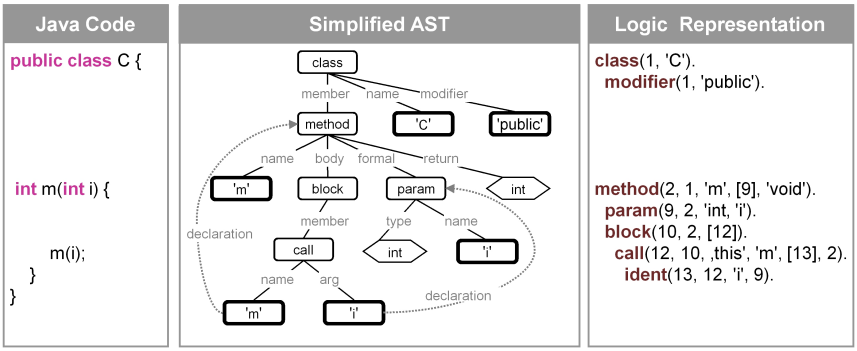

Figure 2.11: Representation of classes in JTransformer

trast with JQuery, JTransformer also supports describing code transformations, called *conditional transformations* [34].

Although LogicAJ relies on Prolog, it remains almost completely hidden from its users. Instead of following the Prolog notation, LogicAJ uses the syntax of AspectJ for describing pointcuts so that it save the programmer working in two different environments at the same time (logic-based and object-oriented).

There is a second generation of the project called LogicAJ2, in which you can specify source code pattern in PCDs. This is similar to the quoted terms of TyRuBa.

As already mentioned, LogicAJ processes the source code of the application so that it can weave aspects at compile-time. This is a static approach, and more flexibility could be achieved by a load-time or run-time weaver. However, this would be a challenge for LogicAJ2, because the method bodies in the bytecode are flattened, so recognizing loops or even iteration on a collection in would not be simple. There are Java decompilers, though, which can accomplish this [41].

Another disadvantage of LogicAJ and JTransformer is that they do not support Java5 yet.

2.4.3 ALPHA

ALPHA is a small, statically typed aspect-oriented language [49]. ALPHA programs are processed by an interpreter written in Java. The language supports a minimal subset of Java, e.g. class definition and single inheritance. It is based on the L2 language [15].

In [49] Ostermann et al. showed that although crosscutting concerns can be separated into distinct modules using e.g. AspectJ, the aspect-oriented solution of a problem performs not much better then the object-oriented one in consideration of program evolution. In case of the examples of Fig. 2.9 and Fig. 2.10, the pointcut designators are sensitive to certain changes of the non-aspect code, e.g. in the following cases:

- Object graph change: Outsource part of the drawing relevant state of a figure element to a class that is not in the FigureElement hierarchy.
- Control flow change: Change the condition under which a display update is necessary.
- Class definition change: Inserting/removing a field whose change does not affect the display.

This kind of changes results that the pointcuts have to be modified reflecting the change. To cope with the problem, [49] provides different models of program semantics: the abstract syntax tree (AST), the execution trace, the heap and the static type assignment.

Pointcuts can be specified using Prolog queries. A query is made up of primitive queries, connected by *and* (“,” in Prolog). Primitive queries operate on one of the Prolog databases representing the individual models of the program. Since the whole structure of the program is stored in a Prolog database (AST model), instead of referring to program elements by enumerating their concrete name (or by using wildcards), you can specify them by describing their use in the program. For example, you can express such queries like “setting fields which are accessed from the `DisplayUpdate.update()` method” using the formalism of Prolog.

Using the other databases, even more complex queries may be done. The database of the execution trace allows you to reason about the time order of join points. For example, you can examine whether an event has been happened since another one. Regarding the figure editor example, you can express the condition of updating the display as “modifying a field of a figure element that was read last time when the display was updated”. Defining the pointcut in this way, it will not break when new fields are introduced in `FigureElement`, either they are used when refreshing the display or not.

However, the PCDs still may need to be updated when the values used by `Display.update` are not stored in the fields of `FigureElement` directly, but in the fields of their fields, for example. Using the database of the heap, you can

```
class DisplayUpdate extends Object {
  Display d;
  // enum pointcut
  after set(P, x, _); set(P, y, _); set(P, 'start', _); set(P, 'end', _),
    instanceof(P, 'FigureElement') { this.d.draw(P); }

  // set * pointcut
  after set(P, _, _), instanceof(P, 'FigureElement') { this.d.draw(P); }

  // pcflow pointcut
  after now(ID), set(ID, ExpID1, P, F, _), instanceof(P,'FigureElement'),
    pcflow(Display , 'drawAll', (_, get ((ExpID2, _), F)));
    hastype(ExpID2, 'FigureElement') { this.d.draw(P); }

  // cflow pointcut
  after set(P, F, _), get(T1, _, P, F, _),
    mostRecent(T2, calls (T2, _, @this.d, 'drawAll', _)),
    cflow(T1, T2), instanceof(P, 'FigureElement') { this.d.draw(P); }

  // cflowreach pointcut
  after set(P, F, _), get(T1, _, P, F, _),
      mostRecent(T2, calls (T2, _, @this.d, 'drawAll', _)),
      cflow(T1, T2), reachable(Q, P), instanceof(Q, ' FigureElement') {
    this.d.draw(P);
  }
}
```

Figure 2.12: Pointcut definitions in ALPHA

reason about the association hierarchy between objects, so you can express in your query whether an object is *reachable* from another or not.

Fig. 2.12 shows several examples for the display updating advice. [49]

It is easy to see that the whole execution trace of a program can be very huge, and could cause using up the available memory for long running programs. To avoid this situation, [49] suggests a technique for computing the *shadows* of pointcuts. The shadow of a pointcut is a small set of expressions[4] which may affect the result of a pointcut. Moreover, the *lifetime* of the facts generated at the shadow is also computed. The computation of the shadow and the lifetimes is performed by a Prolog meta-interpreter. When the lifetime of a fact of the

[4]Statements are also stored as expressions in the AST database.

database of the execution trace is over then the fact will be removed from the database.

Since ALPHA is implemented by an interpreter written in Java, the system is completely dynamic. The main disadvantage of ALPHA is that it is a prototype language used only for research purposes, which lacks of many important feature of Java.

Chapter 3

The Prolog4J framework

In Section 2.2 I introduced two Prolog implementations which can be embedded into Java applications in an easy and portable way. These implementations make the interaction possible in both directions between Prolog and Java. Although the communication between the Prolog and Java parts of a program is more fluent than at FLIs, since the type system of the languages is radically different, you still have to convert the objects of your application domain to objects representing Prolog terms and vice versa.

However, an ideal multiparadigm programming framework or language should allow intermixing the constructs of the languages without forcing the programmer to deal with the internal representation of terms in the system. In the current chapter, I propose a multiparadigm programming framework for Java and Prolog, which do not suffer from this weakness.

The framework is called *Prolog4J* and it has two parts. Its Prolog part defines a library called *OOLibrary*, which allows object-oriented style programming in Prolog. The library allows to define a hierarchy of classes. The purpose of this hierarchy is to introduce *object polymorphism* into Prolog: although the instances of classes are ordinary (compound) terms, the instance of a subclass can stand for an instance of one of its superclass at any place of a program. The library is discussed in Section 3.1.

In Section 3.2 I introduce the Java part of the framework that allows running Prolog queries easily from Java. I will show that using the OOLibrary in Prolog, Java types can be mapped into Prolog types, which makes the conversion between Java objects and Prolog terms automatic and natural. Furthermore, using the metadata facility of Java 5, a programming interface can be specified

for accessing Prolog rules from Java, through which logic queries can be done with the easiness of method invocations.

3.1 Object-oriented interface on Prolog

In this section I present a tuProlog library called OOLibrary that support *object-oriented style* programming in Prolog. The main purpose of this library is to introduce type checking and object substitutability into Prolog. It provides a set of predefined types and you can define new types as well. The user-defined types always have a supertype, and any term of a type can be substituted by a term of its subtype, just like objects in object-oriented programming.

Following an established convention in built-in argument template description, which takes root into an imperative interpretation, the symbol `+` in front of an argument means an input argument, `-` means an output argument, `?` means an input/output argument and `@` means an input argument that must be bound.

3.1.1 Type system

In OOLibrary a type can be assigned to any term that will be called its *base type*. Although the mapping from term to its base type is unambiguous, there may also be other types (e.g. derived types, see later), which the term is compliant with. There is also a special type, which any term is compliant with: the type `term`. Any type is a *subtype* of `term` or with other words that `term` is a common *supertype* of types.

Base type of ground terms

The base type of ground atomic (non-compound) terms is a *primitive* or *atomic* type. There are six atomic types: `int`, `long`, `float`, `double`, `atom` and `[]`.

`int` and `long` are signed integer types of size 4B and 8B, respectively. The base type of integer literals within the 4B range is `int` and of those that exceed this range it is `long`. The base type of floating point literals is `double` at default (8B). The type `float` is 4B long. Float literals can be specified by an `f` or `F` suffix (e.g. `5.4f`). The type `atom` is the base type of terms for which the predicate `atom` of ISO Prolog succeeds, except the empty list (`[]`). The base type of the empty list is itself. Moreover, this is the only term, whose base type is `[]`. In contrary, the empty list is regarded as element of any type.

The base type of a compound term is another compound term that can be constructed from the original term by replacing its arguments with their base

```
?- type(udine, T).
T / atom
yes.
?- type(complex(1.2, 3.4), T).
T / complex(double, double)
yes.
?- type([alpha, beta], T).
T / [atom, atom]
yes.
?- type([alpha, beta, gamma], T).
T / [atom, atom, atom]
yes.
?- type([alpha | complex(1, 2)], T).
T / [atom | complex(int, int)]
yes.
?- type(alpha, atom).
yes.
```

Figure 3.1: The type/2 predicate

type, recursively. These types are called *compound types*. With other words, the structure of the base type of a compound term is identical to the structure of the term itself, but the occurrences of its atomic elements are replaced with their (atomic) types. The base type of lists or pairs (e.g. `[a|b]`) can be computed in the same way, since they are also compound terms (e.g. `[a, b, c]` is equivalent with `'.'(a, '.'(b, '.'(c, [])))`). The base type of a term can be constructed by the `type/2` predicate. Its first argument must be a bound term. If the type argument is bound, the predicate checks whether it is the base type of the term. If not, it will be bound to the base type. The usage of the predicate is illustrated in Fig. 3.1. As you can see, using these types two lists of different length will have different base type, even if their elements have the same type. These are called *fixed length list types*.

Base type of nonground terms

Variables may be declared by the `':'/2` infix predicate. The first argument of the predicate must be an unbound variable and the second a type. Declaration is not mandatory. The type of variables depends on whether they are bound and declared. The base type of bound variables is the base type of their value. The base type of declared and unbound variables is the type they have been

```
?- C: complex, type(C, Type).
Type / complex
yes.
?- C: complex, type([x, C], Type).
Type / [atom, complex]
yes.
?- type(C, Type).
Type / term
yes.
?- X: atom, X = 2.
X / 2.
yes.
?- X: atom, X := 2.
no.
?- X: atom, Y: int, X := Y.
no.
```

Figure 3.2: Constructing the base type of nonground terms with the type/2 predicate

declared with. Other variables have the base type `term`.

As well as in the case of ground terms, the base type of a nonground compound term is also another compound term, which can be constructed from the original term by replacing its arguments with their base type, recursively.

The unification predicate (`'='/2`) does not check the type of its arguments. However, there is a new infix predicate for unification with type check: `':='/2`. Just like `'='/2`, this predicate treats its arguments equally as well.

Fig. 3.2 shows examples for declaring variables and constructing the base type of nonground terms.

Type definitions

New types can be constructed from base types by type definitions. They are called *derived types*. Although derived types are not the base type of any ground term, terms may comply with them, and they can be used for declaring variables as well. Type compliance can be checked by the `typeOf/2` predicate, where the first argument is a term, and the second is a type. The second argument has to be bound. The predicate succeeds if the term is the element of the type. There are four kinds of derived types: aliases, generics, union types and classes.

```
?- string: atom.
yes.
?- typeOf(foo, string).
yes.
?- typeOf(complex(1.2, 3.4), complex).
no.
?- complex: complex(double, double).
yes.
?- typeOf(complex(1.2, 3.4), complex).
yes.
?- int_pair: [int | int].
yes.
?- typeOf([5 | 6], int_pair).
yes.
?- atom_list: [atom | atom_list].
yes.
?- typeOf([one, two, three], atom_list).
yes.
```

Figure 3.3: Alias types

Alias types You can introduce a new name for a type by declaring an *alias type* (later on *alias*). Using aliases you can refer to a type with an alternative (maybe simpler or more adequate) name.

Just as variables, aliases can also be declared by the `':'/2` infix predicate. Its first argument must be an atom, which will be a synonym for the type specified by the second argument. Aliases can be declared for base types and derived types as well.

Recursive type definitions are also allowed, where the alias name just being defined appears on the right hand side as well. Using this technique arbitrary length list types or trees can be defined. Fig. 3.3 shows some examples for the use of aliases and arbitrary length list type definitions. Note that although these types are defined recursively, finite terms may comply with them, since `[]` is the element of any type.

Generic types It is possible to define types, which can be parametrized by other types. Such types are called *generic types* or *generics*. Generic type definitions can also be recursive. For example, it is possible to define a generic list type where the concrete type of the elements is not specified but formal type arguments (variables) are used in the type definition.

```
?- list(T): [T | list(T)].
yes.
?- typeOf([], list(term)).
yes.
?- typeOf([alpha, beta], list(atom)).
yes.
?- tree(T): tree(T, tree(T), tree(T)).
yes.
?- int_tree: tree(int).
yes.
?- type([], tree).
yes.
?- typeOf(tree(1, [], []), tree(int)).
yes.
?- typeOf(tree(1, [], tree(4, tree(-4, [], []), [])), int_tree).
yes.
```

Figure 3.4: Using generic types

Generics can also be declared by the `':'/2` infix predicate. Its first argument must be a compound term. Its functor is the name of the generic type, and its arguments must be variables representing the formal type arguments. The type name in its own cannot be used as a type (there are no "raw types" as in Java). When using the type, actual type arguments have to be specified. Fig. 3.4 shows some examples for the definition and use of generics, for example the generic arbitrary length list type. The type `list(T)` is predefined that will be called the *list type* later on.

Union types Type definitions can be overloaded. If you define a type that has already been defined then the new meaning does not override the previous one, instead it extends that. In this way the new type can be regarded as a *union* of these types. In this sense union types are not really a new kind of types but they allow combining several types. There are also some predefined union types, illustrated in Fig. 3.5. (The prompt is not shown for simplicity. All the goals succeed.)

3.1.2 Classes and objects

Beyond the types discussed previously, *classes* can also be defined. Classes may have typed fields, and they must have exactly one (direct) superclass.

```
integer: int.
integer: long.
real: float.
real: double.
number: integer.
number: real.
```

Figure 3.5: Union types

```
person(Name: atom) :: term.
employee(Name: atom, Salary: float) :: person(Name).
employee(Name: atom) :: employee(Name, 5000).
```

Figure 3.6: Class definitions

The superclass may be `term` or an already defined class. `term` is the common superclass of classes.

Classes can be defined by the `'::'/2` infix predicate. Its first argument is a compound term or an atom. In the case of a compound term, its functor denotes the classname, and its arguments are the field declarations. The form of field declarations is the same as of variable declarations described in 3.1.1. The name of the fields is the uncapitalized name of the variables.

If the first argument of `'::'/2` is an atom then the class will not have any fields, and its name will be the atom itself.

The second argument of `'::'/2` specifies an object of the superclass or the class itself in function of the field variables being declared. Fig. 3.6 shows examples for class definitions.

Objects are represented by compound terms, where the functor of the term is the classname and the arguments are the values of the individual fields (the state of the object). The fields of objects can be accessed by `field/3` (template: `field(+Object, @Field, ?Value)`). The predicate can be used both for getting the value of a field and for setting it. Of course, setting a field is possible only if it is still unbound. Fields are inherited.

For convenience, an *accessor method* is generated for each field of a class. (Methods are described in the next section.) The name of the accessor starts with the field name and ends by the `Of` suffix. Accessors has two arguments: the first denotes the object and has to be bound, the second is an input/output argument and denotes the value of the field. Fig. 3.7 shows examples for describing objects and accessing their fields.

Classes are integrated into the OOLibrary type system. Using `typeOf/2` you

```
?- field(employee(john), name, Name).
Name / john
yes.
?- Jack = person(jack), nameOf(Jack, Name).
Jack / person(jack)
Name / jack
yes.
```

Figure 3.7: Accessing object fields

```
?- typeOf(employee(jack, 5000), employee).
yes.
?- typeOf(employee(jack, 5000), person).
yes.
```

Figure 3.8: Checking type of objects

can check whether an object is an instance of a class (Fig. 3.8).

It is important to note that this is not regular object-oriented programming in the sense that instances do not have identity, which would be a key requirement of a real OO system. Without object identifiers (OID) an object referring to itself (maybe indirectly through other objects) can be represented only by cyclic terms. However, cyclic terms are discouraged, because they cannot be converted to Java objects (see Section 3.2).

LogTalk [42] is a full-featured object-oriented logic programming language based on Prolog. Unfortunately it does not work with either tuProlog or JLog. However, this complexity is not needed here, this simplistic model will be sufficient.

3.1.3 Methods

Methods can be defined independently from classes, by the `'::-'/2` predicate. The syntax is similar to the definition of rules, except that the type of the formal arguments has to be declared like in the case of field declarations in class definitions. Fig. 3.9 shows an example for a method definition.

As the effect of a method definition, a rule is generated, which checks the type of its formal arguments dynamically. To simulate overriding, the generated rules are added to the beginning of the database (by `asserta`) that allows for the later defined methods to prevent the application of the already defined ones by a cut. (It is expected that methods having more specific argument types will be defined later.)

```
date: date(int, int, int).
person(Name: atom, BirthDate: date) :: term.
ageOf(P: person, Age: int) ::-
    birthDateOf(P, date(Year, _, _)),
    Age is 2009 - Year.
```

The rule generated from the method above:

```
ageOf(P, Age) :-
    typeOf(P, person), typeOf(Age, int), !,
    birthDateOf(P, date(Year, _, _)),
    Age is 2009 - Year.
```

Figure 3.9: Defining methods

3.1.4 Generic classes

Just like types, classes can also be supplied by type arguments. The formal type arguments are variables, and appear before the field declarations, but in contrast with them, type arguments do not have a type. Fig. 3.10 shows examples for defining generic classes.

```
?- vector(T, Elements: list(T)) :: term.
yes.
?- intVector(Elements: list(int)) :: vector(Elements).
yes.
?- typeOf(vector([1, 2, 3]), vector).
no.
?- typeOf(vector([1, 2, 3]), vector(int)).
yes.
?- typeOf(vector([1, 2, 3]), intVector).
yes.
```

Figure 3.10: Generic classes

3.2 The Prolog4J Framework

This section describes the Prolog4J Framework, using which you can make Prolog queries from a Java program in an easy way. In contrast with most frameworks, which make it possible to call Prolog code from Java, in Prolog4J there

is automatic conversion between Java and Prolog types that makes it possible to use the types of the application domain from both languages.

The first part of this section present the mapping between the Java and Prolog type systems, based on which the type conversions are automatized. Based on these hidden transformations, the programming interface provides an easy to use way to make Prolog queries and traverse through the solutions. This interface and its use is discussed in Section 3.2.2. The third part of the section suggests to extract solving goals into special methods, so-called *goal methods* so that they can be solved by ordinary method invocations. As you will see, the body of goal methods can be generated based on the metadata facilities of Java. Additionally, goal methods make it also possible to exploit the advantages of static type checking. In the next point I will discuss, how Prolog4J can be used with OOLibrary together to implement some methods in Prolog. Finally, some details of the implementation will be discussed.

3.2.1 Automatic conversions

The framework has been designed to serve as an interface to a Prolog engine, allowing you to use the reasoning capabilities of Prolog in the application domain of the program. In accordance with this, it was not a goal to transform arbitrary Java objects to Prolog terms. Instead, only the relevant classes of the application domain are allowed to be converted. The current section specifies the conversion only for the most basic cases. The conversion of other types is presented in Section 3.2.4.

With regard to the backwards transformation, the *value* of any Prolog term can be converted to Java, although there is no Java equivalent of unbound Prolog variables as such. Unbound variables are converted to `null`. Similarly, you can pass `null` if you solve a goal but do not want to bind a value to one of its variables. In case of bound variables, their value is converted.

Strings, primitive values and their wrapper objects are converted to atomic terms in the same way as described in the tuProlog Guide (Section 6.1) [14]. Arrays and instances of `java.lang.List<E>` are converted to Prolog lists. Table 3.1 summarizes how Java values are converted to tuProlog terms. The types of the first column will be called *convertible types* later on. Note that there is an auxiliary class `org.prolog4j.Compound`, using which still arbitrary compound terms can be constructed. However, its use is supposed to be as minimal as possible (see Section 3.2.4 later).

As you can see, the mapping of tuProlog types to Java types is ambiguous. I denoted with italic letters the default target types at backwards conversions.

Java type (java.lang package)	*tuProlog type* (alice.tuprolog package)
`char`, *`String`*	`Struct`
`byte`, `short`, `int`, *`Integer`*	`Int`
`long`, *`Long`*	`Long`
`float`, *`Float`*	`Float`
`double`, *`Double`*	`Double`
`byte[]`, `short[]`, `int[]`, `long[]`, `float[]`, `double[]`, `char[]`, `boolean[]`, `Object[]`, *`java.util.List<E>`*	`Struct`
`boolean`, *`Boolean`*	`Struct` ('true' or 'false')
`org.prolog4j.Compound`	`Struct`

Table 3.1: Type mapping

Atoms (`Struct` objects without arguments) are converted to strings, `Int` objects to `java.lang.Integer` objects. Other `Struct` objects are converted to `List` objects if they represent a Prolog list. Otherwise, an instance of `Compound` will be created.

3.2.2 The Prolog4J API

In this section I introduce the `Prover` and `Solution<S>` classes and the `SolutionIterator<S>` interface. The types belong to the `org.prolog4j` package. The public API of the types is shown in Fig. 3.11.

Solving a query can be initiated by one of the `solve()` methods of the `Prover` object. (The class is singleton.) Their first argument has to be a goal (String). If a goal should be solved for some specific value of its variables, these values can be passed as additional arguments to the method. If you would not like to bind a value to a variable (used for input/output) you should pass `null` to it. `solve()` methods return a `Solution<S>` object where `S` should be the type of the variable occurring at last in the goal. The methods are generic, the actual type argument to be substituted to `S` is supposed to be specified before the method name at invocation.

Using a `Solution` instance you can check whether the query is satisfiable (`isSuccess()`), and you can traverse through the solutions if needed. To make this easy, `Solution<S>` implements the `Iterable<S>` interface so that the values

bound to the last variable can be walked through simply by a *for-each* loop. If you would like to walk through the values of another variable, not the last one, you have to call the `on()` generic method of the solution, which will return another `Iterable` object suitable for that.

Since a query may contain several variables which can get bound during the reasoning, the the solutions provide iterators of type `SolutionIterator<S>`. This interface provides additional methods (`get()`) to access the values of the individual variables by their name. The `Solution` class also define `get()` methods that are supposed to be used if you are interested only in the bindings of the first solution, and do not want to find other solutions.

In some rare cases it may be necessary to pass also the type of a variable to `get()` or `on()`. This is the case when you want to retrieve the value of a list as an array, not as a `java.util.List` object. For this reason these methods have a variant, which can also the type be passed to.

Finally, the `Solution` class also provides methods for collecting all solutions. These methods take the collection(s), which the solutions has to be collected into, as their argument(s). For convenience, there are special methods to collect solutions into a `Set`, a `List` or an array of lists.

Fig. 3.12 shows examples for their use. `member/2` and `append/3` are ISO Prolog predicates defined by tuProlog. The examples are rather simple, they are only intended to illustrate the basic use of the framework. The first two examples are equivalent with the examples of Fig. 2.1. As you can see, this solution is much more concise and elegant then the one shown in Fig. 2.1. You do not have to bother with constructing the query and converting the result back. No type casts are needed. You even do not need to control the process of finding the solutions.

The third example checks for the existence of a solution, binding a list to the second variable occuring in the query. The fourth example iterates over the values bound to `X` in the solutions of the same query. The fifth example appends two lists of humans to another. Finally, the last example creates a list (`L2`) which has to be appended to another list (`L1`) to get a third one (`L12`). In the last two examples there is only one solution, which is a list. So, instead of the outer `for` loop this single solution could have been accessed simply by `get()`. Note that there are no type casts in the examples.

3.2.3 Annotations

The use of the framework can be made even more simple and safe using the metadata facility of Java [5]. Using the annotations presented in this section

```
public class Prover {
  public static Prover get();

  public void addTheory(String... clauses);
  public <A> Solution<A> solve(String goal);
  public <A> Solution<A> solve(String goal, Object... values);
  public <A> Solution<A> solve(String goal, String[] varNames,
                               Object[] values);
}
public class Solution<S> implements Iterable<S> {
  public boolean isSuccess();

  @Override
  public SolutionIterator<S> iterator();

  public <A> Iterable<A> on(final String argName);
  public <A> Iterable<A> on(final String argName, Class<A> type);

  public <A> Iterable<A> get(final String argName);
  public <A> Iterable<A> get(final String arg, Class<A> type);

  public <C extends Collection<? super S>>
  C collect(C collection);
  public void collect(Collection... colls);

  public Set<S> toSet();
  public List<S> toList();
  public List<?>[] toLists();
}
public interface SolutionIterator<S> extends Iterator<S> {
  <A> A get(String argName);
}
```

Figure 3.11: The public interface of the Solution class and the SolutionIterator interface

```
public class Prolog4JTest {
  private static final Prover p = Prover.get();
  static {
    p.addTheory("mortal(X) :- human(X).",\
         "human(socrates).", "human(plato).");  }
  public static boolean isMortal(String somebody) {
    return p.solve("mortal(X).", somebody).isSuccess();
  }
  public static List<String> getMortals() {
    List<String> mortals = new ArrayList<String>();
    for (String s: p.<String>solve("mortal(X)."))
      mortals.add(s);
    return mortals;
  }
  public static void main(String[] args) {
    System.out.println(isMortal("socrates")); // true
    System.out.println(getMortals());      // socrates, plato

    List<String> philosophers = Arrays.asList("socrates", "plato");
    Solution<?> solution =
      p.solve("member(X, List).", null, philosophers);
    System.out.println(solution.isSuccess()); // true

    for (String s: solution.<String>on("X"))
      System.out.println(s);               // socrates, plato

    List<String> h1 = Arrays.asList("socrates");
    List<String> h2 = Arrays.asList("thales", "plato");
    for (List<String> humans:
        p.<List<String>>solve("append(L1, L2, L12).", h1, h2))
      for (String h: humans)
        System.out.println(h);             // socrates, thales and plato

    List<String> h3 = Arrays.asList("socrates", "homeros", "demokritos");
    for (List<String> humans: p.solve("append(L1, L2, L12).", h1,null,h3)
         .<List<String>>on("L2"))
      for (String h: humans)
        System.out.println(h);             // homeros and demokritos
  }}
```

Figure 3.12: Applying rules from Java

the prover does not require to be referred to directly, and some more checks can be performed statically.

Prolog theories can be specified by the `@Theory` annotation. The argument of `@Theory` is an array of strings, the elements of which represent Prolog clauses. The annotation is processed at load time.

As discussed in the Section 3.2.2, Prolog goals can be solved by `Prover`. When a goal has to be solved several times, it is suggested to define a method for that which makes its use even more simpler. The body of such methods can also be generated by specifying the `@Goal` annotation for the method. These methods will be called *goal methods* later on. The goal itself can be defined as the default argument of the annotation (`value`).

Goal methods can expect arguments of convertible types as discussed in Section 3.2.1. The return type of goal methods can be one of the followings:

- `boolean` or `java.lang.Boolean` if you are interested only in the satisfiability of the goal,
- a convertible type if you are interested only in the first solution (or no more solutions are possible) and there is only one output variable whose value important,
- `Solution<S>`, where `S` is the type of an *output* variable. Using the returned `Solution` object, the solutions of a query can be traversed through.

The formal arguments can be annotated by `@In` or `@InOut`, denoting input or input/output arguments, respectively. They can be omitted, the default is `@In`. The name of the variable in the goal, which the formal argument has to be bound to, can be specified as an argument of these annotations. In default, the arguments bind to the variables in the order of their appearance.

The return type (if not `boolean` or `Boolean`) can be annotated by `@Out`[1], which makes it possible to specify the name of the output variable. If not specified, the value of the last variable will be returned.

Fähndrich and Leino showed how an object-oriented language such as Java or C# could be extended with *non-null types* [25]. Non-null types provide a type-based approach to detect possible null pointer violations in code statically at compile time. Java Specification Request 308 (JSR-308) [17] proposes an extension to Java's annotation system that permits annotations to appear on nearly any use of a type. (By contrast, Java SE 6 permits annotations only on

[1] In fact, the annotation belongs not to the return type but the method, although formally it can also be written after the modifiers, directly before the return type.

```
@Theory({
  "remove([X|Xs],X,Xs).",
  "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
  "permutation([],[]).",
  "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs), permutation(Zs,Ys)."})
public class PermutationTest {
  @Goal("remove(X, Y, Z).")
  static List<Integer> remove(List<Integer> list, Integer i) { throw null; }

  @Goal("permutation(X, Y).")
  static @Out("Y") Solution<List<Integer>> perms(@In("X") List<Integer> list){
    throw null;}
  public static void main(String[] args) {
     List<Integer> list = Arrays.asList(1, 2, 3);
     for (List<Integer> li: perms(list))
       System.out.println(Collections.max(li));
  }
}
```

Figure 3.13: Defining goal methods in Prolog4J

class/method/field/variable declarations.) Using the extended syntax of JSR-308 the `@NonNull` annotation can be specified for any reference type. There are prototype implementations available for JSR-308 and the nullness checker [16]. The changes proposed by JSR-308 are planned to be part of the Java 7 language.

The `@NonNull` annotation can be used later (from Java 7 on) to specify *ground* input arguments. This way such situations could be prevented by static type checking where a ground input argument of a goal method would get a `null` value.

The body of rule methods can be arbitrary (e.g. `{ throw null; }`), because it will be replaced during the annotation processing. Fig. 3.13 shows an example for the use of Prolog4J annotations. The example is borrowed [8], the original version is shown in Fig. 3.14. As you can see the Prolog4J version is much more concise, without loosing the expressiveness and type safety of P@J.

At this point it makes sense to specify static goal methods only. However, *term classes*, which are the topic of the next part of the section, may also have non-static goal methods.

```
@PrologClass (
    clauses = {"remove([X|Xs],X,Xs).",
               "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
               "permutation([],[]).",
               "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs), permutation(Zs,Ys)."})
public abstract class PermutationUtility {
  @PrologMethod (link="remove($1,$2,$0)",
                 style=PrologInvocationKind.FUNCTIONAL)
  abstract @GROUND List<Int> remove(@INPUT @GROUND List<Int> c1,
                                    @INPUT @GROUND Int i);
  @PrologMethod (link="permutation(@1,@2)",
                 multipleOutput=true,
                 style=PrologInvocationKind.RELATIONAL)
  abstract @GROUND Iterable<Compound1<List<Int>>> perms(
          @HIDE @INPUT @GROUND List<Int> c1, @OUTPUT @GROUND Var<List<Int>> c2);
  public static void main(String[] args) throws Exception{
     PermutationUtility pu = Java2Prolog.newInstance(PermutationUtility.class);
     java.util.Collection<Integer> l=new java.util.LinkedList<Integer>();
     l.add(1); l.add(2); l.add(3);
     Var<List<Int>> x = new Var<List<Int>>("X");
     List<Int> list = Term.fromJava(l);
     for (Compound1<List<Int>> compound : pu.perms(list,x)) {
       Collection<Integer> ci = compound.get0().toJava();
       System.out.println(Collections.max(li));
     }
  }
}
```

Figure 3.14: Defining Prolog methods in P@J [8]

3.2.4 Term classes

The set of convertible types discussed in Section 3.2.1 is rather poor. To allow to reason about the objects of your application domain, *term classes* can be defined. Term classes are also regarded as convertible types. A class can be declared a term class by annotating it with `@org.prolog4j.Term`. The annotation is inherited, so annotating a class with it causes that its subclasses will be term classes as well.

Processing `@Term` relies on OOLibrary. An OOLibrary class definition is generated from term classes.The conversion between Prolog and Java objects is performed using reflection. At converting towards Java the corresponding constructor is used. Fig. 3.15 shows a simplistic example for term class definition. Term classes may have fields of any type, but static or transient fields and those having a non-convertible type will not be converted.

As noted in 3.2.3, term classes, may also have non-static goal methods. In this case the actual object (`this`) will be bound to the variable called `This` that is regarded as an input only argument.

```
@Term
public class Human {
  private String name;
  public Human(String name) { this.name = name; }

  public static void main(String[] args) {
    System.out.println(isMortal("socrates")); // true
    System.out.println(getMortals());      // socrates, plato

    List<String> philosophers = Arrays.asList("socrates", "plato");
    Solution<?> solution = p.solve("member(X, List).",null,philosophers);
    System.out.println(solution.isSuccess()); // true

    for (String s: solution.<String>on("X"))
      System.out.println(s);               // socrates, plato

    List<Human> h1 = Arrays.asList(new Human("socrates"));
    List<Human> h2=Arrays.asList(new Human("thales"),new Human("plato"));
    for (List<Human> humans:
        p.<List<Human>>solve("append(L1, L2, L12).", h1, h2))
      for (Human h: humans)
        System.out.println(h.name);        // socrates, thales and plato

    List<Human> h3 = Arrays.asList(new Human("socrates"),
        new Human("homeros"), new Human("demokritos"));
    for (List<Human> humans: p.solve("append(L1, L2, L12).", h1,null,h3).
          <List<Human>>on("L2"))
      for (Human h: humans)
        System.out.println(h.name);        // homeros and demokritos
  }
}
```

Figure 3.15: Applying rules from Java

3.2.5 Implementation

The specification of methods annotated with `@Goal` contains every information to construct a `Solution` object. The body for these methods are generated at compile time. The annotations are processed by the Pluggable Annotations Framework [39] of Java 6. Although this framework does not provide means for code generation directly, it gives an environment, through which the whole compiler tree can be manipulated [38]. Unfortunately this feature works only with the Sun's Java compilers. tuProlog terms that represent a goal and its input variables are created at the first time the method is invoked then they are cached in a private static field of their class.

If `@In`, `@Out` or `@InOut` is specified in a goal method, the compile-time annotation processor verifies whether parameters of the annotations occur in the Prolog query.

Chapter 4

The Japlo language

Although Prolog is a very expressive language, its use may be rather strange for most of the users of object-oriented (OO) languages. The aim of this chapter is to present a Java language extension, in which it is easy to get familiar with writing logic rules also for programmers experienced in Java. Though the syntax of these rules differs from the Prolog standard (ISO/IEC 13211-1), they can be converted to the standard easily, and vice versa, that is the new syntax does not take away anything from the expressiveness of Prolog. The language is called *Japlo*[1]. Japlo [21] is the result of my previous research on integrating Prolog into Java.

4.1 Identifiers, constant symbols

Two kinds of notation are mainly used in Prolog implementations for differentiating variables and atoms. In the Prolog standard the character sequences with a capital initial letter denote variables, and those with a lower case one denote atoms. In some embedded Prolog variants, a special initial character is used, usually `?` (Allegro Prolog [30], TyRuBa [53], LogicAJ [55]). In these languages the character sequences starting with a question mark denote variables, any other identifier is an atom.

[1]The name is a – maybe strange – contraction of the words *Java* and *Prolog*. It was created with respect to the simple sounding and that it should not conflict with the name of any similar project.

The syntax of variables and number constants is the same as in Java. Wildcard variables start with `_` (underscore). Atoms and compound terms can be written using the *symbol operator*: ‘ (backquote). Some examples are shown below.

```
anIdentifier    AnotherOne    ‘anAtom    ‘aCompoundTerm(1, 2)
_    _bornToBeWild   // wildcard identifiers
```

4.2 Types, declarations

Japlo is a strongly typed language, which means that – in contrast with Prolog – variableshave to be declared specifying their type, which makes static type checking possible.

Java types can be used in the language and there are also some new types introduced for classifying Prolog data objects. These include the *atom* type and the *list* types. The new types fit into the Java type system: they are instances of `java.lang.Class`, lists and atoms are instances of `java.lang.Object` (more precisely, atoms are `japlo.lang.Atom` objects).

4.2.1 Lists

List types are characterized by the type of their elements (that is a common supertype of each of their elements) that is called its *base type*. The name of a list type looks like the name of its base type followed by braces (e.g. `Object{}`). List literals can be specified between braces, the elements of the list are separated by commas. The tail of a list can be specified after the `|` character, as illustrated in Fig. 4.1. Generally, for any list object of type `T{}` its head has to be of type `T` and its tail of type `T`.

If a list literal stands in a declaration, its type corresponds to the type of the variable just being declared. In other cases the compiler considers it `java.lang.Object{}`. List types different from the default can be specified by explicit type conversion.

```
Double{} dList = {5.3, 3 - 4};
{‘first, ‘second | dList}
{1, 2, 3 | {4, 5, 6}} = {1, 2, 3, 4, 5, 6}
(Atom{}) {‘apple, ‘pear}
```

Figure 4.1: Defining lists.

4.2.2 Variables, declarations

Every variable must be declared before use, including the formal arguments of rules and the variables introduced within the body of a rule. The type of the variables can only be a reference type, the value of the unbound variables is undefined or `null`.

Three kinds of objects can be declared within a Japlo rule: formal arguments, the elements of the pattern of formal arguments and local variables. The way of declaring formal arguments and local variables is the same as in Java. The elements of the argument patterns are declared automatically according to the declaration type of the argument. An example can be seen later, in Fig. 4.6.

4.3 Operators

Following the Java terminology, the equivalents of Prolog predicates are called operators. Most of the Java operators can be used in rules, but some of them have a modified meaning. There are also new ones.

The `=` (matching) operator unifies its operands. One of the most conspicuous difference contrary to the assignment operator of Java is that the concept of left and right values does not exist: operands take part in the operation equally:

```
Integer{} list1 = {1, _, 3};
{_, 2, 3} = list1;
```

Its opposite, the `!=` operator succeeds if and only if unification has failed. The operator `==` succeeds when its operands are identical. There is no unification: when one of the operands is an unbound variable then it fails. Its opposite is the `!==` operator.

The operators `false`, `true`, `!`, `!+` correspond to the standard Prolog predicates `fail`, `true`, `!`, `\+`, respectively. There are also control operators following the syntax of the Java if, if-else, while and do-while statements. Their semantics is summarized in Table 4.1.

The `&`, `|` operators are discussed in the next subsection.

Arithmetic predicates are substituted by the corresponding Java operators where they exist. They are summarized in Table 4.2.

[2] the value of the expression is K

Japlo operator	*Prolog predicate(s)*
`if (`*condition*`)` *action*	*condition* `->` *action*
`if(`*condition*`)` *action1* `else` *action2*	*condition* `->` *action1* `;` *action2*
`do` *action* `while (`*condition*`);`	`repeat,`*action*`,\+`*condition*`,!`
`while (`*condition*`)` *action*	*condition*`,repeat,`*action*`,\+`*condition*`,!`
goal1 `&` *goal2*	`(K =` *goal1*`, K =` *goal2*`)`[2]
goal1 `|` *goal2*	`(K =` *goal1*`; K =` *goal2*`)`[2]

Table 4.1: Control operators

Prolog predicate	`<`	`>`	`=<`	`>=`	`=\=`	`=:=`	`mod`
Japlo operator	`<`	`>`	`<=`	`>=`	`!!=`	`===`	`%`

Table 4.2: Arithmetic operators

4.4 Rules

The definition of rules is similar to method definitions. The definition starts with the `rule` keyword. Rules have a name, can have formal arguments and a return type. The return type can be `void`. Rules with `void` return type correspond to legacy Prolog rules, and are called *procedure rules*. In the case of rules with some other return type (from now *function rules*), it can be regarded as a special (zeroth) formal argument. This argument has no name, value can be *'assigned to'* (unified with) it using the `return` operator: `return <expression>;`. In contrast with other formal arguments, the return type argument must be output argument. Fig. 4.3 shows an example for function rules.

The body of a rule is a so called compound query, which can contain other queries or may be empty. The form of specifying non-compound queries may remind Java: non-compound queries end with a semicolon. Compound queries are represented as a sequence of queries within braces, and can contain variable declaration, application of rules or operators, Java expressions or compound queries.

Queries written successively are AND-connected. OR connection can be expressed using the `|` symbol. AND connection is stronger than OR, so it is not necessary to parenthesize OR-connected compound queries. Fig. 4.2 illustrates an example for that. It contains two rules. The first one describes an *acyclic directed graph* with its directly connected nodes, and the second one specifies whether two nodes are connected through a path.

```
rule static void arc(Atom from, Atom to) {
  from = ‘a; to = ‘b;
| from = ‘a; to = ‘c;
| from = ‘b; to = ‘c;
| from = ‘c; to = ‘d;
}
rule static void path(Atom from, Atom to) {
  arc(from, to);
|
  Atom through;
  arc(from, through);
  path(through, to);
}
```

Figure 4.2: Defining rules.

```
rule static Atom{} path(Atom from, Atom to) {
  arc(from, to);
  return {from, to};
|
  Atom through;
  arc(from, through);
  return {from | path(through, to)};
}
```

Figure 4.3: Function rules

4.5 Rule dispatch

Rules can be overloaded. In contrast with the method dispatch of Java, not only the method name and the number and type of arguments are taken into consideration at selecting rules. Declaration of formal arguments can contain a pattern, with which the actual arguments of rule applications are unified. Fig. 4.4 shows an alternative definition of the rules of Fig. 4.2. In case of overloaded rules a matching rule is searched for in the order of their definitions. The selection of the rule which can be applied, is performed using unification. Patterns can contain free names, which are declared implicitly according to the type of the pattern.

Using up return values you can express problems in a more concise way, in which successive manipulation on some data is needed. Fig. 4.5 shows an example for that. The problem is to find the nodes which cannot be avoided

```
rule static void arc(Atom _=‘a, Atom _=‘b) {}
rule static void arc(Atom _=‘a, Atom _=‘c) {}
rule static void arc(Atom _=‘b, Atom _=‘c) {}
rule static void arc(Atom _=‘c, Atom _=‘d) {}
rule static void path(Atom from, Atom to) { arc(from, to); }
rule static void path(Atom from, Atom to) {
  Atom through;
  arc(from, through);
  path(through, to);
}
```

Figure 4.4: Overloading rules.

```
findall(P, path(a, d, P), L1), intersect(L1, L2),
sort(L2, L3), write(L3).

write(sort(intersect(findall(path(‘a, ‘d)))));
```

Figure 4.5: Using up return value for subsequently manipulation on data.

going from *a* to *d*. (Every path between *a* and *d* contains them.) The name of the nodes has to be printed in sorted order. The solution uses the `intersect` rule that takes a list of lists as its argument and returns a new list that contains the intersection of the lists in the argument.

The figure illustrates solving the problem in Prolog and in Japlo using function rules. Besides that the latter solution is somewhat shorter, the introduction of the list variables is also avoided.

The `&` and `|` operators (in contrast with the short-circuit `&&` and `||` operators) unify their operands and the value of the expression will be the unified value. Fig. 4.6 shows a modified version of `member/2` using return value. The code sample writes out the common elements of two lists, in sorted order.

4.6 Accessing Java elements from rules

Within the definition of rules (apart from their formal arguments and local variables) any visible Java name can be referred to. You can construct (instantiate) objects, call methods or access fields just as in Java. The members of the declaring class can be accessed without qualification. Non-static rules are also allowed: the `this` pseudo variable can be used as itself or for qualifying members of the declaring class.

```
rule static Object memberf(Object{} _={first | rest}) {
  return first; | return memberf(rest);
}

// writing the nodes which are on one of the paths
// between ‘a and ‘d and between ‘b and ‘d:
write(memberf(path(‘a, ‘d)) & memberf(path(‘b, ‘d)));
// Compile error: The elements are not boolean!
write(memberf(path(‘a, ‘d)) && memberf(path(‘b, ‘d)));
```

Figure 4.6: Unificator operators

The fields do not behave as Prolog variables, they can be assigned to any times, the modification is a simple assignment, not unification.

Any Java expression ending with a semicolon makes a goal, whose evaluation always succeeds.

4.7 Application of rules from Java

Atoms, variables and lists can be specified in the same way as in rules. Variables are Java variables, including the wildcard variable that is not allowed to be declared. Variables having undefined or `null` value are unbound, other variables are regarded as bound. Primitive typed values are wrapped automatically, these values are considered as bound. Lists can also be specified in the same way as in rules. There is implicit conversion between lists and arrays of the same base type.

The form of rule applications is the same as at method invocations, but arguments are passed by value-result. Because of this, left-values have to be passed for every output arguments[3]. In the case of function rules – as long as you would like to use up the returned value – the form *left_value* `=` *rule_name*`(` *actual_arg_list*`)` has to be used. Just like method invocations, rule applications are expressions, too. The name of a rule can be qualified by the name of a class or an instance. Rules can be applied anywhere, where an expression of the required type can occur.

[3]In predicate descriptions the arguments, which are output of a predicate, are usually preceded by a '-'.

4.7.1 Existence of a solution

When you are only interested in whether a rule can be satisfied for some arguments then the `+` unary prefix operator has to be used. The operator has to be placed before the name of the rule applied. The type of the result of this expression is `boolean` and its value is `true` if and only if the rule can be satisfied.

4.7.2 Obtaining one solution

It happens often that there is no need for every solution, or it is known that exactly one solution exists. In this case the rule can be applied in the form *rule_name*`(`*actual_arg_list*`)` or *left_value* `=` *rule_name*`(`*actual_arg_list*`)`, respectively, according as it is a function rule or not. Fig. 4.7 shows an example for that. If the unification does not succeed, a `japlo.lang.NoMatchException` unchecked exception is thrown.

```
Atom{} p = path(‘a, ‘d);
```

Figure 4.7: Obtaining one solution: looking for a path between *a* and *d*.

4.7.3 Executing a statement for each solution

The most common intention with rule application is to perform some activity depending on the solutions. This is what the *enhanced for loop* serves for. In case of function rules the head of the loop can be specified like the for-each loop of Java5 ([40]). An example can be seen in Fig. 4.8. In case of procedure rules the head should contain only the application of the rule, the declaration and the colon is omitted.

```
for (Atom{} p: path(‘a, ‘d))
  System.out.println(p);
```

Figure 4.8: For each solution

4.7.4 Collecting solutions

Collecting every solution can be performed in the similar way as obtaining the first solution. The only difference is that in case of an unbound variable of

type `T` the type of the argument has to be `Collection<? super T>` instead of `T`. During the application of the rule the elements are added to the collection in the order of being found. That is, you should use a sorted container (e.g. `SortedSet`) if you would like to obtain the elements in a certain order as in Fig. 4.9.

```
Collection<Atom> to = new TreeSet<Atom>();
path(‘a, to);
for (Atom node: to)
  System.out.printf("There is a path from a to %s\n", node);
```

Figure 4.9: Collecting the places that can be reached from *a*.

As it is not guaranteed for any `Collection` that the `add` method inserts its argument after the last element (so traversing the collection would reflect the order of finding the elements), so in case of more unbound variables `List<? super T>` objects should be passed. Using lists it can be guaranteed that the bound values of the individual solutions are accessed together when traversing the solutions parallelly. The order of the elements reflects, in which order the Prolog engine has found them. An example for the use can be seen in Fig. 4.10.

```
List<Atom> from = new ArrayList<Atom>();
List<Atom> to = new ArrayList<Atom>();
path(from, to);
for (int i=0; i<from.size(); ++i)
  System.out.printf("There is a path from %s to %s\n",
                from.get(i), to.get(i));
```

Figure 4.10: Collecting every path in arbitrary order.

4.8 Japlo vs. JLog

Japlo is intended to provide an intuitive interface over JLog. To set Japlo against JLog, I show, how one of the previous examples could be programmed using the JLog API. Fig. 4.11 shows a JLog variant of Fig. 4.8. The `jPrologAPI` constructor consults the "path.pl" file, which contains the definition of the `path` and `arc` rules. The solutions of a query (sets of bindings) are obtained in `Hashtable` objects, in which the keys are the name of the variables.

```
jPrologAPI prolog = new jPrologAPI(new FileInputStream("path.pl"));
Hashtable varz = prolog.query("path(a, d, P)");
while (varz != null) {
    System.out.println(varz.get("X"));
    varz = prolog.retry();
}
prolog.stop();
```

Figure 4.11: Obtaining one solution using JLog.

The disadvantage of querying this way is that Java objects cannot be passed to rules as arguments. Although in this simple example there is no need to pass Java objects, it is obvious that passing objects to rules should be a very rudimentary feature of such a hybrid language. To achieve this the goal object should be constructed at runtime. The second example show how this can be done for the previous example. The `buildKnowledge()` method constructs the *knowledge*, which contains the definition of the `path` and `arc` rules. This code is 14 lines long, the previous solution stands of 7 lines while the Japlo version is only 2 lines long.

```
jKnowledgeBase knowledge = buildKnowledge();
jProver prover = new jProver(knowledge);
jPredicateTerms goal = new jPredicateTerms();
jCompoundTerm goalArgs = new jCompoundTerm(3);
goalArgs.addTerm(new jAtom("a"));
goalArgs.addTerm(new jAtom("d"));
jVariable P = new jVariable();
goalArgs.addTerm(P);
goal.addPredicate(new jPredicate("path", goalArgs));
boolean b = prover.prove(goal);
while (b) {
    System.out.println(P.getValue());
    b = prover.retry();
}
```

Figure 4.12: Obtaining one solution using JLog.

4.9 Implementation

The implementation is based on *JLog* [29] and the *Polyglot* compiler framework [46].

The compiler was created using Polyglot. Polyglot is an extensible compiler framework for Java. It is an ideal choice for creating Java-like languages, because it is highly extensible: the grammar, the classes representing the nodes of the abstract syntax tree (AST) and any phase of the compilation process can be modified in such way, that the original remains untouched. Of course, new AST node types and new phases in the compilation process can be introduced, and you can also remove some when they are not needed. Meanwhile, the original compiler is not modified, the new one is created only by code reuse.

The Japlo compiler translates rules into pure Java code. It creates a final field for a rule in the declaring class that will hold the JLog rule object (`ubc.cs.JLog.Foundation.jRule`), and an initializer block that constructs this object. The rule objects must be constructed at run-time because they may need to access the private fields of the declaring class. For example, assigning to a field is translated to an anonymous class extending an abstract predicate class. (Anonymous classes can access the private members of their declaring class.) Dynamic creation of rule objects may be slow, especially in case of instance rules. This performance problem could be avoided when not the whole rule would be constructed at run-time, but just its predicates which need to access some non-public member of the declaring class. Then these predicates can be passed to the static part of the rule as an additional argument, and the rule can perform them by the `call` meta-predicate. In case of instance rules the target object (`this`) has to be passed, too. Then the static part can be instantiated at load-time (through static initializators) or even at compile-time. In the latter case this rule object is created by the compiler and is serialized, and a custom class loader is responsible for loading the rules. Separating the static and dynamic part of rules in this way is the subject of later development.

Besides that the 'call sides' of Japlo rules have also to be replaced, so that the values of the variables that get bound by the rule application, are assigned to the corresponding actual arguments (when they are variables). To achieve this the application of procedure rules is translated into a block, in which the necessary assignments are performed. Similarly, the application of function rules is translated into a complex expression. Because Java does not have the comma operator, this functionality is performed by the `|` logical operator (not short-circuit), `==` and `?:`.

Chapter 5

Declarative metaprogramming framework for Prolog4J

In this section I present a declarative metaprogramming framework for Java. The framework is based on OOLibrary and Prolog4J. It has two layers. The low-level layer stores the metainformation of the program as facts (meta-database). The high-level layer defines OOLibrary classes and methods for examining the contents of the database.

5.1 The low-level layer

The meta-database stores information about the structure of the program: the specification of classes, fields, constructors and methods. Because of space reasons the body of methods is not stored in the database. Instead of that, only the dependencies between methods and fields are stored, e.g. which fields and methods are accessed from which methods. The database is created at load time by a Java agent through processing the bytecode of classes. Although the whole method definitions (with method bodies) are not stored in the meta-database as facts, they can be accessed from Prolog. Then the body of the method is retrieved at run-time by the same Java agent mentioned above.

The database contains facts of the following forms:

- `'$class'/3`
  `'$class'(ClassID, Scope, ClassSpec)` facts store the specification of classes. `ClassID` is the internal name of the class (for example 'java/lang/Object'). `Scope` is a package ID, class ID or method ID, according to the type of the class. Finally, `ClassSpec` represents the specification of the class by a compound term of the following form: `classSpec(Annotations, Access, Name, Signature, Superclass, Interfaces)`.

- `'$field'/3`
  `'$field'(FieldID, Scope, FieldSpec)` facts store the specification of fields. `FieldID` is the concatenation of the ID of the declaring class, a dot and the name of the field. `Scope` is the ID of the declaring class. Finally, `FieldSpec` represents the specification of the field by a compound term of the following form: `fieldSpec(Annotations, Access, Name, Descriptor, Signature, Value)`.

- `'$method'/3`
  `'$method'(MethodID, Scope, MethodSpec)` facts store the specification of methods. `MethodID` is the concatenation of the ID of the declaring class, a dot, the method name and the method descriptor. `Scope` is the ID of the declaring class. Finally, `MethodSpec` represents the specification of the method by a compound term of the following form: `methodSpec(Annotations, Access, Name, Descriptor, Signature, ParameterAnnotations, Exceptions)`.

- `'$get'/2`
  `'$get'(FieldID, MethodID)` facts store places of reading fields. `FieldID` denotes the field and `MethodID` the method from which the field is read.

- `'$set'/2`
  `'$set'(FieldID, MethodID)` facts store places of writing fields. `FieldID` denotes the field and `MethodID` the method from which the field is assigned to.

- `'$call'/2`
  `'$set'(Callee, Caller)` facts store the places of method invocations. `Callee` denotes the ID of the called method and `Caller` the ID of the method from which `Callee` is invoked.

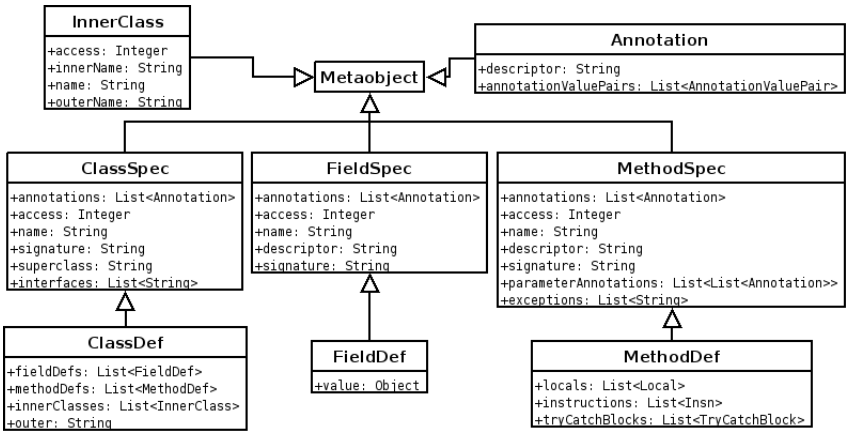

Figure 5.1: Metaclass hierarchy: classes, fields and methods

5.2 The high-level layer

The high level layer consists of OOLibrary class and method definitions. The metaclass hierarchy is shown in Fig. 5.2 and Fig. 5.2. For the better illustration, the figures contain UML class diagrams. The hierarchy reflects the structure of the ASM Tree API [7].

The `classSpec`, `fieldSpec` and `methodSpec` objects can be retrieved by the `class/1`, `field/1` and `method/1` predicates, respectively, where the first argument is the ID and the second the specification. Neither of the arguments need to be bound. Similarly, `classDef`, `fieldDef` and `methodDef` objects can be retrieved by the `classDef/1`, `fieldDef/1` and `methodDef/1` predicates. Method definitions are not stored in the database, they are retrieved at run-time by these predicates.

As it can be seen in Fig. 5.2, there are no distinct types for representing class, interface, enum and annotation types. Similarly, constructors, methods and static initializers are represented by `methodSpec` metaobjects, equally. The following methods serve for examining the access modifiers and the type of a `classSpec` or a `methodSpec` metaobject:

- `isAbstract(Access: int)`
  An abstract class or method.
- `isAnnotation(Access: int)`

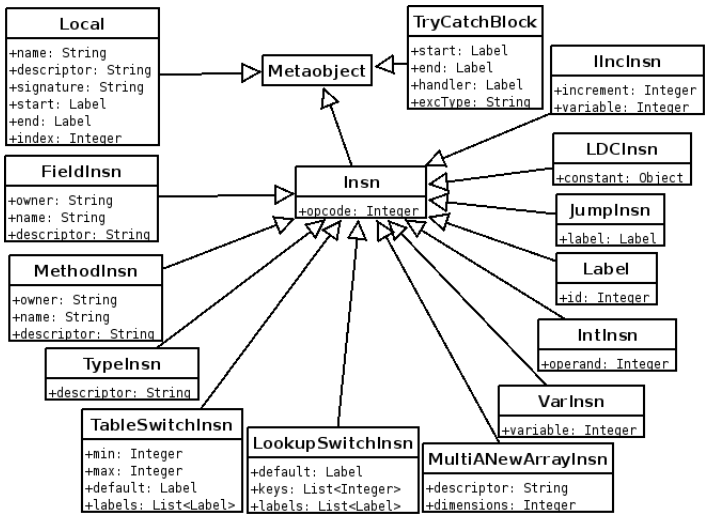

Figure 5.2: Metaclass hierarchy: inside method bodies

An annotation type.

- `isBridge(Access: int)`
  A bridge method.
- `isDeprecated(Access: int)`
  A deprecated type, field or method.
- `isEnum(Access: int)`
  An enum type.
- `isFinal(Access: int)`
  A final type, field or method.
- `isInterface(Access: int)`
  An interface type.
- `isNative(Access: int)`
  A native method.
- `isPrivate(Access: int)`
  A private member.

- `isProtected(Access: int)`
  A protected member.
- `isPublic(Access: int)`
  A public top-level type or member.
- `isStatic(Access: int)`
  A static member.
- `isStrict(Access: int)`
  A member with strictfp modifier.
- `isSynchronized(Access: int)`
  A synchronized method.
- `isSynthetic(Access: int)`
  A type or member not appearing in the source.
- `isTransient(Access: int)`
  A transient field.
- `isVarargs(Access: int)`
  A method with variable length argument list.
- `isVolatile(Access: int)`
  A volatile field.
- `isPackage(Access: int)`
  A type or member with package visibility.
- `isClass(CS: classSpec)`
  CS is a class type.
- `isInterface(CS: classSpec)`
  CS is an interface type.
- `isEnum(CS: classSpec)`
  CS is an enum type.
- `isAnnotation(CS: classSpec)`
  CS is an annotation type.
- `isConstructor(MS: methodSpec)`
  MS is a constructor.

- `isMethod(MS: methodSpec)`
  MS is an ordinary method.

- `isStaticInitializer(MS: methodSpec)`
  MS is a static initializer.

The fields of metaobjects can be accessed by the accessor methods (e.g. `nameOf`). There are additional predicates for examining the method descriptors and relationships between metaobjects:

- `returnTypeOf(Method: methodSpec, ReturnType: typeDesc)`
  The return type of a method.

- `argType(Method: methodSpec, ArgNo: int, ArgType: typeDesc)`
  An argument type of a method.

- `childOf(P: metaobject, C: metaobject)`
  C is declared in P.

- `'childOf+'(P: metaobject, C: metaobject)`
  Transitive closure of `childOf`.

- `subtypeOf(C1: classSpec, C2: classSpec)`
  C2 is a direct subtype of C1.

- `'subtypeOf+'(C1: classSpec, C2: classSpec)`
  Transitive closure of `subtypeOf`.

- `'subtypeOf*'(C1: classSpec, C2: classspec)`
  Reflexive transitive closure of `subtypeOf`.

- `fieldOf(C: classSpec, F: fieldSpec)`
  F is a field declared by C.

- `'fieldOf+'(C: classSpec, F: fieldSpec)`
  F is a field inherited from C.

- `'fieldOf*'(C: classSpec, F: fieldSpec)`
  F is a field declared by or inherited from C.

- `methodOf(C: classSpec, M: methodSpec)`
  M is a method declared by C.

- `'methodOf+'(C: classSpec, M: methodSpec)`
  M is a method inherited from C.
- `'methodOf*'(C: classSpec, M: methodSpec)`
  M is a method declared by or inherited from C.
- `overrides(M1: methodSpec, M2: methodSpec)`
  M2 overrides M1.
- `overloads(M1: methodSpec, M2: methodSpec)`
  M2 overloads M1.
- `reads(M: methodSpec, F: fieldSpec)`
  Identical with '$get'.
- `changes(M: methodSpec, F: fieldSpec)`
  Identical with '$set'.
- `calls(M1: methodSpec, M2: methodSpec)`
  Identical with '$call'.
- `'calls+'(M1: methodSpec, M2: methodSpec)`
  Transitive closure of `calls`.
- `'calls*'(M1: methodSpec, M2: methodSpec)`
  Reflexive, transitive closure of `calls`.
- `reachable(C1: classSpec, C2: classSpec)`
  Determines whether there is a path of directed associations from `C1` to `C2`.

Naturally, new methods can be defined over the existing ones, freely. So, logic-based analysis of the program is possible. Fig. 5.3 shows a simple example for this. The `isSingleton` method determines whether the Singleton pattern ([26]) is applied for a class or not. If the class is singleton, method and the field which take part in the pattern will be bound to the arguments of `isSingleton`. A class is regarded as a singleton if it has a public, static method which returns an instance of the class and does not expect any argument. The method has also to read a private static field of the class whose type is the class itself. Additionally, the class can have only private constructors.

Since the whole class definitions (including method bodies) can be accessed from Prolog, transformations can be performed on them by transforming the terms representing the metaobjects. Unfortunately, there are no predicates predefined by the framework for this purpose. The standard Prolog predicates

```
isSingleton(C: classSpec, M: methodSpec, F: fieldSpec) :-
  methodOf(C, M), isPublic(M), isStatic(M),
  returnTypeOf(M, CDesc), descriptorOf(C, CDesc), argTypesOf(M, []),
  fieldOf(C, F), isPrivate(F), isStatic(F), descriptorOf(F, CDesc),
  reads(M, F),
  methodOf(C, Ctr), isConstructor(Ctr),
  (not(isPrivate(Ctr)) -> !, fail).
```

Figure 5.3: The isSingleton method

can be used for the transformation. However, it is planned to define an easy-to-use interface that makes it possible to specify transformation on the code. Conditional Transformations [34] used by JTransformer is an exemplary work for the same task.

5.3 Implementation

The static database is created using the Java Instrumentation mechanism (`java.lang.instrument` package) introduced in JDK 5 that allows you to provide *Java agents* that can inspect and modify the bytecode of the classes as they are loaded. Java agents can be distributed as ordinary Java archives (e.g. agent.jar) and can be enabled at launching a program by passing the `-javaagent:agent.jar` to the JVM.

An agent can register a *class file transformer* with the system class loader that provides a `transform` method. This method is then called as the part of class loading for each and every class from then on, and may manipulate the bytecodes before they are processed by the class loader into a real class.

This framework makes it easy to access the bytecode before the class definition, but does not provide tools for carrying out the bytecode manipulation itself. I used ASM [7] for this purpose. ASM is an all purpose Java bytecode manipulation and analysis framework, focused on simplicity of use and performance.

ASM provides two APIs for processing compiled classes: the core API provides an *event based* representation of classes, while the tree API provides an *object based* representation. Both API relies on the Visitor design pattern [26]. When processing a bytecode by a `ClassReader`, it generates one event per element that is parsed. Using the core API, you have to write your own visitor (implementing e.g. the `ClassVisitor` interface) that will be notified of each step of process. The advantage of this technique against other bytecode manip-

ulation framework is that ASM uses less memory, because it is not needed to build a tree of objects representing each part of the class.

You are still able to build objects representing a class using the tree API. The tree API is based on the core API but it provides predefined visitors (called nodes, e.g. `ClassNode`) that build up the object representation of the bytecode. Moreover, the two APIs can be combined: you can build up the object representation for some parts of the bytecode, while neglecting other parts.

In case of the current metaprogramming framework, the tree API is used for constructing objects representing the class, field and method specifications. These objects are then transformed to Prolog terms, and stored in the database (`'$class'`, `'$field'` and `'$method'` predicates). However, within the method definitions the core API is used to detect the dependencies between classes (`'$get'`, `'$set'` and `'$call'` predicates).

As mentioned before, the body of methods is not stored in the database. This has two reasons. At first, this would make it necessary to use the tree API for the whole class definition, which could result in a significant time overhead at load-time. At second, these objects representing the method definitions may be fairly large, so storing them all along the program execution may not be ideal. For these reasons, the whole method definitions can be loaded at runtime by the `methodDef/2` predicate. The predicate is implemented in Java as a primitive tuProlog predicate. The first argument of the predicate is a method ID. It loads the body of the method, constructs the method definition object and binds them to its second argument.

The primitive predicate loads the method body by requesting the Java agent to redefine the declaring class. Indeed, the class will not be changed, only its bytecode will be processed by ASM to construct the `MethodNode` object representing the body. To minimize the number of class redefinitions, using the `methodDefs/2` predicate the definition of several methods can be retrieved at the same time. Moreover, the method bodies are cached during the execution of a query.

It is important to note that although `java.lang.instrument` had been introduced in JDK 5, it supports class redefinition only from JDK 6. However, "hot swap" class file replacement is possible through the Java Debugger Interface (JDI) from JDK 1.4.0. JDI is one of the three interfaces of the Java Platform Debugger Architecture (JPDA). Using this feature through JDI, you need to run the JVM in `debug mode` (-Xdebug option). As of JDK 1.4.0 "full speed debugging" is supported by the Java HotSpot VM, which minimizes the performance loss caused by the debug mode. Full speed debugging means that the programs being debugged do not need to be interpreted as before JDK 1.4.0.

Chapter 6

Aspect-oriented programming framework for Prolog4J

In this chapter I introduce an aspect-oriented framework based on the logic metaprogramming library presented previously. In the first section its pointcut language is introduced that is a set of Prolog predicates, essentially. The next short section will show how aspects can be defined using the Metadata Facility of Java. After that I present a controlled natural language interface over the pointcut language discussed before. Finally, I give an insight into some details of the implementation.

6.1 Pointcuts

Pointcuts can be expressed by Prolog goals that contain predicates called *primitive pointcuts*. Primitive pointcuts can be used to refer to a join point of the program. It can be specified as their first argument. Primitive pointcuts can be categorized into three classes. The elements of the first category describe the type of the join points. These are the followings:

- `get(JP, Field)`
  Succeeds iff the field is read at the join point.

- `set(JP, Field)`
  Succeeds iff the field is set at the join point.

- `call(JP, Method)`
  Succeeds iff the method is called at the join point.

- `execution(JP, Method)`
  Succeeds iff the method is executed at the join point.

- `handler(JP, ExceptionType)`
  Succeeds iff the an exception of the given type is handled at the join point.

The second category contains pointcuts designating the context of the join point. These are the followings:

- `this(JP, This)`
  Succeeds iff the join point is within a constructor or an instance method. The actual object will be bound the second argument, which must be a variable.

- `target(JP, Target)`
  Succeeds when the join point reads or writes an instance field (`get`, `set`) or invokes an instance method (`call`). The target object will be bound to the second argument, which must be a variable.

- `arg(JP, Arg)`
  Succeeds in case of `set`, `call`, `execution` and `handler` join points. For `call` and `execution` the formal argument list must be of length 1. The argument will be bound to the second argument, which must be a variable.

- `args(JP, Args)`
  Succeeds in case of `call`, `execution` and `handler` join points. The arguments passed to the method will be bound to the second argument, which must be a variable or a list of variables.

- `returns(JP, ReturnValue)`
  Succeeds in case of `get`, `call` and `execution` join points. For `call` and `execution` the return type must not be `void`. The value of the field read or the return value of the method will be bound to the second argument, which must be a variable.

- `instanceof(Object, ClassSpec)`
  The first argument must be a context variable. Succeeds iff the denoted contex element is an instance of `ClassSpec`.

The context exposer pointcuts bind some elements of the context of a join point to Prolog variables. These variables will belong to the *interface* of the pointcut. This binding is symbolic: the value of context elements cannot be accessed through the variables representing them within the pointcut. However, the context variables can be assigned to the formal arguments of an advice, and then the context elements they represent will be passed to it.

Finally, the third category of primitive pointcuts contains predicates using which you can locate the join points within the program flow. Using "within" you can set constraint on the (static) location of the join point. With the help of the rest of the predicates, you can correlate the time of the join points with each other.

- `within(JP, MO)`
  Succeeds iff the location of the join point is within the specified program element.
- `pcflow(JP1, JP2)`
  Succeeds iff the activity at JP2 is under the predicted control flow of JP1 based on the AST. (`'calls*'`)
- `pcflowbelow(JP1, JP2)`
  Like "pcflow", but the join points does not refer to the same execution of a method. (`'calls+'`)
- `cflow(JP1, JP2)`
  Succeeds iff the activity at JP2 is under the control flow of JP1.
- `cflowbelow(JP1, JP2)`
  Like "cflow", but the join points does not refer to the same execution of a method.
- `now(JP)`
  Succeeds iff the activity at the join point is running currently.

High-level pointcuts can be defined over the primitive ones by Prolog rules. The predicates of the Prolog4J metaprogramming library can also be used in the rule bodies. There are predefined high-level pointcuts, and you can define new ones by the `@Theory` annotation. Some of the available high-level pointcuts:

```
get(Field) :- get(JP, Field), now(JP).
set(Field) :- set(JP, Field), now(JP).
call(Method) :- call(JP, Method), now(JP).
execution(Method) :- execution(JP, Method), now(JP).
handler(ExceptionType) :- handler(JP, ExceptionType), now(JP).
this(This) :- this(JP, This), now(JP).
target(Target) :- target(JP, Target), now(JP).
args(Args) :- args(JP, Args), now(JP).
returns(ReturnValue) :- returns(JP, ReturnValue), now(JP).
within(Method) :- within(JP, Method), now(JP).

arg(JP, Arg) :- args(JP, [Arg]).
args(JP, Arg1, Arg2) :- args(JP, [Arg1, Arg2]).
args(JP, Arg1, Arg2, Arg3) :- args(JP, [Arg1, Arg2, Arg3]).

arg(Arg) :- arg(JP, Args), now(JP).
args(Args) :- args(JP, Args), now(JP).

p_reachable(Obj1, Obj2) :-
  instanceof(Obj1, Class1), instanceof(Obj2, Class2),
  reachable(Class1, Class2).
```

6.2 Annotations

Similarly to AspectWerkz [6] and AspectJ 5 [11], this AOP framework relies on the Metadata Facility of Java 5 [5], so it does not need the extension of the Java language. Aspects can be defined by annotating a class with *@Aspect*.

Advices can be specified in any aspect by the `@Before`, `@After`, `@AfterReturning` and `@AfterThrowing` annotations. The argument of these annotations is a pointcut. Context variables of the pointcut can get bound to the formal arguments of the advice by the `@Arg` annotation. The only argument of the annotation is the name of the context variable.

Fig. 6.1 gives two examples which illustrate how aspects can be defined in the system. They are based on the examples for ALPHA [49].

6.3 Controlled natural language interface

The aim of multiparadigm languages is to unify the advantages of several programming paradigms. The Java-Prolog integration is useful in several fields,

```
@Aspect
public class DisplayUpdating {
  private Display d;

  @After("set(F), 'fieldOf*'('FigureElement', F), target(P)")
  public void update(@Arg("P") FigureElement fe) {
    d.draw(fe);
  }
}
@Aspect
public class DisplayUpdating {
  private Display d;

  @After("set(F), 'fieldOf*'('FigureElement', F), target(P)," +
    "call(T1, 'Display.drawAll()'), get(T2, F), pcflow(T1, T2)")
  public void update(@Arg("P") FigureElement fe) {
    d.draw(fe);
  }
}
```

Figure 6.1: Examples for the use of the AOP framework

where there are subproblems which have a more obvious solution in Prolog. For example, logic programming is more suitable for writing metaprograms than imperative programming, even if the base program is written in an imperative language. However, because of the radically different concepts of the two languages (Prolog and Java) and because the very different group of their users, Prolog may be merely unusual for Java programmers. For bridging the gap I introduced the Japlo language and the Prolog4J framework. As mentioned in Sec. 2.4.2, LogicAJ hides the inner use of Prolog by the use of Java-style logic expressions in pointcuts.

Now, an alternative solution will be presented, namely the use of a controlled natural language. Controlled natural languages usually comprise the vocabulary specific to an application domain, and using these terms one can compose sentences in a predefined, restricted way. Naturalistic programming suggests the use of a controlled natural language in *general purpose* programmming languages as well, for example for writing the body of methods. Here, the domain of the language contains the concepts of structured and object-oriented programming.

Controlled natural languages are prefectly suitable for defining pointcuts,

because of several reasons:

- They describe a specific domain. Here, the domain is not the application of the program but metaprogramming as such.
- Pointcut designators have well-defined, simple structure, which provides two benefits:
  - The programmer will probably not constrained of the rigid structure of the controlled language.
  - The language can be constructed quite easily.
- Pointcuts should be defined in a declarative way for avoiding fragility, and declarations are the most basic form of communication in natural languages.

As you will see, the pointcuts defined in this controlled natural language will not be much more verbose than by using AspetcJ logic expressions or a Prolog goal in ALPHA. Sometimes they may be even shorter. An additional advantage is that the specification of pointcuts become readable for users not familiar with aspect-oriented programming at all. Moreover, phrasing them is also more natural.

6.3.1 The language

Since the pointcuts pick up events in the program flow instead of describing statements performing a given task, you cannot use full sentences in the language, but only clauses. At first, *event clauses* will be introduced that start with an *event phrase* and may continue with *adverbial phrases*. An event phrase consists of the type of the event (a gerund) and its `object`. There are five types of event phrases: `reading`, `changing`, `calling`, `executing` and `handling`. The simplest way of specifying the object is by a literal containing its class and signature:

```
reading 'FigureElement.x'
calling 'FigureElement.setX(int)'
```

The keywords of the language contain lower-case letters. Literals starting by an upper-case letter are enclosed within apostrophes. It is possible to use variables, which start with a capital letter. Variables are used for two purposes. At first, for exposing values at the context of the join point, just like in AspectJ.

Context phrase	*Valid for*	*Exposes*
storing *Value*	reading, changing	the value of the field
to *Value*	changing	the value assigned to the field
with *Arg1*, *Arg2*, ...	calling, executing	the arguments passed to the method
resulting *Value*	calling, executing	the value returned from the method
Exception	handling	the exception
of *Object*	reading, changing, calling, executing	the target object
by *Object*	any	the actual object (this)

Table 6.1: Context phrases

These variables are called *context variables* and they can be used in prepositional phrases (there is no preposition for `handling`) written after the event phrase. Context variables belong to the `interface` of the pointcut. They will be bound to the formal arguments of the advice, as discussed in Section 6.2. The *context phrases* are summarized in Table 6.1. Of course, they can be combined, for example if you would like to refer to the arguments and the target object of a method invocation.

The declaring class of fields or methods can also be specified by the `of` preposition: e.g. `reading x of a 'FigureElement'`. The indefinite article suggests that `x` is an instance variable. Although this form is longer, it will be useful later. If you want pick out the target object, you can place a variable name after the type:

```
reading x of a 'FigureElement' F
```

Another use of variables is to denote program elements. In this case, variables appear usually in the object of a clause but there are some other phrases accepting metavariables. These variables are *local* and do not belong to the interface of the pointcut. There are no wildcard variables (matching any program element). The uninterest of the identity of a metaobject can be expressed by using the type of the metaobject (`a field`, `a method`, etc.). The type of the program element can also be given before the variable name.

```
reading X of a 'FigureElement'
reading a field of a 'FigureElement'
```

Clauses can be concatenated by a comma possibly followed by `or`. The single comma means "and" connection. A typical case of concatenating clauses

Metaclause	*Succeeds iff the subject*
Object is a *Type*	is an instance of *Type*
Object is of type *Type*	has the type *Type* (exactly)
MO is *Modifier*	has the modifier
MO is subtype of *Type*	is a subtype of *Type*
MO is supertype of *Type*	is a supertype of *Type*
MO is annotated by *Annotation*	has the annotation
MO is a member of *Metaobject*	the value returned from the method
MO returns *Type*	has return type *Type*
MO has argument types *Type1*, *Type2*, ...	has arguments of the given types
MO throws *Type*	declares exception of the given type

Table 6.2: Metaclauses

is when the properties of a program element has to be specified. These are called metaclauses, because their subject is a metavariable. Another reason for using several clauses is to express the spatial or temporal connection of join points (relative clauses).

Some examples of the metaclauses are shown in Table 6.2. Within metaclauses you can place a metavariable after any metaobject so that you can describe its properties in another clause.

If you want to express the spatial or temporal relation between events, you have to supplement the event clause by the *at phrase*: `at` *Variable*. Then the variable will denote the execution point at the event, and it can be used in relational clauses. *Relational clauses* are summarized in Table 6.3.

Relational clause	*Succeeds iff*
JP is within *MO*	the code of the event is contained by the given program element
JP is now	`JP` is the current join point
JP is last time	The last occurence of the event
JP1 is during *JP2*	`JP1` has started after `JP2`, which has not finished
JP1 is before *JP2*	`JP1` has finished before `JP2`
JP1 is after *JP2*	`JP2` has finished before `JP1`
JP1 is below *JP2*	`JP1` is in the control flow of `JP2` (not reflexive)
JP1 is above *JP2*	inverse of *below*

Table 6.3: Relational clauses

6.3.2 Conciseness

Although using controlled natural language phrases, pointcuts can be defined in an easy-to-read way, this formalism cannot be said to be concise, at all. The natural language gives a chance to the user to comprehende (or even define) pointcuts, even if he/she is not familiar with aspect-oriented programming. Despite this, a verbose definition can decrease readability in some cases when it is not complex, but much longer than its functionally equivalent version in e.g. LogicAJ or ALPHA. Fortunately, natural languages provide several ways to express these phrases in a more concise way, so we only need to explore some tricks our native language provide, and integrate it to the controlled language.

One such trick is that you can unify similar or-connected clauses if they differ only in their object. The resulted phrase will contain or-connected objects. Table. 6.4 compares the definition of an enumeration pointcut in AspectJ, LogicAJ and the current controlled language with and without or-connected objects. As it can be seen, the AspectJ variant is the longest. Using metavariables makes the pointcut definition somewhat shorter. Using or-connected objects, however, halves the length of the pointcut.

Description	*Pointcut*	*Length*
AspectJ	set(FigureElement.x) || set(FigureElement.y) || set(FigureElement.start) || set(FigureElement.end)	98
LogicAJ	(set(?T.x) || set(?T.y) || set(?T.start) || set(?T.end)) && equals(?T, FigureElement)	85
or-connected phrases	changing x of 'FigureElement' P, or changing y of P, or changing start of P, or changing end of P	93
or-connected subjects	changing x, y, start or end of a 'FigureElement'	48

Table 6.4: Or-connected objects

Similarly, the metaphrases can also be combined. For example, instead of writing

```
reading F,
F is public,
F is static,
F is final
```

you can write simply

```
reading F,
F is public, static and final
```

The use of variables is not typical at all for natural languages. (Although, they may occur in scientific text.) Since local variables do not take part of the interface of the pointcut, they are just its internal details and they are not necessary for the advice body. So, pointcuts could be expressed in a more natural way if the local variables could be eliminated from them.

One way to achieve this is unifying several phrases. This is possible only if the variable is used at exactly one time. Table 6.5 shows three examples for this. The first one has already been discussed. In the second one, modifiers are specified as *attributes* of the subject. Finally, in the last one the temporal phrase has been appended to the previous phrase.

Table 6.6 shows an example for expressing a pointcut in ALPHA and in controlled natural language. The left coloumn contains the "cflowreach" pointcut discussed in Section 2.4.3 (146 character). The equivalent controlled natural language pointcut is in the right coloumn (113 character).

With local variable	*Without local variable*
reading a field of P, P is a 'FigureElement'	reading a field of a 'FigureElement'
reading F, F is public, static and final	reading a public,static and final field
calling 'Display.update()' at T1, reading a field of a 'FigureElement' at T2, T2 is during T1	calling 'Display.update()' at T1, reading a field of a 'FigureElement' during T1

Table 6.5: Eliminating local variables

ALPHA	*Controlled natural language*
set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, this.d, 'drawAll', _)), cflow(T1, T2), reachable(Q, P), instanceof(Q, 'FigureElement')	changing F of P, calling drawAll of d last time at T, reading F of P during T, P is reachable from a 'FigureElement'

Table 6.6: The "cflowreach" pointcut in ALPHA and CNL

Anaphoric relations

Another possible way of removing local variables could be to express grammatical relations by *co-referring* phrases. Co-reference means that two noun phrases refer to the same object. The first is called the *antecedent* and the second the *anaphora*. [3]

Local variables could be substituted by anaphoras that can refer to various objects of a pointcut. Then first use of a local variable (denoting the antecedent) could be ommitted or replaced by its type. Subsequent uses of the same variable could be replaced by an anaphora.

Anaphoras are not supported by the Prolog4J project because the problems they would raise are not compensated by their benefits. At first, anaphoric relations can be ambiquous. Although the chance of ambiguity could be reduced if the anaphora would contain the type of the antecedent (e.g. "that field", "that time", etc.), it cannot be excluded totally. At second, a minor issue is that the use of anaphoras make the pointcuts a bit more verbose. At third, they would require a complex semantical analysis [3].

6.4 Implementation details

The controlled natural language is defined by a *definite clause grammar* (DCG). DCGs are a special notation provided in most Prolog systems which provide you with a convenient way of defining grammar rules. tuProlog supports the use of DCGS. A DCG rule can be specified in a similar way as normal prolog rules, but the "neck" of the rules is `-->` instead of `:-`. Inside the body of the rules terminals have to be enclosed within brackets. Using DCGs non-context-free grammars can also be expressed, if the grammar rules have formal arguments. A short excerpt of the full definition of the language is show below to illustrate the easiness of use of definite clause grammars.

```
pointcut(PCD) -->
   clause(PCD).
pointcut(','(PCD1, PCD2)) -->
   clause(PCD1), [','], pointcut(PCD2).
pointcut(';'(PCD1, PCD2)) -->
   clause(PCD1), [',', or], pointcut(PCD2).

clause(PCD) -->
   event_clause(PCD).
clause(PCD) -->
```

```
   meta_clause(PCD).
clause(PCD) -->
   temporal_clause(PCD).
```

Chapter 7

Summary

Summarizing the results of the dissertation it can be declared that the Java/Prolog multiparadigm programming framework (Prolog4J) that I have developed is easier to use than other similar projects in this field for several reasons. At first, it exploits the new language features of Java 5 like for-each loop, autoboxing or annotations. At second, it performs type conversions between Java objects and Prolog terms automatically, even for complex data types, without introducing new (wrapper) types. Finally, because it allows defining interfaces for Prolog queries as Java methods whose body can be generated automatically.

I also developed a declarative metaprogramming framework that makes it possible to reason about Java programs. The DMP framework is based on processing of the bytecode of Java classes, which makes it an ideal foundation for a dynamic aspect-oriented programming system.

The AOP framework I created has several benefits against current AOP systems. At first, it supports the use of metavariables, using which pointcuts can be expressed in a much more robust way. At second, temporal relationships between join points can be expressed. Another advantage is that aspects can be weaved dynamically, at load-time or run-time. Although some of these features appeared in other AOP systems, none of them supports each of them.

Finally, I developed a controlled natural language interface for defining pointcuts. This is a novel approach, which improves the readability of pointcuts significantly and makes it possible to formulate pointcut definitions even for programmers not experienced in aspect-oriented programming at all. CNL pointcuts are not only easy-to-read but usually even shorter than equivalent pointcuts in other AOP languages.

In the followings I give a brief overview on the results achieved.

An ideal multiparadigm programming framework or language should allow intermixing the constructs of the languages without forcing the programmer to deal with the internal representation of terms in the system. I developed the Prolog4J multiparadigm programming framework, in which this internal representation remains completely hidden from the programmer. The framework integrates Prolog into Java. Additionally, I presented a tuProlog library that allows object-oriented style programming in Prolog. As a third result in this topic, I also introduced a lingual symbiosis of the Java and Prolog language, which I named Japlo [21].

In contrast with foreign language interfaces (like JPL [51] for SWI-Prolog [54]) and Prolog engines written in Java (like tuProlog [14] or JLog [29]), the Prolog4J framework exploits the advantages of Java 5. In this new generation of the Java platform many new language features were introduced which allows to build a more easy-to-use and type-safe interface over Prolog programs.

In Prolog4J a Prolog query produces a `Solution<A>` object, through which the results can be accessed. The `Solution<A>` class implements `java.lang.Iterable<A>` so that the individual solutions can be traversed by a for-each loop easily. If you are interested in the binding of another variable of the goal, other iterable objects can be achieved through the result of the query. There are helper methods in `Solution<A>` as well for collecting the solutions into specific Java collections.

Using generics and autoboxing allows the programmer to get rid of type casts and wrapping primitive values to objects or vice versa. There is also a well-defined way of converting Java objects to Prolog terms. Java objects are directly translated into tuProlog terms and vice versa. This is in contrast with P@J, which provides an additional type system over tuProlog to introduce type safety with generics. The P@J type system is a bridge between Java objects and Prolog terms, which doubles the number of conversions required and also needs more memory since an intermediate representation has also to be stored. Another advantage of Prolog4J is that the programmer can use Java types directly, so he/she does not have to deal with an intermediate type system at all.

The Prolog4J framework uses the Metadata Facility of Java 5 to specify an interface for a Prolog query. Such an interface is a method annotated by `@Goal`. The query can be specified as the argument of `@Goal`, and the variables of the goal can be bound to the formal arguments or the return value of the method by annotations as well (`@In`, `@InOut` or `@Out`). From the appearance of JDK 7, `@NonNull` annotations will be allowed to denote ground arguments, which will

allow using these methods in a more type-safe way.

Another innovative feature of Prolog4J is that the body of goal methods are generated automatically, using the Java Compiler API [38]. This feature works only with the Sun's Java compiler, however.

OOLibrary is a tuProlog library provided as part of the Prolog4J framework. The library has dual purpose. Its main goal is to allow object-oriented style programming in Prolog. The library defines a coherent type system. Over the atomic types of Prolog, the type of compound terms is defined, there are list types and generic types as well. New types can also be defined.

OOLibrary allows you to define a hierarchy of classes. Classes can have fields, and they can have one superclass (single inheritance). The purpose of this hierarchy is to introduce *object polymorphism* into Prolog: although the instances of classes are ordinary (compound) terms, the instance of a subclass can stand for an instance of one of its superclass at any place of a program.

Methods can also be defined. They are not part of the class definition. The type of their formal arguments are specified in the formal argument list. The type of the actual arguments is checked dynamically at the application of the method.

The second purpose of the library is to improve the integration between Prolog and Java. If a term which is an instance of an OOLibrary class has to be converted to Java, Prolog4J will construct an instance of the corresponding Java class annotated by `@Term`. Similarly, the objects of these classes can be converted to OOLibrary objects. This improves the integration of Java and Prolog significantly, since the same class hierarchy of the application domain can be used in both languages, and the type conversion between them happens automatically.

The Japlo language [21] provides an alternative way of Java-Prolog integration, which is one of my first results in the field. The language is an extension of Java, in which Prolog rules can be expressed with a Java-like syntax. The syntax was constructed in such a way that it should be easy to learn for programmers experienced in Java but not in Prolog.

Many of the concepts of Java appear in the way you can formulate rules. For example, a few control structures of Java can be used (if, while, do-while), rules can have return types, like methods, and the application of such rules are expressions. On the other side, the application of rules from Java methods is also easy: at first, there is a special operator (unary `+`) for testing the existence of a solution, at second, the solutions of a rule application can be iterated through by a for-each loop.

The syntax of Japlo provides a convenient way of writing rules to be accessed

from Java. I showed that a Japlo rule can be applied in a much more concise way than JLog, if some Java objects have to be passed to it. Another advantage of Japlo is that it provides static type checking even for rules.

I developed a logic metaprogramming framework for Java. The framework is based on OOLibrary and Prolog4J. It has two layers. The low-level layer stores the metainformation of the program as facts (meta-database). The high-level layer defines OOLibrary classes and methods for examining the contents of the database.

Naturally, new methods can be defined over the existing ones, freely. So, logic-based analysis of the program is possible, for example for detecting design patterns in the program and identifying the role of the individual program elements in the pattern.

To retrieve the required metainformation, only a query has to be formulated, which specifies the properties of the required data and logical relationships between them. Within the framework the body of methods can be achieved in the form of an OOLibrary object so that they can be transformed with the means of Prolog. Moreover, the metainformation can be extracted from the bytecode, not the source, which allows to use the framework in a dynamic way.

I developed an *aspect-oriented programming* framework upon the aforementioned DMP framework. It makes it possible for the programmer to refer to events of the execution of a program and makes certain subprograms run automatically when these events occur (*implicit invocation*). The descriptions that pick out the points of the program flow (called *pointcuts*) can be formulated as Prolog queries on the code. Besides of the predicates exploring the program code, spatial and temporal relations between events can also be expressed. Pointcuts can be specified as annotations on methods.

I showed that the expressivity of the pointcuts is high for two reasons. At first, since pointcuts can refer to the program elements by metavariables and the properties of these elements and their relationships can be examined through the DMP framework. At second, since temporal relationships between the events can be expressed.

The use of metavariables increases the robustness of pointcuts in terms of the program evolution because it allows to describe the pointcuts by their semantics instead of their syntactic appearance in the code.

Another advantage of my AOP system is that it is based on a mainstream programming language, not a prototype language. Moreover, aspects are weaved into the bytecode of classes, not their source. This made it possible to weave aspects at load-time. Using the Java Debugger Interface (JDI), run-time weaving is also possible. I showed in [20] that in certain cases run-time weaving does

not decrease the speed of a program but it may even improve it.

I developed a controlled natural language interface over the pointcut definition language of the AOP system. Controlled natural languages are perfectly suitable for defining pointcuts, for several reasons:

- They describe a specific domain. Here, the domain is not the application of the program but metaprogramming as such.
- Pointcut designators have well-defined, simple structure, which provides two benefits:
  - The programmer will probably not constrained of the rigid structure of the controlled language.
  - The language can be constructed quite easily. (An implementation issue only.)
- Pointcuts should be defined in a declarative way for avoiding fragility, and declarations are the most basic form of communication in natural languages.

Although natural language texts are typically more verbose than formal descriptions, I built several grammatical techniques in the language that keep pointcut definitions as short as possible. I showed that the CNL formalism is more concise than the formalism of AspectJ or ALPHA.

Although natural languages texts are inherently ambiguous, this is (typically) not true for controlled natural languages. The grammar of the language I proposed does not allow any ambiguity.

The CNL interface provides other benefits as well that are much more essential than conciseness. At first, CNL pointcut definitions are easy to read or write even for those who are not familiar with aspect-oriented programming at all. At second, it hides the internal use of Prolog definitively, saving the programmer learning and using a second programming language in his/her program.

Chapter 8

Összefoglalás

Az értekezés eredményeit összefoglalva kijelenthető, hogy az általam kifejlesztett Java/Prolog többparadigmás programozási keretrendszer (Prolog4J) könnyebben használható mint a jelenleg elérhető hasonló célkitűzésű projektek, több okból is. Egyrészt, mivel a rendszerem kihasználja a Java 5 új elemeit (for-each ciklus, automatikus csomagolás, annotációk). Másrészt a Java objektumok és Prolog termek között automatikus konverziót végez összetett típusok esetén is anélkül, hogy új típusrendszert vezetne be. Végül, mivel lehetővé teszi egyszerű programozói felület kialakítását Prolog lekérdezések fölé annotált Java metódusok formájában, melyek törzsét a rendszer automatikusan generálja.

Szintén kifejlesztettem egy deklaratív metaprogramozási keretrendszert, amely lehetővé teszi Java programokra vonatkozó állítások megfogalmazását. A DMP rendszer Java osztályok bájtkódjának feldolgozására épül, így ideális alapja lehet egy dinamikus, aspektusorientált programozási rendszernek.

Az általam létrehozott AOP keretrendszernek számos előnye van a jelenlegi rendszerekkel szemben. Egyrészt, támogatja a metaváltozók használatát, ami a vágáspontok megadásának sokkal robusztusabb módját teszi lehetővé. Másrészt a csatlakozási pontok időbeli viszonya is kifejezhető. További előny, hogy az aspektusok szövése dinamikusan, betöltési vagy futási időben történik. Bár ezen képességek külön-külön már megjelentek AOP rendszerekben, egyikük sem támogatja ezek mindegyikét.

Végül, kialakítottam egy kontrollált természetes nyelvi felületet a vágáspontok megadására. Ez egy újszerű megközelítés, amely jelentősen javítja a vágáspontok olvashatóságát, és lehetővé teszi vágáspontok megadását olyanok számára is, akik egyáltalán nem jártasak az aspektusorientált programozás

terén. Az ilyen vágáspont-definíciók nem csak könnyen érthetőek, de jellemzően tömörebbek is mint az azonos jelentésű vágáspontok egyéb AOP nyelvekben.

Az alábbiakban tömören ismertetem az elért eredményeket.

Egy ideális Prolog/Java többparadigmás keretrendszernek vagy nyelvnek anélkül kell lehetővé tenni a nyelvi konstrukciók kombinálását, hogy a programozót arra kényszerítenék, hogy a termek belső reprezentációjával foglalkoznia kellene. Kifejlesztettem a Prolog4J többparadigmás programozási keretrendszert, amelyben ez a belső reprezentáció a programozó elől teljesen rejtve marad. A keretrendszer a Prolog és Java nyelvek integrációját biztosítja. Ennek részeként elkészítettem egy tuProlog programkönyvtárat, amely objektumorientált stílusú programozást tesz lehetővé Prolog programokon belül. A témakör harmadik eredményeként bemutattam a Japlo nyelvet, amely a Java és Prolog nyelvi szimbiózisa [21].

A külső nyelvi felületekkel (amilyen pl. a JPL [51, 54]) és a Java nyelven írt Prolog motorokkal (mint a tuProlog [14] vagy a JLog [29]) szemben a Prolog4J kihasználja a Java 5 előnyeit. A Java platform eme új változatában számos olyan új képességet vezettek be, amely lehetővé teszi könnyebben olvasható és típusbiztosabb felület kialakítását Prolog programok fölé.

Prolog4J-ben egy lekérdezés egy `Solution<A>` objektumot szolgáltat, amelyen keresztül az eredmények elérhetők. A `Solution<A>` osztály implementálja az `Iterable<A>` interfészt, így a megoldások könnyedén, egy for-each ciklus segítségével bejárhatók. Ha a cél egy másik változójának a kötései érdekelnek minket, a kapott objektumból egyéb iterálható objektumokat is megkaphatunk. A `Solution<A>` osztály segédmetódusokat is tartalmaz, amelyekkel a megoldások különböző kollekciókba gyűjthetők.

A generikusok és az automatikus csomagolás használatával a programozó megszabadul a típuskonverzióktól, és a primitív értékek, illetve csomagoló objektumok közötti oda-vissza átalakításoktól. A Java objektumok Prolog termmé alakításának jól definiált módja van. A Java objektumok közvetlenül tuProlog termekké lesznek konvertálva és viszont. Ez másképp van a P@J keretrendszer esetén, amely egy új, köztes típusrendszert biztosít a tuProlog típusai fölé azért, hogy a generikusok segítségével nagyobb típusbiztonságot érjen el. A P@J típusrendszere tehát egy hidat képez a Java objektumok és a Prolog termek között, ami megduplázza a szükséges konverziók számát, és némi többlet tárat is igényel, hiszen a köztes reprezentációt szintén tárolni kell. A Prolog4J megvalósításnak további előnye, hogy a programozó közvetlenül Java típusokat használhat, így nem kell foglalkoznia semmiféle köztes típusrendszerrel egyáltalán.

A Prolog4J keretrendszer kihasználja a Java 5 metaadatkezelő képességét

a Prolog lekérdezésekhez programozói felület megadására. Az ilyen felületet `@Goal` annotációval ellátott metódusok jelentik. A lekérdezés az annotáció paramétereként adható meg, és a lekérdezés változói szintén annotációk segítségével rendelhetők a formális paraméterekhez, illetve visszatérési értékhez (`@In`, `@InOut` vagy `@Out`). A JDK 7 megjelenésétől kezdve a `@NonNull` annotáció is használható lesz alap (ground) argumentumok megjelölésére, ami ezeknek a metódusoknak egy még típusbiztosabb használatát teszi majd lehetővé.

A Prolog4J további innovatív jellemzője, hogy a cél metódusok törzsét automatikusan generálja a Java Compiler API-n keresztül [38]. Ez a képesség csak a Sun Java fordítóját használva működik.

Az OOLibrary egy tuProlog programkönyvtár, amely egyben a Prolog4J rendszer része. A programkönyvtárnak kettős célja van. Fő célja objektumorientált stílusú programozás lehetővé tétele Prologban. A programkönyvtár egységes típusrendszert definiál. A Prolog atomi típusain túl az összetett termekhez is típust rendel, vannak lista típusok és generikus típusok is. Új típusok is definiálhatók.

Az OOLibrary segítségével osztályhierarchia definiálható. Az osztályoknak lehetnek mezőik, és lehet egy ősosztályuk (egyszeres öröklődés). Az osztályhierarchia célja az objektum polimorfizmus bevezetése Prologba: bár az osztályok példányai közönséges (összetett) termek, egy osztály példánya szerepelhet a program bármely pontján, ahol az ősosztály objektuma szerepelhet, és emellett a viselkedése saját osztályára jellemző marad.

Metódusokat is definiálhatunk. Ezek nem képezik az osztálydefiníció részét. A formális paramétereik típusa meg van adva a formális paraméterlistán. Az aktuális paraméterek típusa a metódus hívásakor kerül ellenőrzésre.

A programkönyvtár másodlagos célja a Prolog és Java közötti integráció javítása. Ha olyan termet kell Java objektummá konvertálni, amely egy OOLibrary osztálynak példánya, akkor a Prolog4J a megfelelő, `@Term` annotációval ellátott Java osztály példányát fogja létrehozni. Hasonlóan, ezen osztályok objektumai OOLibrary objektumokká konvertálhatók. Ez jelentősen javítja a Java és Prolog nyelvek kapcsolatát, mivel az alkalmazási szakterület ugyanazon osztályhierarchiája használható mindkét nyelvben, és a típuskonverzió közöttük automatikusan megtörténik.

A Japlo nyelv [21] a Java-Prolog integráció alternatív módját biztosítja, ami a területen elért első eredményeim egyike. A nyelv a Java nyelv kiterjesztése, amelyben Prolog szabályok Java-szerű szintaxissal adhatók meg. A nyelvtant úgy alkottam meg, hogy olyanok számára is könnyen elsajátítható legyen, akik a Java használatában járatosak, a Prologéban viszont nem.

Számos Java elv használható a szabályok megfogalmazásakor. Például

használható néhány vezérlési szerkezet (`if`, `while`, `do-while`), a szabályoknak visszatérési értéke lehet, akárcsak a metódusoknak, és az ilyen szabályok alkalmazása kifejezésben is lehetséges. A másik oldalt tekintve, a szabályok egyszerűen alkalmazhatók Javából: egyrészt az unáris `+` operátor segítségével a megoldás létezése dönthető el, másrészt a szabály alkalmazásával kapott megoldások egy for-each ciklussal roppant egyszerűen bejárhatók.

A Japlo nyelvtana kényelmes módot biztosít Javából elérhető szabályok írására. Megmutattam, hogy egy Japlo szabály sokkal tömörebben használható, mint ugyanez JLogban, ha a szabály alkalmazásának eredménye futás közben előálló értékektől függ. A Japlo másik előnye, hogy statikus típusellenőrzést tesz lehetővé a szabályokra.

Kifejlesztettem egy logikai metaprogramozási keretrendszert Javához. A keretrendszer alapja a Prolog4J és az OOLibrary. Két rétege van. Az alacsony szintű réteg a program metainformációit tények formájában rögzíti (metaadatbázis). A magas szintű réteg OOLibrary osztályokat és metódusokat definiál az adatbázis tartalmának vizsgálata céljából.

Természetesen új metódusok is szabadon definiálhatók a meglévőek segítségével. Ez lehetővé teszi a program logikai elemzését, például tervezési minták felismerését a programban, és az egyes programelemek mintában betöltött szerepének azonosítását.

A keresett metainformáció eléréséhez csupán egy lekérdezést kell megadni, amely megadja a kívánt adat tulajdonságait és a közöttük rejlő logikai összefüggéseket. A keretrendszeren belül elérhető a metódusok törzse is OOLibrary objektum formájában, amely így a Prolog eszközeivel átalakítható. Továbbá a metainformáció a bájtkódból lesz kinyerve, nem a forrásból, ami a keretrendszer dinamikusabb felhasználását teszi lehetővé.

Kifejlesztettem egy aspektusorientált keretrendszert az előbb tárgyalt DMP keretrendszer fölé. Ez lehetővé teszi, hogy a programozó a program végrehajtásának eseményeire hivatkozzon, és így bizonyos alprogramok (ún. *javaslatok)* meghívását ehhez kösse. A program futásának pontjait kijelölő leírások (a vágáspontok) megadása Prolog lekérdezések formájában történik. A program kódját felderítő predikátumok mellett az események térbeli és időbeli viszonyai szinték kifejezhetők. A vágáspontokat annotációk formájában lehet metódusokhoz kötni.

Megmutattam, hogy a vágáspontok kifejezőereje magas, két okból is. Egyrészt, mivel a vágáspontok a program elemeire metaváltozók segítségével hivatkozhatnak, és ezen elemek tulajdonságai és a köztük lévő kapcsolatok a DMP keretrendszer eszközeivel vizsgálhatók. Másrészt azért, mivel az események közötti időbeli viszonyok is kifejezhetők.

A metaváltozók használata növeli a vágáspontok robusztusságát a program evolúciója tekintetében, mivel lehetővé teszi, hogy a vágáspontokat szemantikájuk segítségével írjuk le ahelyett, hogy a kódbeli szintaktikai megjelenésükre hivatkoznánk.

Az AOP rendszerem további előnye, hogy napjaink talán legelterjedtebb programozási nyelvére épül, nem pedig egy kutatási prototípus nyelvre. (Nem elvitatva a prototípus nyelvek fontosságát.) Emellett az aspektusok az osztályok bájtkódjába lesznek beleszőve, nem a forrásukba. Ez lehetővé teszi a betöltési idejű szövést. A Java Debugger Interface (JDI) felületen keresztül futási szövés szintén lehetséges. Egy korábbi cikkemben megmutattam, hogy a futási idejű szövés akár javíthatja is a program teljesítményét a betöltési idejűhöz képest.

Kifejlesztettem egy kontrollált természetes nyelvi felületet az AOP keretrendszer vágáspont definíciós nyelve fölé. A kontrollált természetes nyelvek tökéletesen alkalmasak vágáspontok megadására, számos okból:

- Speciális ,,szakterületet" képeznek. Itt a szakterület nem a program alkalmazási szakterülete, hanem a metaprogramozás, mint olyan.
- A vágáspontoknak jól definiált, egyszerű szerkezete van, ami két előnnyel is jár:
  - A programozót nem fogja megkötni a kontrollált nyelv merev szerkezete.
  - A nyelv viszonylag könnyen megalkotható. (Ez csupán megvalósítási kérdés.)
- A vágáspontokat deklaratív módon kell megadni ahhoz, hogy azok törékenységét elkerüljük. A deklarációk (kijelentések) a természetes nyelvek legalapvetőbb kommunikációs formái.

Bár a természetes nyelvű szövegek jellemzően bőbeszédűbbek a formális leírásoknál, számos nyelvtani technikát építettem a nyelvbe, amelyekkel a vágáspont-definíciók – amennyire csak lehet – tömören tarthatók. Megmutattam, hogy a kontrollált nyelvi megfogalmazás tömörebb, mint az azonos jelentésű definíciók AspectJ vagy ALPHA nyelveken.

Bár a természetes nyelvű szövegek eredendően többértelműek lehetnek, ez (jellemzően) nem igaz a kontrollált természetes nyelvekre. Az általam készített nyelvtan nem enged meg többértelműséget, így az utalószavak használatát sem.

A CNL felületnek egyéb előnyei is vannak, amelyek sokkal lényegesebbek a tömörségnél. Először is, a CNL vágáspont-definíciók könnyen olvashatók vagy

akár megfogalmazhatók olyanok számára is, akik egyáltalán nem jártasak az aspektusorientált programozás terén. Másrészt, elrejtik a végleg elrejtik Prolog belső használatát, ezáltal megkímélve a programozót egy másik programozási nyelv megtanulásától, illetve használatától.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects (Monographs in Computer Science)*. Springer, April 1998.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[3] James Allen. *Natural Language Understanding*. The Benjamin/Cummings Publishing Company, Redwood City, CA, second edition, 1995.

[4] Uwe Bardey. Abhängigkeitsanalyse von Softwaretransformationen. Diploma thesis, CS Dept. III, University of Bonn, Germany, Feb 2003.

[5] Joshua Bloch. JSR 175: A metadata facility for the Java programming language.
http://jcp.org/aboutJava/communityprocess/final/jsr175/index.html.

[6] Jonas Bonér. What are the key issues for commercial aop use: how does aspectwerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM.

[7] Eric Bruneton. ASM 3.0: A Java bytecode engineering library.
http://download.forge.objectweb.org/asm/asm-guide.pdf, 2007.

[8] Maurizio Cimadamore and Mirko Viroli. A Prolog-oriented extension of Java programming based on generics and annotations. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 197–202, New York, NY, USA, 2007. ACM.

[9] BinNet Corporation. BinPrologJ: a fast and flexible Prolog-in-Java. http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html.

[10] Xerox Corporation. AspectJ 5 Quick Reference. http://eclipse.org/aspectj/doc/released/quick5.pdf.

[11] Xerox Corporation. The AspectJ 5 development kit developers notebook. http://eclipse.org/aspectj/doc/released/adk15notebook/index.html.

[12] Xerox Corporation. The AspectJ$^{\mathrm{TM}}$Programming Guide. http://eclipse.org/aspectj/doc/released/progguide/index.html.

[13] Bart Demoen and Paul Tarau. jProlog home page. http://www.cs.kuleuven.be/ bmd/PrologInJava/.

[14] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[15] Sophia Drossoupolou. Lecture notes on the L2 calculus. http://www.doc.ic.ac.uk/˜scd/Teaching/L1L2.pdf.

[16] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Proceedings of TOOLS Europe 2007, Journal of Object Technology*, 6(7), 2007.

[17] Michael D. Ernst. Annotations on Java types – Java Specification Request 308 working document. http://pag.csail.mit.edu/jsr308/.

[18] Miklos Espak. Soul: új metaobjektum-protokoll Javához. Diploma thesis, University of Debrecen, Hungary, May 2001.

[19] Miklos Espak. Aspektusorientált programozás megvalósítása metaobjektum-protokollal. http://www.date.hu/if2002, 2002.

[20] Miklos Espak. Improving Efficiency by Weaving at Run-time. In *In 5th GPCE Young Researchers Workshop 2003*, pages 415–421, 2003. http://citeseer.ist.psu.edu/636848.html.

[21] Miklos Espak. Japlo: Rule-based Programming on Java. *Journal of Universal Computer Science*, 12(9):1177–1189, 2006.

[22] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?
http://www.martinfowler.com/articles/languageWorkbench.html.

[23] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[24] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English (ACE) Language Manual, Version 3.0. Technical Report 99.03, Department of Computer Science, University of Zurich, August 1999.

[25] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312, New York, NY, USA, 2003. ACM.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[27] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, September 1993.

[28] Object Management Group. Semantics of Business Vocabulary and Business Rules (SBVBR), v1.0.
http://www.omg.org/spec/SBVR/1.0/PDF, 01 2008.

[29] Glendon Holst. JLog - Prolog in Java.
http://jlogic.sourceforge.net.

[30] Franz Inc. Allegro Prolog Documentation.
http://www.franz.com/support/documentation/7.0/doc/prolog.html.

[31] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. A Model-Driven Pointcut Language for More Robust Pointcuts. In *Proceedings of the 4th International AOSD Workshop on Software Engineering Properties of Languages for Aspect Technology (SPLAT'06)*, Bonn, Germany, mar 2006.

[32] Gregor Kiczales. AspectJ(tm): Aspect-Oriented Programming in Java. In *NODe '02: Revised Papers from the International Conference NetObject-Days on Objects, Components, Architectures, Services, and Applications for a Networked World*, page 1, London, UK, 2003. Springer-Verlag.

[33] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[34] Günter Kniesel. Conditional Transformation. http://roots.iai.uni-bonn.de/research/jtransformer/cts, 2003.

[35] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond AOP: toward naturalistic programming. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 198–207, New York, NY, USA, 2003. ACM.

[36] Edward McCormick and Kris De Volder. JQuery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10, New York, NY, USA, 2004. ACM.

[37] Tom Mens, Roel Wuyts, Kris De Volder, and Kim Mens. Declarative Meta Programming to Support Software Development: Workshop Report. *SIGSOFT Softw. Eng. Notes*, 28(2):1, 2003.

[38] Sun Microsystems. JSR-000199 JavaTM Compiler API. http://jcp.org/aboutJava/communityprocess/final/jsr199/index.html.

[39] Sun Microsystems. JSR-000269 Pluggable Annotation Processing API. http://jcp.org/aboutJava/communityprocess/final/jsr269/index.html.

[40] Sun Microsystems. Java 2 Platform Standard Ed. 5.0 API. http://java.sun.com/j2se/1.5.0/docs/api/, 2004.

[41] Jerome Miecznikowski and Laurie Hendren. Decompiling Java using staged encapsulation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 368, Washington, DC, USA, 2001. IEEE Computer Society.

[42] Paolo Moura. *LogTalk*. PhD thesis, Universidade da Beira Interior, 2003.

[43] John Von Neumann. *The Computer and the Brain*. Yale University Press, New Haven, CT, USA, 2000. Foreword By-Churchland, Paul M. and Preface By-Neumann, Klara Von.

[44] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and PROLOG*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[45] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. 1992.

[46] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[47] European Association of Aerospace Industries (AECMA). AECMA Simplified English (SE) Guide. Available through various AECMA-appointed publishers, (The current version is Issue 1 Revision 2; SE has been regularly updated since 1986.), 2001.

[48] Harold Ossher and Peri Tarr. Hyper/J(tm): Multi-Dimensional Separation of Concerns for Java(tm). *Software Engineering, International Conference on*, 0:0821, 2001.

[49] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.

[50] Nestor Rychtyckyj. Standard Language at Ford Motor Company: A Case Study in Controlled Language Development and Deployment. 2006.

[51] Paul Singleton, Fred Dushin, and Jan Wielemaker. JPL: A bidirectional Prolog/Java interface.
http://www.swi-prolog.org/packages/jpl/.

[52] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM*, 44(10):95–97, 2001.

[53] Kris De Volder. Logic Programming and Logic Meta Programming in TyRuBa.
http://tyruba.sourceforge.net/tyruba_language_reference.html.

[54] Jan Wielemaker. SWI-Prolog 5.4.3 Reference Manual. http://www.swi-prolog.org, 2004.

[55] Tobias Windeln. LogicAJ - Eine Erweiterung von AspectJ um logische Meta-Programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.