Vrije Universiteit Brussel
Programming Technology Laboratory
Faculteit Wetenschappen - Departement Informatica
(1998)

Type-Oriented Logic Meta Programming

Kris De Volder

Proefschrift ingediend met het oog op het behalen van de graad van Doktor in de Wetenschappen

Promotor: Prof. Dr. Theo D'Hondt

Acknowledgments

I thank my thesis advisor Prof. Dr. Theo D'Hondt. I could not have wished for a better advisor. He provided me with a stable low-pressure working environment. He was always stimulating and encouraging, providing hints, gently pushing me in one direction or another, but always in such a way that my personal freedom to choose the research direction that interested me most was left open. I am very grateful to have received and been trusted with this kind of freedom. After some time of wandering around from one research topic to the next it allowed me to find a topic I am really passionate about, and to finally produce this dissertation.

Another person that I am greatly indebted to is Wolfgang De Meuter. He was there exactly at the right time to give me the proverbial "kick in the butt" or "pep talk" when I needed it.

Thanks goes also to all of my other colleagues, Carine Lucas, Mark Willems, Koen De Hondt, Kim Mens, Tom Mens, Tom Lenaerts, Tom Tourwé, Werner Van Belle and Patrick Steyaert. They helped me indirectly by collectively taking some of my teaching and organizational responsibilities onto themselves, during the last year of my Ph.D. This allowed me to spend more time on my dissertation.

Thanks also to all the people who helped in improving the quality and clarity of my writing by proofreading and commenting on preliminary versions of this dissertation. Tom Tourwé, Kim Mens, Carine Lucas, Wolfgang De Meuter, Tom Mens, Roel Wuyts, Johan Fabry and Kim Bruce have made a considerable effort in proofreading.

I also thank my friends, Yves Samyn, Bruno Van Bogaert, Koen Vermeulen and Schonbrodt Yong Lak, for providing valuable moral support at times I really needed it.

I thank my parents. They have helped and supported me in so many ways that I can hardly start mentioning it here.

I also thank my sister and Stefano Corvaglia. They always made me feel welcome, so that I always had a place to go during weekends to take some time off and relax.

Finally, there are so many people that have either directly or indirectly supported me at one time or another, and contributed to the realization of this Ph.D. that inevitably there will be some that I have still forgotten about. My gratitude goes out to all of them, and I apologize for not thanking them more explicitly.

Contents

Chapter 1

Introduction

1.1 Thesis

This dissertation will show that undecidable and ambiguous type systems have useful applications and should therefore be considered viable options for future statically typed object-oriented languages.

There is an overwhelming number of static type systems floating around, for all kinds of programming languages: purely functional languages, imperative procedural languages, object-oriented languages, logic languages etc. Object-oriented languages themselves represent a vastly diversified group of sub paradigms: multi methods, class-based languages, prototype-based languages, etc.

The number of different static type systems is therefore so immense that it is virtually impossible to oversee. All of them however seem to subscribe to the solid convictions that:

1. A static type system should be decidable.
2. A static type system should ensure that ambiguously typed programs are either impossible to express or rejected by the type checker.

These restrictions seem to have been with type systems forever, almost without questioning them. Undecidable or ambiguous type systems are considered, almost by definition, to be completely useless.

This dissertation questions that these seemingly sound principles should be accepted out of hand. We will show that undecidability and ambiguity of type systems is not "by definition" a bad thing. Ensuring unambiguity and decidability of typing imposes strong restrictions on the expressiveness of the type system. When abandoning the restrictions, the expressiveness of the type system can be increased to the level of a real programming language, thus lifting its capability for active manipulation of static types to an entirely new level. We will show by means of examples that this extra expressiveness *does* have very useful and interesting applications.

1.2 A Type Language

What is a static type system and what does it do? Basically, a static type system is a mechanism with which static types are attached to variables and expressions in a programming language. The static type of an expression or a variable corresponds to a set which is an upper bound estimate of the possible values the expression or variable may actually return or hold at runtime. These sets are denoted, by type expression, in a formalism for denoting sets of runtime values. We call this formalism a *type language*. As an example, the type language for Pascal [JW85] comprises type expressions such as:

```
INTEGER
REAL
ARRAY [1..3] OF INTEGER
```

There are type systems which require the programmer to "estimate runtime values" himself and declare these estimates explicitly in so called type declarations. All the system will do is verify whether the labeling the programmer provides is a valid labeling, without any type conflicts. Such systems are called type checkers. Other systems allow partial type declarations or even omitting type declarations altogether. These systems do the estimating work themselves, they infer the missing static types from the source code by means of some algorithm. These systems are therefore called type inferencers. One hundred percent pure type checkers don't really exist since it is too impractical to manually annotate every expression with a type. Usually only key points such as variables are annotated explicitly and the rest is inferred, even in systems typically called type checkers. In both checkers and inferencers the end result is the same: static upper bound estimates are placed on expressions and variables. The flexibility and expressive power of the type system is mostly derived from the expressive power of the type language which is used for denoting the estimated sets. The more accurately these sets can be denoted, the more flexible the type system is. Therefore type languages are becoming more and more sophisticated in order to be able to give better static upper bound approximations of the sets of runtime values.

As type languages become more sophisticated they start to look more and more like programming languages for manipulating and constructing types. The type manipulating programs written in the type language treat static types as data structures and are "run by the compiler", at compile time, as part of the type checking or inferencing process. Consider for example what is happening to object-oriented languages. Object-oriented languages are particularly difficult to statically type and the typing problem for them is far from being solved [FM96]. Type systems and type languages for object orientation are very diverse and still in constant evolution. We pick one particular language, Java [GJS97], because we think it is representative for a large class of mainstream and widely used OO programming languages. The type system for Java is a nice, simple, pretty conventional and easy to understand static type system for a class-based language. The type language is too simple in fact to express adequate estimates. The programmer often has no choice but to override the static type system with an explicit dynamically checked type cast to be able to say or do what he wants to. Already several proposals for increasing the expressive power of Java's type language are emerging to alleviate this problem [Tho97, OW97, AFM97, MBL97, BOW98]. Most of these proposals offer the notion of parametric types as part of the solution [OW97, AFM97, MBL97, BOW98]. A parametric type is a type which is only partially given. It has one or more type parameters which are filled in when the type is "instantiated". A parametric type is very much like a function which constructs a new type from a number of existing types given to it as arguments. So in a way, a parametric type system offers a limited functional programming language for manipulating static types at compile time. Declaring a parametric type is like a function declaration. Instantiating it is like a function invocation. This example clearly shows that the type languages are starting to look more and more like programming languages. Other examples can be found. Functional programming languages like Haskell [PH$^+$97] and Gofer [Jon95] have very sophisticated type systems that combine parametric and qualified types [Kae88, WB89, Jon94a, Jon94b]. Qualified types are particularly interesting, since they allow writing type declarations which look very much like logic inference rules, with conditions and a conclusion inferring implementations for types. In a way Gofer and Haskell have a logic programming language embedded in their type system. This is not a full-fledged logic programming language however because of restrictions imposed by the type system to avoid ambiguity and ensure termination.

1.3 Passive Versus Active Usage of Types

The main argumentation to support the thesis presented in this dissertation depends on illustrating the importance and usefulness of actively using static types at compile time. Therefore, let us begin by explaining what is meant by "active usage" of static types, in contrast to the traditional "passive usage".

1.3.1 Passive Usage

Originally static type systems were meant to increase the robustness of programs. They can detect a whole class of errors, i.e. type errors, *before* the program is running. Programs written in statically typed languages are more robust because when running, they are already guaranteed to be free of type errors. A nice bonus that comes with static types is that they also constitute valuable documentation. The static types of variables, procedures, functions, methods, etc. reveal a lot about how they are intended to be used. The greatest virtue of static types as documentation is that they are verified and enforced by the type system. This makes static types as documentation far superior to informal comments which tend to get "out of sync" as programs evolve. This use of static types for increasing robustness and as documentation is passive in nature in the sense that types do not play an active role in programs but only "describe" them.

1.3.2 Active Usage

There is an evolution towards a complementary, more active role of static types. More expressive type languages, which are starting to look more and more like programming languages, allow active manipulation of types as data at compile time. This is a new aspect of using static types, complementary to their traditional descriptive usage. That this is indeed an important aspect can be concluded for example from the successful construction of modular interpreters [LHJ95] which was made possible by the expressiveness of parametric and qualified types. Note that construction of modular interpreters is a *very* hard problem which has been eluding denotational semanticists for years [Mos90]. Together with the use of monads [Wad92, JD93, Ste94a, LHJ95] as an elegant abstraction for "computations", active usage of static types was the key to the solution. Implementations for interpreters are generated almost magically, puzzled together from pieces of abstract code which are neatly fitted together by the type inference system on the basis of static type information.

1.4 Usefulness of Undecidable and Ambiguous Type Systems

In this section we will present some tentative and informal arguments why we think that the issues of decidability and unambiguity of typing should not be taken as prerequisites, and why undecidable and ambiguous type systems can be potentially useful and should also be considered.

1.4.1 Undecidability

The principle that a static type system should be decidable is very sound when static types play only a descriptive role. However, as the importance of actively manipulating types increases, this restriction must be questioned again. Clinging to decidability implies that the type language can never evolve into a real (Turing-complete) programming language. Would the type language become Turing complete, it becomes possible to write non-terminating type programs and therefore termination of the type checker or inferencer cannot be guaranteed and the type system is no longer decidable. Is this necessarily a bad thing? We do not think so. We are assuming that active type manipulation by type programs is what we *want* to do. We *do* want to write "real" type programs which are executed as part of the type checking or inferencing process. If this is what we want to do, then evidently we need a real programming language to write our programs with. Not surprisingly we also get greater responsibility. As "type programmers" we must make sure that our type programs do not contain "bugs" which make them go into infinite loops.

1.4.2 Unambiguity

It is harder to argue that ambiguously typed programs may have their use, especially in the presence of a feature such as overloading. In a system with overloading, ambiguous typing has implications on the semantic level. Overloading means that static argument types and/or return types[1] affect which method will be invoked at runtime. Therefore an ambiguous static type means that the semantics of the program may also be ambiguous. Therefore one usually argues that ambiguously typed programs must be rejected simply because they have unclear semantics.

We think this line of reasoning is a bit too simple. Especially in object-oriented languages where great value is placed upon polymorphism. We think that this "unclear semantics" may actually be regarded as a form of polymorphism which can be put to good use.

Also, the restrictions built into a type system in order to avoid ambiguity of typing can be quite severe and incur a heavy loss of flexibility and expressiveness of the type language. We will see excellent examples that illustrate this point in chapter 2 which discusses the programming language Gofer and its type system. Gofer's type language has features which make it look a lot like a declarative logic programming language. Prolog derives most of its expressive power from its ability to deal with "ambiguous code". The Prolog interpreter deals with this ambiguity by backtracking over the alternatives. This implicit backtracking mechanism is where Prolog gets its expressive power. Compared to a Prolog interpreter, Gofer's type language is considerably less expressive and never has to resort to backtracking because of the non-ambiguity related restrictions built into it.

[1]The functional languages Haskell and Gofer for example allow overloading on the return type of a function.

1.5 Problems with Object-Oriented Type Systems

From here on we are going to focus mainly on object-oriented type systems, as we are mostly interested in object-oriented languages. However, when we think it is relevant we will sometimes refer to other type systems for functional or procedural languages as well. In this dissertation many arguments and examples will be given in the context of the object-oriented language Java. An argumentation why we have chosen Java can be found in section 1.7.2.

The first object-oriented language, Simula [DN67, BDMN73], was statically typed. So from the very birth of object orientation, research effort has been put into the design and implementation of static type systems. Surprisingly, after more than 30 years *the* type system for object-oriented languages has not been discovered yet. Type systems for object orientation are abundant and still in constant evolution. How is this possible?

Type systems for object orientation are very difficult to design because object-oriented languages are a very complex mix of features such as inheritance, dynamic method dispatch, encapsulation and subtype polymorphism. Fisher and Mitchell [FM96] give an excellent overview of the issues involved in designing object-oriented type systems.

From the start there have been problems with object-oriented typing. Major type insecurities were discovered in historically significant type systems such as the Simula [BDMN73] type system and the type system which was developed for Smalltalk[GK76, GR81] by Borning and Ingalls [BI82]. This means that these type systems infer or allow declaration of "incorrect estimates" of possible runtime values in some cases. Consequently a program which is judged to be type correct by the type system may still occasionally send messages to objects that have no method for handling them. We consider these type insecurities no more than initial technical problems found in early systems.

A more elusive problem also mentioned in [FM96] is the feeling that existing type systems are too restrictive to allow true object-oriented programming. Polymorphism is a key concept of the object-oriented paradigm. Polymorphism in dynamically typed languages is naturally obtained through dynamic method dispatch. However, statically typing polymorphic code is very hard. The problem is to find a type language with sufficient expressive power to give estimates of possible runtime values which are accurate enough not to be too restrictive. Too primitive a type language will force the estimates to be overly conservative (i.e. the sets will contain many more elements than can actually occur at runtime). Since the static type system will only allow invocation of messages supported by all of the elements in the estimated sets this means the type system will be too restrictive. As an illustration of the problem, consider the type system for Java. This is essentially a first order type system in the sense that it does not allow type parameterization. The Java type system has problems with generic data structures such as lists, stacks and queues. The elements in these data structures have to be declared of type `Object` (the class from which all other classes inherit) to allow any class as element type. The problem with Java is its first order type language. A first order type language is too simple to discriminate between lists of elements of type A and lists of elements of type B. This forces them to be treated equally, as lists of elements of the most general type `Object`. The "solution" provided by Java is to allow an explicit downcast from `Object` to a more specific type. In order to ensure type safety these downcasts will be checked at runtime. Java extensions such as Pizza provide a better solution to this particular problem. Pizza [OW97] introduces second order (i.e. parametric) types into the type language. This allows declaring parametric types such as `List<E>` which have a type parameter, `E` in the example. Such a parametric type is like a "type function" that creates a new type when instantiated.

Thus it is possible to instantiate specific list types for different element types, for example `List<A>` or `List<B>`.

In the "quest" for good static typing of polymorphic code, a plethora of different "kinds" of polymorphism have been identified and invented. The following are all different mechanisms for supporting polymorphism in static type systems which are briefly discussed in [FM96]: parametric polymorphism, subtype polymorphism, bounded universal quantification, F-bounded universal quantification, higher-order bounded universal quantification, existential quantification, F-bounded existential quantification, higher-order bounded existential quantification. There are more mechanisms to support polymorphism, which we have not mentioned yet: match-bounded polymorphism [BPF97], type substitution [PS90], signature-bounded polymorphism [LC98], virtual types [Tho97, MMP89]. This list is far from complete and still growing constantly. We can safely state that research on OO type systems is still far from being completed.

1.6 Motivating Examples

The main point put forward in this dissertation is that it is useful to consider using a "real" programming language as a type language. Already, the evolution of type languages is such that they tend to become more and more like programming languages as higher-order features (such as parametric types) are introduced. This section presents a few simple examples illustrating a lack of expressiveness in existing proposals for adding parametric types to Java. This lack of expressiveness can be regarded to be due to a deficiency of the type language as a programming language. This in turn can be brought back to the fact that type languages are usually not considered as programming languages due to the decidability and unambiguity constraints. As a result, the type language is missing features which are considered essential for programming languages.

The examples presented here are expressed by means of Pizza [OW97], one of the currently existing proposals. The examples in this chapter are only introductory and it is merely the intent to give a tentative idea of the motivations behind the writing of this dissertation. A more elaborate treatment, covering more examples and the other existing Java parametric-types proposals will be given in chapter 3.

1.6.1 Conditional Implementation

It sometimes happens that the implementation of a certain method of a parametric class invokes methods in one of its type parameters. For example, a dictionary has a type parameter for indicating the kind of keys that will be stored in the dictionary. Unlike simple containers such as stacks or queues, dictionaries need to rely to some degree on functionality provided by the keys. For example, the implementation of a dictionary needs to invoke `equal` on keys when keys are being looked up in the dictionary. In most parametric type proposals it is possible to express this by constraining the type parameter. This is called bounded parametric polymorphism [DGLM95, OW97, AFM97, MBL97]. Exactly what kind of type constraint can be expressed differs between proposals. In Pizza a type constraint can express that a type parameter should be a subtype of a type that provides the needed functionality, for example the `Equality` interface. In Pizza this implies an all or nothing situation: Either the type parameter meats the required type bound in which case the parametric class can be instantiated, or it does not and the class cannot be instantiated. Often however not all method implementations in a class depend on the type bound and it would be meaningful to be able to instantiate the parametric class albeit with a smaller interface, omitting methods which cannot be implemented because the needed constraints are not fulfilled. An example of this is a parametric array implementation in Pizza as given in figure 1.1. Note that type parameters are given between "`<>`".

```
interface Equality<This> {
  boolean equal(This e);
}

interface Searchable<El> {
  boolean contains(El e);
}

class Array<El implements Equality<El> >
implements Searchable<El>
{
  El[] contents;

  /** Construction */
  public Array(El[] init) { contents = init; }

  /** Basic Array functionality */
  public El at(int i) { return contents[i]; }
  public void atPut(int i, El e) { contents[i]=e; }
  public int length() { return contents.length; }

  /** Searchable Interface */
  public boolean contains(El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equal(at(i++));
    return found;
  }
}
```

Figure 1.1: A Parametric Array Class in Pizza

The example in figure 1.1 still deserves some explanation. First note the `Equality` interface. This interface has a type parameter `This` which is supposed to be filled in with the type of the class on which it is implemented. Using a type parameter in this way is a programming technique to deal with binary methods and simulate F-Bounded polymorphism [CCH$^+$89].

Now let us come back to what we really wanted to illustrate with this example. The `Array` implementation as presented has a type parameter for its elements that is restricted to classes implementing `Equality` in order to be able to implement `Searchable`. Suppose however we wish to use an array simply as storage structure somewhere and we do not use the `Searchable` interface. In this case we want to be able to instantiate it with any class as element type even a class not implementing `Equality`.

It could be argued that the example given above can be expressed by splitting up the functionality into several classes linked through inheritance. This solution is not satisfactory because it tangles up the class tree and becomes unmanageable with multiple dependencies in a single class. In that case it would require some kind of multiple inheritance or a proliferation of subclasses for all possible extensions and variations of the base class.

We consider the problem illustrated in this examples as due to the type language lacking some kind of `if-then-else` construct for making decisions. Any real programming language has an `if-then-else` in some form or other[2]. Type languages are not regarded as programming languages and usually do not provide features for making real decisions. For example, the type language of bounded parametric polymorphism in a way has some kind of `if`, but it is only an `if-then`, not an `if-then-else`. We can check that `El` implements `Equality` but we can only specify what to do when the check succeeds and have no control over what happens when the check fails.

Another example of the same problem (i.e. "missing `if-then-else`") is a class whose implementation varies depending on a type parameter. Consider a `Dictionary<Key,Value>` class. Typically this imposes the type constraint that `Key` should be `Hashable`. However it makes perfect sense to allow dictionaries whose keys are not hashable but merely provide an `equal` method (i.e. implement `Equality`). In this case we just provide a different implementation which uses a linked list as internal storage rather than a hash-table. Again, to be able to express this we need an `if-then-else` construct of some kind.

1.6.2 Positioning Abstract Code

Abstract code often only depends on interfaces and purposefully ignores the implementation details of a specific class. Hence it should be possible to write abstract code independent of the class hierarchy. In most class-based languages this is impossible because method implementations are associated with a specific class. Therefore, abstract code usually ends up in an abstract class which depends on subclasses to fill in the implementation details. Consider the abstract class in figure 1.2 implementing the `Searchable` interface for collections that provide a way of enumerating their elements. The problem with this abstract class is that it is unclear where to insert it into the class tree. The problem becomes exponentially worse when there are several abstract interface implementations.

A better, more "Type-Oriented Programming" flavored solution can be accomplished in some parametric type systems by simulating a *mixin* as a parametric class inheriting from one of its parameters. Not all parametric type systems allow inheriting from type parameters because of implementation-related restrictions. Pizza for one does not allow using a type parameter in this way. For the sake of the argument we nevertheless present a hypothetical mixin solution in Pizza syntax in figure 1.3.

A "mixin class" is like a function which can create a subclass implementing the `Searchable` interface for any class meeting the required type constraint. The solution with a mixin is not ideal either. Sometimes we want to affect the appropriate classes directly rather than through subclassing. In this case mixin classes will not work. Another problem with mixin classes is that the mixin's implementation should sometimes be dependent on the base class to provide a more efficient implementation in particular cases. As an example consider a collection which stores its elements in a hash-table. It would be more efficient to implement the `SearchableMixin` by hashing rather than by enumeration. We could try to accomplish

[2]Imperative languages have an `if-then-else` statement. Functional language provide an `if` as function. Logic languages can make decisions by "pattern matching". Etc.

```
interface Enumerable<El> {
  Enumeration<El> elements();
}
interface Enumeration<El> {
  boolean hasMoreElements();
  El next();
}
abstract class Searchable< El implements Equality<El> >
implements Enumerable<El>
{ boolean contains(El e) {
    boolean found = false;
    Enumeration<El> elems = elements();
    while (!found && (elems.hasMoreElements()))
      found = e.equal(elems.next());
    return found;
  }
}
```

Figure 1.2: Abstract Class Implementation of `Searchable`

```
class SearchableMixin<Super implements
                        Enumerable<El implements Equality<El> >
                     >
extends Super
implements Searchable<El>
{ boolean contains(El e) { ... } }
```

Figure 1.3: Mixin implementation of `Searchable`

this by implementing another mixin, `HashtableSearchableMixin`, specifically for hash-tables but this would be annoying since the user of our collection library must then be told he should use this other mixin for the specific case of a hash-table. Ideally we would like the mixin `SearchableMixin` to be smart enough to decide for itself which version it should use. Regrettably the functional "programming" language implicitly present in current parametric type systems lacks an `if-then-else` to make the decision.

1.7 Motivation of Choices

In order to arrive at a concrete system and delimit the area of research somewhat we have made a number of choices and imposed a number of restrictions. The following subsections describe and motivate these.

1.7.1 How we will Prove our Thesis

This dissertation will show that undecidable and ambiguous type systems have useful applications and should therefore be considered to be viable options for future statically typed OO languages.

Traditionally, type systems have been included into programming languages for the following reasons [MMMP90, Mit96a]:

documentation Static types improve the readability of programs.

robustness Early detection of "type errors".

efficiency Optimizations based on static type knowledge.

The aforementioned uses are passive in the sense that types have mostly a descriptive nature. This makes them great as documentation. The type checker verifies whether the description is a consistent one and this increases robustness by eliminating type errors. Optimizers also make use of the descriptive nature of types to perform optimizations.

In this dissertation we want to draw attention to the potential of "actively" using static types, writing real programs that manipulate types: consult them, construct them, or make decisions regarding interfaces or internal representations of classes. In short, we want to draw attention to the potential of active manipulation of types by real "type programs". It is in this active type manipulation that lies the currently unharvested potential which makes undecidable and ambiguous type systems potentially useful.

Because of limited resources we have been forced to take a less than perfect approach. We did not implement a type system which has a Turing-complete type language as one might expect us to. Type systems for OO languages being very complicated artifacts, both theoretically and implementation wise, this was not a feasible option to us. Instead we implemented a preprocessor which generates code for an existing statically typed OO language: Java. We do not do any type checking of our own. What our system does offer as part of the "preprocessor" is a Turing-complete logic programming language with which it is possible to actively manipulate types and pieces of source code. The checking aspect of the system is completely ignored by the system. Potential "type errors" are only detected as far as the generated output code violates the Java typing rules and are detected by the Java compiler. Clearly this is not what one wants for a production-level environment. However, since the added potential of having a Turing-complete type language is in active type manipulation rather than in the type checking aspect of the system, this will be sufficient to prove our point.

1.7.2 Java as Base Language

In order to arrive at a concrete system we had to pick a particular base language for our experiments. We have chosen the Java language. There are several reasons for choosing Java. One of them is of course that Java is very popular at the moment. However, this is not the only reason. We wanted our experiments to be conducted in the context of a widely accepted programming language platform and Java certainly meets this requirement. At the same time Java is also fairly simple and fairly well designed. Further, it has a static type system, which is a must since we want to manipulate static type information. Also, the notion of interfaces in Java seems a natural candidate for describing object types. TyRuBa, the concrete system we will propose, allows writing meta programs that talk about classes and interfaces, and about how the former implement the latter.

1.7.3 Logic Meta Language

The meta language chosen for manipulating types at compile time is a Prolog-like [CM81, SS94] logic language. The reason for choosing a logic language is that we believe it offers the right mode of expression for type manipulation. We can give some initial indications that this is true. One such indication is the Gofer type system [Jon95]. The way type classes can be used in Gofer has a very strong resemblance to writing logic Horn clauses. Gofer's type system is a good guideline as it is one of the most mature, sophisticated, flexible and expressive type systems around. Another indication can be found in the way type theorists formulate the semantics of a type system: as a set of inference rules with conditions and conclusions [Sch94, Mit96b].

The Prolog-like meta language has no static type system, but is dynamically typed. This is the right choice for the experimental system we have in mind. We mainly aim at making the meta-language as expressive as possible to allow as great a freedom as possible in manipulating the static type information of the base language programs. A static type system for the meta language is a limitation of its expressiveness. Also the design of a type system for the meta language is a complicated issue in itself. The meta language is an unconventional mix of logic language with quoted Java code in it. A suitable static type system should not only type the logic code but should also keep track of the type properties of the quoted pieces of Java code. Clearly such a "mixed" type system is not a trivial matter and we consider it future work and outside the scope of this dissertation. Some speculations about type checking issues can be found in section 11.4, near the end of this dissertation.

1.8 Overview of this Dissertation

We will give a summary of the goals and boundaries of this dissertation in the form of two central research hypotheses.

In this chapter, we have shown a few examples that illustrate the apparent lack of expressiveness in existing type systems offering limited capabilities for manipulating types at compile time through the use of parametric types. These seem to offer a limited compile time functional programming language with which to manipulate types. However, mainly due to the restrictions of decidability and unambiguity of typing, type languages have never been regarded as programming languages. Consequently they lack even the most basic features one expects from a programming language. The examples given illustrated the loss of expressiveness due to a missing `if` construct.

Research hypothesis 1 *There is still a large unused and mostly unexplored potential in the* active *use of static type information by means of a fully Turing-complete type language.*

In order to investigate and validate the above research hypothesis we will design and implement a concrete system. This implies picking a specific base and meta language. We have opportunistically chosen Java as the base language. The choice for the meta language is a logic language similar to Prolog. The reason for choosing a logic meta language is stated in the following research hypothesis:

Research hypothesis 2 *A logic language offers the right kind of paradigm for describing and manipulating type-related information.*

These hypotheses clearly establish the direction which the rest of the dissertation will follow.

We start by providing arguments for hypothesis 2. We already gave some preliminary arguments to support this hypothesis. One argument is the way type theorists usually specify type systems: by means of a kind of logic inference rules. Another argument is the Gofer type system. The latter will be discussed in detail in chapter 2 which presents an informal introduction to Gofer and its type system. Gofer [Jon95] is a small experimental functional language offering a type system with type and constructor classes very similar to Haskell's type system but somewhat more experimental and more flexible. The similarity with logic inference rules is pointed out. It will become clear from the discussion in this chapter that the kind of logic rules offered by Gofer's type system have great expressive power.

Argumentation for hypothesis 1 covers most of this dissertation. Chapters 2 and 3 discuss existing "state of the art" type systems, and illustrate by means of some benchmark programming assignments how these type systems suffer a considerable loss of expressiveness due to restrictions in their type language. The benchmark programming assignments are chosen explicitly to probe the expressive power of the type language as a programming language to actively manipulate static types and make decisions about them. Due to the fact their type language is not a true programming language, existing type systems perform poorly on the benchmark programming assignments.

Chapters 4 through 7 work towards an experimental system with which we will illustrate the potential of active type manipulation in subsequent chapters. We start in chapter 4 by presenting some terminology used in the remainder of the dissertation. The most important concept defined in this chapter is the notion of *type oriented meta programming*, which is the technique we will use to achieve active manipulation of static types and type information. Meta programming is achieved by means of some meta language. According to hypothesis 2 we have chosen a to use a *logic* meta language. Chapter 5 therefore presents the conceptual framework of logic meta programming we have adopted. This is presented first in a base-language independent manner. Subsequently in chapter 6 the idea is applied to the base language Java and a concrete system, TyRuBa, with its design and implementation is presented. Chapter 7 shows that this system can effectively be used to achieve type-oriented logic meta programming for Java.

The remainder of the dissertation illustrates the potential of active manipulation of static types and type related information by means of the TyRuBa system. In chapter 8 we assess the expressiveness of the TyRuBa system by using it to implement the benchmark problems. In contrast with Gofer and existing parametric-type extensions of Java, the benchmark problems can all be solved adequately in TyRuBa. Chapter 9 presents an example on a somewhat larger scale.

The benchmark solutions from chapter 8 and the case study example from chapter 9 only barely scratch the surface of potential applications. In order to give a taste of the immense potential which still lies unharvested underneath the surface, chapter 10 shows how we used the TyRuBa system to support aspect-oriented programming [MLTK97, KLM$^+$97]. We show how an aspect weaver can be implemented as a special-purpose TyRuBa code generator, and how aspect declarations can be represented by TyRuBa logic facts. This approach allows embedding the special-purpose aspect language into TyRuBa's general purpose logic language, and thus to achieve *aspect-oriented logic meta programming*. This example illustrates that the ideas presented in this dissertation can be applied in a much broader context.

In chapter 11 we present a historic overview of the development of TyRuBa up to its current version employed throughout this dissertation. Subsequently we present some ideas about possible future research directions, and about how the system could be upgraded from an experimental system to a usable product. We also speculate about what the "type system of the future" might look like. This would be a statically typed system. Presumably, the system will offer a full-fledged declarative logic language as part of its type language. This will allow manipulating types by means of logic programs, and will provide ways to characterize types by means of arbitrary Turing-computable logic predicates.

Finally, chapter 12 summarizes and concludes the dissertation.

Chapter 2

Gofer: Constructor Classes and Higher-Order Types

2.1 Introduction

In this chapter we are going to have a close look at the functional language Gofer [Jon95]. Gofer has a "state of the art" type system that offers a sophisticated combination of universally quantified parametric polymorphism and qualified types. The result is a powerful type system with which a large variety of polymorphic coding idioms can be successfully expressed and statically typed. Qualified types [Kae88, WB89, Jon94b] are particularly interesting in the context of this dissertation. Declarations of qualified types have a very strong resemblance to logic inference rules. In a way, qualified type declarations introduce a kind of logic programming language into the static type system. However, this language remains limited and is not truly a programming language because of the strong restrictions it imposes. These restrictions are mostly related with unambiguity and decidability of typing, and with implementation-related concerns. Despite the limitations, the expressiveness of the system goes a very long way in supporting genericity. This was one of the major motivating forces behind our choosing a *logic* meta language for our system.

The languages Gofer and Haskell and their type system are highly similar. Of the two, Haskell takes the more conservative approach and imposes more limitations on the declarations of qualified types than does Gofer. From the point of view of our thesis and dissertation Gofer is therefore preferable over Haskell. An overview of the decisions involved in designing a qualified-type system and a comparison between the particular choices made by Gofer and Haskell can be found in [JJM97].

In the following sections we will informally introduce the functional language Gofer and its type system. We will show some examples that illustrate its potential for writing generic code by making use of qualified types and universally quantified type variables. We will also illustrate where the restrictions built into the type language cause complications, making things hard or impossible to express.

An interesting side note which supports our thesis is that some versions of the Gofer type system are not truly decidable [JJM97]. This however does not seem to create problems for practical Gofer programs.

The reader who is familiar with Gofer (or Haskell) and its type system may consider skipping the introductory sections and move directly to section 2.9.

2.2 A Purely Functional Language

Gofer is a statically typed lazy purely functional language. This essentially means that it is modeled after simply typed lambda calculus [Mit96c]. Some syntactic sugar is added to make it more easily usable and, more importantly, the type system is extended. Universally quantified type variables are added by adopting the Hindley-Milner type system [Hin69, DM82]. This is further extended to support overloading by means of type classes and qualified types [Kae88, WB89, Jon94a, Jon94b].

The statement that Gofer is a *purely functional* language entails that it has no assignment feature nor any other feature for performing side effects directly[1]. Usually purely functional languages have what is called lazy evaluation. This means that evaluation of expressions in the language are delayed until the latest possible moment.

We are now going to make an informal tour of Gofer. We will not try to give a complete picture of the entire language and all of its syntax and semantics. We only want to give a general idea of what the language looks like and how it is used in practice. For a more complete description we refer to [Jon95, Jon91a, Jon93b, JJM97]. Issues such as operator syntax will not be treated in detail. These and other mostly syntactic features are interesting and definitely worthwhile to make the language more usable in practice, but in the context of this dissertation we are more interested in the type system which will therefore be more thoroughly explored. Nevertheless, it is not our intent to repeat the Gofer manuals [Jon91a, Jon91b, Jon93b] and uncover every little detail about the type system. Instead, after informally introducing it by means of examples, we will look at it from the point of view of this dissertation and discuss how closely the type language resembles a programming language and explore the limitations that hamper its expressiveness when considered as a programming language.

2.3 Function Declarations and Invocations

The most important features of a functional language are function declarations and invocations. This corresponds to *abstraction* and *application* in lambda calculus. The syntax of these is nearly identical to that of lambda calculus. The following is a very simple function declaration illustrating the Gofer syntax of abstraction and application. It defines a function `compose` which accepts two functions, f and g, as arguments and returns their composition $f \circ g$.

```
compose f g = \x -> f (g x)
```

A "\" in Gofer has the same meaning as "λ" in the lambda calculus. Function application is denoted by simple juxtaposition and associates to the left. The parenthesis around `g x` in the example are therefore necessary because otherwise the body of the lambda expression would be considered equivalent to `(f g) x`.

Just as in lambda calculus, it is standard practice in Gofer to "simulate" functions with multiple arguments by means of higher order single argument functions. This technique is called currying. Even though the syntax of the `compose` declaration in the example above seemingly declares a two argument function, it in fact does not. In reality this definition is considered equivalent to the following:

[1]Side effects are simulated by passing an explicit "state" in and out of functions.

```
compose = \f -> \g -> \x -> f (g x)
```

Readers sufficiently familiar with lambda calculus or functional programming will understand this at once. However, for those not so fortunate we provide a more intuitive example to help illustrate and explain currying a little better. Consider the following rather trivial function declaration which defines a function `add` that adds its "first argument" to its "second argument".

```
add x y = x + y
```

As we just explained, this is not really a two argument function but is equivalent to:

```
add = \x -> \y -> x + y
```

It is convenient that application associates to the left since we may simply invoke `add` on numbers 3 and 4 by writing an expression "`add 3 4`". This expression will be implicitly parenthesized as "`(add 3) 4`" which has the desired effect. The function `add` gets an argument 3 for its formal parameter `x`. The returned function is equivalent to `\y -> 3 + y`. This function receives a parameter 4 and finally returns the result 7.

Function declarations may refer to themselves or to each other to define recursive or mutually recursive functions[2]. The prototypical example is of course the declaration of a function `fac` which computes $n!$ when given an argument n.

```
fac n = if n==0
          then 1
          else n * fac (n-1)
```

This example also illustrates the use of the `if then else` special form and infix operator syntax. Gofer offers a number of predefined infix operators such as `+`, `-`, `*`, etc. to do arithmetic on numbers. There are also a number of comparison operators such as `==`, `<`, `<=`, etc. which return a boolean. The infix notation for operators is merely syntactic sugar. The non-infix form for the expression `3 + 4` for example is `(+) 3 4`. Thus, the function `(+)` is not a real two argument function but a curried function. We can for example declare an increment function as follows:

```
increment = (+) 1
```

2.4 Types and Type Declarations

In this section we introduce Gofer's type system and type language. The simple examples presented above are all valid statically typed pieces of Gofer code even though we have not used a single type declaration up until now. There is no need for type declarations because the type system has an inferencer which can derive static types from the source code of the Gofer program. For providing (verified!) documentation or for enforcing types which are more restrictive than the ones the system would infer itself[3], it is possible to declare types

[2]Actually not only *function* declarations but also declarations of data structures such as lists may use recursion. This is possible (and useful) due to lazy evaluation.

[3]The inferencer will derive the "most general" type.

explicitly. The following example declares the function `add` to be of type "`Int -> Int -> Int`". What this type expression means we will come back to later. For now we merely want to illustrate the syntax of a type declaration.

```
add :: Int -> Int -> Int
add x y = x + y
```

2.5 Simply Typed Gofer

We will now make a tour of Gofer's type language, illustrating and explaining the different type expressions it offers to specify function and data types. We will introduce these in a gradual manner. This section will treat only types which conform more or less to the simply typed lambda calculus [Mit96c]. This means there are no polymorphic types. Section 2.6 illustrates what happens when universally quantified type variables are introduced to support polymorphism (Hindley-Milner type system [Hin69, DM82]). Finally, in section 2.7, we turn our attention to the more experimental extension of the type system which offers type classes and qualified types to support yet another kind of polymorphism through "overloading" [Kae88, WB89, Jon94a, Jon94b].

Restricting ourselves to non-polymorphic types and always providing explicit type declarations for everything, Gofer becomes practically identical to the simply typed lambda calculus. Type expressions in the type language of "simply typed" Gofer can be divided into two groups. One group of primitive types and one group of derived types. Primitive types are like constants in the type language whereas derived types are more like the result of applying a "type function", in Gofer jargon called a *type constructor*, to a number of argument types. Every type constructor has a certain arity[4] indicating how many parameters the "type function" needs. Type and constructor names are distinguished from names for normal functions and values because they start with a capital letter.

Primitive Types

There are a number of primitive types in Gofer. These are summarized in the table in figure 2.1. There is one rather strange primitive type denoted `()` in the table. This is called the *unit* type and corresponds exactly to the type *unit* in simply typed lambda calculus. There is exactly one value of this type, also denoted `()` and called *unit*.

Type	Some values of the type
`Int`	`0`, `-1`, `1`, `-2`, `2`, `-3`, ...
`Float`	`3.14`, `-1.1`, `5.2e-2`, ...
`Char`	`'a'`, `'1`, `'X'`, `'*'`, ...
`Bool`	`True`, `False`
`()`	`()`

Figure 2.1: Primitive types in Gofer

If we have only primitive types the only thing we can declare types for are constants[5]. For example we can declare a constant of type `Float`.

[4]Primitive types can be regarded as type constructors of zero arity.

[5]Strictly speaking, every globally declared name in Gofer is a constant since Gofer is purely functional.

```
pi :: Float
pi = 3.14159
```

Function Types

Things only start getting interesting when we have function types. The constructor for function types is `->`. It has arity 2 and is used as an infix operator. To accommodate expressing types of curried functions it associates to the right. Some examples:

```
fac :: Int -> Int
fac n = if n==0
          then 1
          else n * fac (n-1)

add :: Float -> Float -> Float
add x y = x+y

compose :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)
compose f g = \x -> f (g x)
```

Note that the type of `add` could just as well have been `Int -> Int -> Int`. It would seem strange that this function could be typed correctly in two distinctive ways. We will discuss later in section 2.7 how this kind of polymorphism can be assigned a single (polymorphic) qualified type. The type of `compose` could also be more general. There is no reason really to restrict its argument to functions on integers. The compose method too can be assigned a polymorphic type, but this time using universally quantified type variables as will be explained in section 2.6. Many of the examples that still follow in this section would also be best described by a polymorphic type. However, we will defer giving polymorphic types until section 2.6 and section 2.7.

Tuples

Tuples in Gofer correspond to product types in simply typed lambda calculus. The simplest tuple is a pair[6]. The value `(1,'i')` for example is a pair of type `(Int,Char)`. In general there is an n-ary tuple type constructor for every $n \in \{2, 3, 4, \ldots\}$. Some more example tuple types and values can be found in figure 2.2. Note that the examples `(Int,Int,Int)` and `(Int,(Int,Int))` denote distinct types: one is a triplet and the other a pair containing another pair.

Gofer provides predefined polymorphic accessor functions for pairs which correspond to the projection functions in simply typed lambda calculus. These are rarely used because Gofer allows a kind of pattern matching syntax on arguments in functions to extract component values from tuple values. An excellent example of this is the definition[7] of the accessor functions themselves.

However, the word "constant" is usually used to only signify something which is *not* a function constant. Otherwise it is simply called a "function" instead of a "function constant"

[6]The type `()` could be considered a tuple of arity zero. This would then of course be an even simpler type of tuple. Note that tuples of arity 1 do not exist. The expression `(1)` for example is not a tuple but is simply a parenthesized expression and semantically equivalent to `1`

[7]Found in the "`standard.prelude`", an initialization file that contains definitions for predefined functions.

Type	Some values of the type
`(Int,Char)`	`(0,'a')`, `(-1,'<')`, `(55,'F')`, ...
`(Int,Int,Int)`	`(0,0,0)` ,`(0,1,0)`, `(3,4,5)`, ...
`(Int,(Int,Int))`	`(0,(0,0))` ,`(0,(1,0))`, `(3,(4,5))`, ...

Figure 2.2: Some example tuple types and values

```
fst (x,y) = x
snd (x,y) = y
```

The formal parameter `(x,y)` is "matched" to the actual argument when the function is called, binding variable `x` to the first component of the pair and `y` to the second.

The above functions can be used on all pairs and therefore have a polymorphic type. We will come back to the (polymorphic) types of these examples later.

Lists

Complementary to tuples, which are fixed arity data structures, lists are variable arity data structures. A list may hold any number of elements of a given type. Figure 2.3 gives some example list types and values. Note that strings are considered to be lists of characters in Gofer. The type `String` is actually just a synonym for `[Char]`.

Type	Some values of the type
`[Int]`	`[]`, `[4]`, `[4,5,-1]`, ...
`[Char]`	`[]`, `['a','b']`, `"BarFoo"`, ...
`[[Int]]`	`[]`, `[[],[]]`, `[[1,2],[3],[4,5]]`, ...

Figure 2.3: Some example list types and values

Lists in Gofer can be conveniently created in a number of different ways. We will not go into detail about this. We only briefly discuss the most basic of them and leave special "syntactic sugar" notations aside. The example values in figure 2.3 can be used exactly in that form in Gofer programs as literals denoting list constants. The literal `[]` denotes an empty list. This is a polymorphic constant since it can be any type of list. Non-empty lists can be constructed with the predefined ":" operator. This is an infix operator which associates to the right to make constructing lists syntactically simpler. The list `[1,2,3]` for example may be constructed as follows:

```
oneTwoThree = 1:2:3:[]
```

Gofer provides some predefined accessor functions to access elements in a list. The most basic of these are `head` and `tail`. The function `head` returns the first element of a non-empty list whereas the function `tail` returns a list of all elements but the first. There is also a convenient pattern-matching syntax for lists illustrated below by the standard definitions for the functions `head` and `tail`.

```
head (x:xs) = x
tail (x:xs) = xs
```

Another, more complicated example, also using pattern matching syntax on lists is a function which computes the sum of a list of integers.

```
sumInt :: [Int] -> Int
sumInt [] = 0
sumInt (x:xs) = x+sumInt xs
```

Note that there are seemingly two declarations for this function. However this is merely syntactic sugar. When the function is called, the different declarations are tried in the order in which they are declared. The first one that matches the actual argument is chosen. The pattern `[]` only matches an empty list whereas the pattern `(x:xs)` matches a non empty list and binds `x` and `xs` to the head and the tail of the list respectively. The same append function could thus be expressed more verbosely without pattern matching and a single declaration as follows:

```
sumInt :: [Int] -> Int
sumInt xs = if null xs
            then 0
            else head xs + sumInt (tail xs)
```

"Data" types

The most general of type declarations are `data` declarations. They offer a combination of sum types (i.e. "disjoint unions" of types) and product types. In fact, all of the types listed up until now could in fact be interpreted as syntactic sugar for types declared by a `data` declaration.

The following is an example `data` declaration for a type of binary trees of integers, called `TreeInt`.

```
data TreeInt = Leaf Int
             | Node Int TreeInt TreeInt
             | Empty
```

The "`|`" should be read as a union of "tagged types". The symbols `Leaf`, `Empty` and `Node` are tags which will be part of the runtime value of `TreeInt` to identify whether it is a leaf node containing an `Int`, an internal node containing an `Int` with two subtrees, or simply an `Empty` tree.

The above not only declares a new type, `TreeInt`, but at the same time it also implicitly declares constructors for building trees. These constructors have the following types.

```
Leaf  :: Int -> TreeInt
Node  :: Int -> TreeInt -> TreeInt -> TreeInt
Empty :: TreeInt
```

The constructor function `Leaf` takes an integer and turns it into a `TreeInt` by adding to it a `Leaf` tag. Similarly `Node` constructs an internal node when given the appropriate arguments. The `Empty` "constructor" does not need any arguments so it is a constant instead of a function.

Pattern matching syntax may be used on `data` types as well. The following example is a function that computes the product of all integers in a `TreeInt` using pattern matching.

```
prod :: TreeInt -> Int
prod Empty = 1
prod (Node i n1 n2) = i*prod n1*prod n2
prod (Leaf i) = i
```

2.6 Parametric Polymorphism

With what was discussed above, already a great deal of useful programs may be written in Gofer. However it is not possible to give a type for the function `compose` that does not restrict it too much. Nor will it allow us to declare types for generic functions on lists or other data structures such as pairs. Nor will it allow us to declare generic data structures of our own. In the following sections we will discuss what provisions Gofer's type system has for supporting genericity.

Gofer provides several mechanisms for supporting polymorphic types. Mostly we divide these into two categories. On the one hand there are provisions for parametric types: parametric `data` declarations, type synonyms and universally quantified type variables. These will be treated in this section. On the other hand we have type or constructor classes and qualified types that support overloading, which we we will discuss in section 2.7.

We will introduce both of these informally and gradually. Each time we introduce another feature of the type language this will be accompanied by a motivating example that illustrates what the feature is needed for. This gradual manner in which the type system is presented reflects nicely how an increasingly more complex type language allows better static "estimates" of runtime values to be expressed and thus results in more flexibility for the programmer. It is also noticeable that as the type language grows more complex it starts to look more and more like a programming language.

2.6.1 Type Variables

Up until now we have considered type expressions in Gofer's type language to consist of type constants and of type constructors applied to a number of argument type expressions. Type constants we have seen were either primitive types, such as `Int` and `Float`, or declared by `data` such as the `TreeInt` example. We have seen predefined built-in constructors for creating function types, tuple types and list types. A constructor for *sum* types which can also be found in simply typed lambda calculus is not explicitly present as a type constructor in Gofer, but it is implicitly present under the form of the "`|`" in `data` declarations.

We now introduce type variables, yet another kind of type expression in Gofer. A type variable is any identifier starting with a lower case letter, occurring in a type expression. The lower case letter distinguishes it from constructors[8] and type constants that start with upper case letters.

Type variables are universally quantified which means essentially that they function as "wild cards" which may be substituted by any type. The first example usually presented in many a textbook to illustrate the use of universally quantified type variables is the identity function.

[8]Not counting type constants which are in fact type constructor of arity 0, we have not seen any type constructors that start with upper case letters yet. We will soon see how these may be defined by means of `data` declarations.

```
id :: a -> a
id x = x
```

The type variable `a` functions as a wild card. It may be replaced by any type. However, if one replaces one occurrence of the variable then the other one must also be replaced by the same type. The function `id` may thus be regarded as being of type `Int -> Int` or `Bool -> Bool`, but not of type `Int -> Char` for example.

Another good example is the `compose` function which composes two functions by applying the first one to the result returned by the second. The type of this function could be declared:

```
compose :: (a -> b) -> (c -> a) -> (c -> b)
```

These and some other examples from section 2.5 whose type can be expressed with universally quantified type variables are listed in figure 2.4.

Value	Type
`[]`	`[a]`
`fst`	`(a,b) -> a`
`snd`	`(a,b) -> b`
`head`	`[a] -> a`
`tail`	`[a] -> [a]`
`id`	`a -> a`
`compose`	`(a -> b) -> (c -> a) -> (c -> b)`

Figure 2.4: Some example polymorphic types with type variables

Universally quantified polymorphism is typically needed for polymorphic functions on generic data structures, such as tuples and lists, which store elements without performing any operations on the stored elements themselves. Since the elements are not operated on in any way they can be of *any* type, and a universally quantified type variable is the perfect fit for them. We present one more example of this, the function `length` which computes the length of a list.

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1+length xs
```

This function is polymorphic and works on lists containing *any* type of elements. It can for example be used both on a list of integers as well as a list of characters.

2.6.2 Parametric Data Structures

With type variables it is possible to conveniently express the types of polymorphic functions on generic lists and tuples. But what about user-defined data structures such as the `TreeInt` example? It is very natural that we would want to define generic functions on trees also. For example a function `fringe` which traverses the tree and returns a list of the elements in its leafs. Let us first express it with the non-polymorphic type `TreeInt`.

```
fringe :: TreeInt -> [Int]
fringe Empty = []
fringe (Leaf i) = [i]
fringe (Node i l r) = fringe l++fringe r
```

Note, the operator `++` computes the concatenation of two lists, and has polymorphic type:

```
(++) :: [a] -> [a] -> [a]
```

Except for the type declaration, we could just as well use the exact same code on a tree of characters or any other type of elements. What we need is a way to declare a generic tree data structure as a type constructor, `Tree`, of one argument so that we may write `Tree Int` for the type of a tree of integers or `Tree Char` for the type of a tree of characters. This can be done easily in Gofer. A `data` declaration in fact allows not only declaring new type constants, but also new type constructors of any arity. For the `Tree` example we can write:

```
data Tree a = Leaf a
            | Node a (Tree a) (Tree a)
            | Empty
```

This declares `Tree` as a type constructor of arity 1. Thus, `Tree` is like a function on types. It takes a type `a` as argument and constructs the type `Tree a` much in the same way the type constructor `[]` constructs a type `[a]` from a type `a`. We can thus conveniently declare the polymorphic type of `fringe`:

```
fringe :: Tree a -> [a]
```

This will allow `fringe` to be used on a tree with any type of elements.

2.6.3 Type Synonyms

Parametric data declarations provide a kind of "type abstraction". They can declare a kind of functions on types. Type synonyms are another way of creating type abstractions. A type synonym is basically a kind of "type macro". It may have parameters which are types and expands to a type. As an example of its use consider the following implementation of a Stack ADT, represented by a list of elements.

```
type Stack a = [a] in emptyStack, isEmptyStack, push, pop, top

emptyStack :: Stack a
emptyStack = []

isEmptyStack :: Stack a -> Bool
isEmptyStack []  = True
isEmptyStack s   = False

push :: Stack a -> a -> Stack a
push s a = a:s
```

```
top :: Stack a -> a
top s = head s

pop :: Stack a -> Stack a
pop s = tail s
```

The type declaration above declares the type `Stack a` to be a synonym which "expands" to `[a]`. To hide the abstract data type's implementation details from its users, one may list names of functions or constants in which the expansion is valid. For all other declarations the type synonym will be treated as a "black box" so that the implementation details are hidden. In the example above this means that the declarations of `emptyStack`, `isEmptyStack`, `push`, `top` and `pop` can treat the stack as a list of elements. Outside of these declarations, list operations such as `head` and `tail` etc. will cause a static type error when invoked on something of type `Stack a`.

2.7 Overloading: Type and Constructor Classes

The most interesting part of Gofer's type system is its mechanism for overloaded functions. We have already encountered a simple example of a polymorphic function which cannot be expressed with universally quantified variables. The example was a simple `add` function:

```
add x y = x + y
```

The examples found in figure 2.4 are mostly the polymorphic examples from the preceding sections for which we either did not declare a static type or for which we declared a type which was more restrictive than needed. Noticeably missing from this list is the type of the function `add` which was also declared with too restrictive a type `Int -> Int -> Int` (and also `Float -> Float -> Float` at another point). We might think the type of this function should be `a -> a -> a` to allow it to be applied to both integers and floats. This type however is too general since we would then be allowing it to be applied to *any* type, whereas we should only allow it to be applied to a type that has a `+` operation defined on it. The type of `add` needs to be expressed with something more restrictive than universally quantified type variables. This is the motivation behind the introduction of type classes and qualified types which are the topics of this section. Historically, in the beginning Gofer only had type classes [Jon91a]. Later it was found that the expressiveness of type classes was too restrictive for some very useful examples such as monads [JD93] and modular interpreters [LHJ95, Ste94a]. This gave an impulse to extend the type system with constructor classes [Jon93a, Jon93b], a "higher-order" generalization of type classes. Again we will introduce these features gradually, starting with type classes and presenting more complicated examples that illustrate the need for constructor classes.

2.7.1 Type Classes

The function `add` from above is polymorphic because it can work with more than one type of arguments. The following are both correct static types for the `add` function and they will both be accepted by the Gofer type checker.

```
add :: Int -> Int -> Int
add :: Float -> Float -> Float
```

The question now is, how can we assign a polymorphic type to `add` that comprises both of the above. We might try it with a universally quantified type variable as follows:

```
add :: a -> a -> a
add x y = x + y
```

However, this type is too general as previously pointed out. By assigning this type we are actually declaring the function to be applicable to any type. What we need is a way to "constrain" the type variable `a` to range not over all types, but only over types that can be added with an operator "`+`". In Gofer the range over which a type variable varies may be constrained by means of one or more type *predicates*. For example, the type that is inferred by the Gofer system for `add` is:

```
add :: Num a => a -> a -> a
```

This type is called a *qualified* type. In a qualified type, type variables may be constrained by one or more predicates. In this example there is one such predicate: `Num a`. This predicate states that the type `a` should be an *instance* of the *type class* `Num`. The instances of a type class are types which all share some common functionality. In this case, the types in the class `Num` are `Int` and `Float` and they share, amongst other things, that they have an operator `+` defined on them[9].

Note that the Gofer terminology "type class" and "instance" should not be confused with object-oriented terminology "class" and "instance". If we want to name something in object-oriented languages which corresponds to a type class in Gofer, the thing which probably resembles it most is an interface (as in Java) or an abstract class. An instance of a type class then corresponds to a concrete Java class that implements the interface or concretizes the abstract class.

We now present an example which is typically found in texts that argue the need for bounded parametric polymorphism in object-oriented languages such as Pizza. The example we present is an implementation of a priority queue and it corresponds roughly with the priority queue example from [AFM97]. A priority queue is the prototypical example of a data structure which stores elements, but unlike lists or tuples, requires some functionality from these elements. The implementation of a priority queue for example, is dependent of the fact that there is a way of computing priorities for its elements. We therefore declare a type class `Priority`. Every instance of this class must have a function to compute a priority.

```
class Priority a where
  priority :: a -> Int
```

Of course this type class does not have instances yet, we still have to declare these. The following trivial instance declaration declares the identity function as the function to compute a priority for integers.

[9]The declaration of the type class `Num`, and instance.declarations for `Int` and `Float` are part of the `standard.prelude`.

```
instance Priority Int where
  priority = id
```

Now let us consider the code for the priority queue itself which is given below.

```
type PQueue a = [a] in
  emptyPQ, isEmptyPQ, enqueuePQ, nextElem, dequeuePQ

emptyPQ      :: Priority a => PQueue a
emptyPQ = []

isEmptyPQ    :: Priority a => PQueue a -> Bool
isEmptyPQ []  = True
isEmptyPQ any = False

enqueuePQ    :: Priority a => PQueue a -> a -> PQueue a
enqueuePQ [] e = [e]
enqueuePQ (x:xs) e = if (priority x<priority e)
    then x:enqueuePQ xs e
    else e:x:xs

nextElem     :: Priority a => PQueue a -> a
nextElem (x:xs) = x

dequeuePQ    :: Priority a => PQueue a -> PQueue a
dequeuePQ (x:xs) = xs
```

This implementation of a priority queue as a list of elements sorted according to priority is pretty straightforward and does not need a lot of explanation. Note that all of the type declarations are qualified types that restrict the type of the elements in the queue. Strictly speaking we did not have to use a qualified type in every type declaration. Only for `enqueuePQ` is this really necessary because it is the only one that invokes the `priority` function. However, a priority queue in which no elements can be inserted would not be of much use anyway. Therefore we thought it best to also impose the constraint on all of the other declarations as well. Instead of putting the qualification `Priority a =>` with each type declaration we would have preferred to declare a type synonym:

```
type PQueue a = Priority a => [a]
```

But, regrettably, qualified types are not allowed in type synonyms.

We give one more example that illustrates the similarity between qualified type declarations and rules in a logic program. The example we have chosen is the class `Eq` from the standard prelude. The `Eq` class is declared as follows:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
```

This declaration says that any type which is declared an instance of the class `Eq` must have the binary operators "`==`" and "`/=`" defined on it. These represent a test for equality and inequality respectively. The class declaration also provides a default implementation for "`/=`". This means that an instance declaration is only obliged to give an implementation for "`==`" while the implementation "`/=`" may be omitted. The prelude also provides a number of standard instance declarations. For example, the primitive types `Char`, `Bool`, `Float` and `Int` are instances of `Eq`. This is the instance declaration for `Eq Int` for example:

```
instance Eq Int  where (==) = primEqInt
```

This declaration links the "`==`" operator for integers to a primitive equality function `primEqInt`. Declarations for `Eq Float` etc. are similar.

There are also some declarations for `Eq` instances of compound types such as lists and tuples. These are more interesting because they resemble rules from a logic programming language. Below is the instance declaration for pairs.

```
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (u,v)  =  x==u && y==v
```

The above can be read as: "If a type `a` is an instance of `Eq` *and* type `b` is an instance of `Eq` *then* the tuple type `(a,b)` is also an instance of `Eq`". Note that the parentheses around the predicates `Eq a` and `Eq b` have nothing to do with the fact that we are talking about pairs. This is merely part of the syntax to express more than one predicate in the condition of a qualified type. As can be seen, there is indeed a striking similarity between rules in a logic language such as Prolog and instance declarations in Gofer. There is a syntactic difference of course: In Gofer the *condition* precedes the *conclusion* whereas in Prolog the *conclusion* precedes the *condition.* There is also an important restriction on the form of "rules" in instance and class declarations that prohibits rules which have one or more variables to the left of `=>` that do not also occur to the right of it. This makes the expressiveness of rules in Gofer's type system considerably weaker than those of a logic programming language such as Prolog.

2.7.2 Constructor Classes

Originally Haskell was very restrictive in the way type classes could be used. It allowed only single-parameter type classes for example. Gofer from the start was much more liberal in this respect and imposed far less restrictions on the form of type and instance declarations.

Gofer for example does allow multi-parameter type classes. An example of their use is the declaration of a type class `Collects c e` which signifies that the type `c` can be used to collect elements of type `e`. As a first attempt we declare the following:

```
class Collects c e where
   emptyCollection :: c
   addToCollection :: c -> e -> c
   toList          :: c -> [e]
```

This declaration captures the most essential features of a collection. First, there must at least be an `emptyCollection` with no elements in it. Further, since collection store elements we must be able to add elements to it by means of a function, `addToCollection`. Finally, a collection data structure is not of much use if there is not some way of accessing the elements that are stored in it. This is what the function `toList` is for, it returns a list of the stored elements. There is a slight problem with this class declaration however. Gofer's type system does not allow it, and reports an "ambiguous type" error for `emptyCollection`. The types of declarations inside a type class implicitly get a qualification which states that they are members of the type class. The full type of `emptyCollection` therefore is:

```
emptyCollection :: Collects c e => c
```

Gofer does not "like" this type because the type `e` is "ambiguous". What this means exactly is that because the type variable `e` does not occur to the right of `=>`, the type inferencer cannot infer the type of `e` when it encounters an `emptyCollection` in a program. Since type classes supposedly can have multiple instances there could be several possibilities for `e`. This problem can be avoided (for now) by splitting up the class `Collects` as follows:

```
class EmptyCollection c where
   emptyCollection :: c

class EmptyCollection c => Collects c e where
   addToCollection :: c -> e -> c
   toList          :: c -> [e]
```

Notice the qualification "`EmptyCollection c =>`" in this class declaration. The meaning of this qualification is that `EmptyCollection c` is a "super class" of `Collects c e` in the sense that every declared instance of `Collects c e` should also be a declared instance of the super class. The ambiguity problem disappears because the type of `emptyCollection` now becomes

```
emptyCollection :: EmptyCollection c => c
```

We can even declare instances of the class. For example, lists can obviously be regarded as collections. Here are the necessary instance declarations:

```
instance EmptyCollection [a] where
   emptyCollection = []

instance Collects [a] a where
   addToCollection c e = e:c
   toList = id
```

The problem seems to be solved. However, it resurfaces with the following simple function declaration that is meant to count the number of elements in a collection:

```
sizeof c = length (toList c)
```

The inferred type for `sizeof` is the following:

```
sizeof :: Collects a b => a -> Int
```

For some reason the Gofer[10] type system does not complain about this inferred type even though it is clearly ambiguous since the variable `b` does not occur in the type `a -> Int`. The ambiguity is however quickly detected when we try to invoke the function, for example on a list of integers.

```
? sizeof [1,2,3]

ERROR: Unresolved overloading
*** type        : Collects [Int] a => Int
*** translation : sizeof [1,2,3]
```

The problem is that the type inferencer cannot determine that `[Int]` is actually a collection of `Int` and not of some other type of value. That is exactly what the error message tells us by putting a type variable `a` in place of the element type. It might seem strange that the type system does not determine the type `a` to be `Int` here. But in fact this not so astonishing if we consider that it is possible to declare multiple instance of the class `Collects`. With a little bit of creativity we can consider an `[Int]` to represent a collection of characters represented by their ASCII code for example.

```
instance Collects [Int] Char where
   addToCollection c e = ord e:c
   toList c = map chr c
```

With the above instance declaration there are now two possible instances for `Collects [Int] a`. The first one is the most straightforward one we already had: `Collects [Int] Int`. The other one is `Collects [Int] Char`. This example clarifies why `sizeof [1,2,3]` results in an error. Basically the error means that the type system has no way of knowing whether `[Int]` should be interpreted as a collection of characters or as a collection of integers. As a consequence it cannot resolve the overloading of the `toList` function which is invoked in the implementation of `sizeof`. In other words, the type system cannot determine whether to use `toList::[Int] -> [Int]` or `toList::[Int] -> [Char]`. Note that even when there is no "real" ambiguity, i.e. when in reality only one instance declaration "fits the shoe", the type system still reports an error because it does not take effective instances into account to decide whether something is ambiguous or not. Simply the fact that the variable `a` is constrained by a type predicate implies potential ambiguity when new instance declarations are added in the future, for example by including other modules. This is sufficient reason for Gofer to report an error. We feel this is a rather strong restriction and it often leads to unexpected ambiguous-type related errors as illustrated by the example. The restriction is nevertheless adopted in Gofer because the possibility for separate compilation was an important concern in its design [JJM97].

Despite the example that declares that an `[Int]` can be regarded as a representation for a collection of characters, it is not entirely unreasonable to assume that a representation for a collection should only be regarded as a collection of a particular type of elements. Indeed, considering a list of integers to represent a collection of characters is a bit contrived. We merely presented this example to explain why exactly the Gofer system reports an ambiguous type error and that there is indeed a potential for ambiguity by adding further instance

[10]We used Gofer 2.30a to try these examples.

declarations. When programming in Gofer we have often felt frustrated at getting unexpected ambiguous-type related errors while knowing with absolute certainty that in reality there was no ambiguity because there was only one applicable instance. Often this was not coincidental but was caused by an intuition of the kind illustrated in the example that a collection holds a *certain* type of value. Real programs therefore probably would not declare instances that cause "real" ambiguity in the type of `sizeof`. It would therefore seem useful to relax the strong "ambiguous-type" restriction such that they are only considered as errors when the ambiguity is real and there are actually several applicable declared instances. Gofer however provides a different solution to the problem which allows it to retain its meticulous ambiguity checking. Gofer's solution is the notion of *constructor classes*. Constructor classes are a generalization of type classes. The instances of a type class are types, whereas the instances of a constructor class are type constructors. Types are considered to be constructors with zero arguments and therefore type classes are just a special case of constructor classes. Let us see how constructor classes solve the problem. Below is a declaration of a constructor class, `Collection`.

```
class Collection c where
  emptyCollection :: c a
  addToCollection :: c a -> a -> c a
  toList :: c a -> [a]
```

The following is an instance declaration which declares the list constructor `[]` which constructs the list type `[a]` from a type `a`.

```
instance Collection [] where
  emptyCollection :: [a]
  emptyCollection = []

  addToCollection :: [a] -> a -> [a]
  addToCollection c e = e:c

  toList :: [a] -> [a]
  toList = id
```

The declaration of the `sizeof` function remains the same except for the type qualification.

```
sizeof :: Collection c => c a -> Int
sizeof c = length (toList c)
```

In this, the variable `c` does not refer to a type, but to a constructor which accepts a type as argument and constructs a new type from it. So if `c` is a "collection constructor" and `a` is a type then `c a`, the application of `c` on `a`, is a "collection of elements of type `a`". Because the relationship between a collection type and the type of its elements is now expressed by means of a constructor, a *function* from types to types, the ambiguity problem disappears. The reason for the problems with the `Collects` *type* class is that it is expresses a many-to-many relationship between a collection type and an element type. The constructor class on the other hand expresses a many-to-one relationship between collection types, `c a`, and element types, `a`. The problems with `Collect` were caused by the fact that there were potentially many element types for a single collection type. These problems therefore disappear when the many-to-many relationship is replaced by a many-to-one relationship.

2.8 The Kind System

It should be realized that the introduction of constructor classes as a generalization of type classes makes the type system considerably more complex. Without constructor classes, there was only a need for one kind of type variables that can be bound to a type. Now, there are also type variables which will be bound to constructors rather than types. Therefore a need arises for some sort of "type checking" for type variables and constructor expressions. To avoid confusion between types of functions and "types" of constructors, the latter are usually referred to as *kinds*. The kind `*` is the kind of all types. A constructor such as `[]` or `Tree` takes a type as argument and returns a type so therefore it has kind `*->*`. The pair constructor `(,)` has kind `*->*->*`, it is a (curried) "type function" taking two types as arguments and returning another type. Figure 2.5 lists examples of kinds and some type expressions of that kind.

Kind	Constructors
`*`	`Int`, `Bool`, `(Int,Bool)`, `Tree Int`
`*->*`	`[]`, `Tree`
`*->*->*`	`(,)`
`*->*->*->*`	`(,,)`

Figure 2.5: Some examples of kinds and constructors of that kind

2.9 The Expressiveness of Gofer's Type Language

In this section we finally come to the most important and interesting part of the chapter. We are going to have a close look at Gofer's type language from the perspective of a programming language. As we will see, Gofer's type language is remarkably expressive. Nevertheless, the type language has some very important restrictions built into it that hamper its expressiveness. As a result, even though it is possible to express a lot of things by means of Gofer type programs, it is not always clear how to go about this, and where exactly the boundary between what can and what cannot be expressed is situated. Often one hits this wall rather unexpectedly. To illustrate this, we will give some examples that try to use the type language as a programming language and point out the problems we encountered.

Gofer's type language has several features that make it look like a programming language. More specifically, it has features of both functional and logic programming languages. The most basic features of a functional programming language are lambda abstraction and application. Gofer's type language has features resembling both of these. Type synonyms and `data` type declarations are like lambda abstraction for types since they define constructors. Since the introduction of the kind system Gofer even allows declaring higher-order type constructors, that take other type constructors as arguments. The kind system plays the role of a "meta type system" for statically typing type abstractions and applications. Gofer's type language also has features which are highly reminiscent of a logic programming language. Qualified types and class and instance declarations look very much like logic Horn clauses.

We will now explore the combined power of the logic and functional features of the type language. The way we will do this is by means of some examples, "benchmarks", that probe the expressiveness of the type language as a programming language. These benchmarks fall into two categories. The first category assesses how well the type programming language can express decision making (`if then else`). The second category determines how well it can handle iteration or recursion. Both of these are essential and basic features of any programming language. The next two subsections present the benchmark programming assignments and our attempts to express them in Gofer.

2.9.1 Decision Making Benchmarks

An essential feature of any programming language is that it allows decision making. Depending on a condition, a program may take different paths of execution or return a different result. This does not necessarily mean that the language must have an explicit `if-then-else` construct. Logic languages for example support decision making implicitly through pattern matching and unification and do not need an explicit `if-then-else`. We have seen some examples that illustrate the lack of a good decision making feature, leading to serious loss of expressiveness in typical parametric typing extensions of Java, in section 1.6. We recycle these examples to probe the expressiveness of Gofer's type language.

Conditional Implementation

The first example in section 1.6.1 was an array of elements and an implementation of an interface `Searchable` on this data structure. The implementation of the searchable interface entailed calling a method that compares elements for equality. Therefore it was only valid for arrays that contain elements which support that method. As a consequence of the lack of a decision making feature in the type language, arrays could only be created with elements that support equality tests. What we wanted was to allow arrays to be created with elements that do not support an equality test. In this case, the implementation of `Searchable` would be invalid and should not be included in the array. Gofer has no arrays[11] so we have adapted this example a little bit and have taken a `List` data structure. The spirit of the example remains unchanged however.

Benchmark 1 *Implement a function* `contains` *on a* `List` *(or* `Array`*) data structure that searches the list for the presence of an element by means of an equality test on elements. If the elements in the list do not support the equality test simply omit the function or method declaration but do not prohibit the instantiation or use of the data structure altogether.*

It turns out that in Gofer's type language it is possible to express this example very elegantly. Simply define the `contains` function as an overloaded function, a member of a constructor class `Searchable` as follows:

```
class Searchable c e where
  contains :: c e -> e -> Bool
```

Note that we used a constructor class rather than a simple type class to express a many-to-one relationship between the collection type `c e` and the element type `e`. Simply using a type class, we would soon run into the kind of problems already illustrated in section 2.7.2 where we explained why constructor classes were introduced in Gofer's type language. Note furthermore that we used a two-parameter class. We need the seemingly redundant parameter `e` to be

[11]Arrays are typically mutable data structures and Gofer is purely functional.

able to constrain it later, when declaring instances of the class. For example, we declare the list constructor `[]` an instance of the class, but only if the elements in the list support an equality test.

```
instance Eq e => Searchable [] e where
  contains [] e = False
  contains (x:xs) e = (x == e) || contains xs e
```

This accomplishes precisely what benchmark 1 asked for. One may still use lists the same way as before. Additionally, lists of elements that support equality have become instances of `Searchable` so we may invoke `contains` on them.

Positioning Abstract Code

The second motivating example from the introduction roughly illustrates the same problem but is somewhat more difficult. Instead of implementing the `contains` method on a specific data structure, it is implemented on top of some abstract functionality to enumerate elements. Therefore it is a somewhat bigger challenge for the expressiveness of the type language. The example in section 1.6 illustrated that in most typical parametric type extensions for Java abstract code must be placed in abstract classes which ties the code too rigidly to a certain position in the class tree structure.

Benchmark 2 *Implement a function* `contains` *on all data structures which provide a way of enumerating their elements. If the elements do not support equality tests, then omit the declaration of* `contains` *but do not prohibit instantiation or use of the data structure altogether.*

This example can also be naturally expressed in Gofer by means of qualified type declarations. Suppose we offer the possibility to "enumerate" elements by means of a function `elements`. This function returns a list of the elements in the collection. To be able to define this function on a variety of different data structures we declare it as an overloaded function, i.e. a member of a type class.

```
class Enumerable c e where
  elements :: c e -> [e]
```

The abstract implementation of the function `contains` for all `Enumerable` types can now be provided for by means of a qualified instance declaration:

```
instance (Enumerable c a,Eq a,Searchable [] a) => Searchable c a where
  contains xs x = contains (elements xs) x
```

This declaration first enumerates the elements of the collection in a list. Subsequently it simply calls `contains` on the list. This will work because we already provided an instance declaration which establishes that lists are searchable in the solution for benchmark 1. Note that we had to include the predicate `Searchable [] a` in the qualification of the declaration because Gofer will not infer[12] it from `Eq a`, even though it is implied by the instance declaration that implements `contains` on lists.

[12]This is connected with the concern for separate compilation.

Every type constructor which is an instance of `Enumerable` now also automatically is an instance of `Searchable` whenever its elements are instances of `Eq`. As an example let us declare a new "home-made" type of list, by means of a `data` declaration.

```
data List a = Cons a (List a)
            | Empty
```

Next we implement a way to enumerate this list by declaring it an instance of `Enumerable`.

```
instance Enumerable List a where
  elements Empty = []
  elements (Cons x xs) = x:elements xs
```

A `List` of elements that support equality will now be an instance of `Searchable` so we can invoke `contains` on it.

```
? contains (Cons 1 (Cons 2 (Cons 3 Empty))) 2
True
(9 reductions, 27 cells)
```

Type-Dependent Internal Representation

This is the last benchmark that probes the power for decision making. It involves a "type function" that creates an abstract data type and makes a decision about its internal representation on the basis of type information. It is similar in nature to the dictionary example briefly mentioned in section 1.6. The example we have chosen here is the implementation of an abstract data type to represent mathematical sets of elements.

Benchmark 3 *Implement an abstract data type* `Set` *representing a set of elements of some type* `a`. *When implementing a set of element types that merely support equality tests use a simple list data structure which is searched sequentially. For the implementation of a set of ordered elements, comparable by means of* `<`, *use a binary search tree.*

This task appears to be just out of reach for the expressive power of Gofer's type language. The benchmark problem can almost be expressed. The partial solution is shown in figure 2.6. Let us take a closer look and explain why it falls short of a complete solution for the benchmark problem.

Since we want to have two different data structures as representations for sets, we declare `Set` to be a constructor class. That is the only way we can implement the same functionality on two different data structures. Notice that we again used a multi-parameter class to be able to constrain the type of elements in the set when giving instance declarations. There are two instance declarations, one for element types that are instances of `Eq`, that support an equality test, and another for instances of `Ord` that can be compared with each other by means of `<`. This seems a good solution. We can write polymorphic code that can handle both kinds of sets, without knowing what kind of set we are actually dealing with. For example we can define the following `singleton` function that takes some type of element and creates a singleton set with it:

```
class Set s a where
  empty :: s a
  add :: a -> s a -> s a
  contains :: s a -> a -> Bool

data Tree a = Leaf a
            | Node a (Tree a) (Tree a)
            | Empty

instance Ord a => Set Tree a where
  empty = Empty
  add e Empty = Leaf e
  add e t@(Node x l r)
     | e == x = t
     | e < x  = Node x (add e l) r
     | e > x  = Node x l (add e r)
  contains Empty e = False
  contains (Node x l r) e
     | e == x = True
     | e < x  = contains l e
     | e > x  = contains r e

instance Eq a => Set [] a where
  empty = []
  add x [] = [x]
  add x (y:ys) | x==y = y:ys
               | otherwise = y:add x ys
  contains = flip elem
```

Figure 2.6: Representing a Set with a Tree or a list

```
singleton :: Set s a => a -> s a
singleton x = add x empty
```

Unfortunately we cannot actually create sets without somehow specifying which of the two implementations must be used. If we for example try to invoke the function `singleton` to actually create a set, this is what happens:

```
? singleton 2

ERROR: Unresolved overloading
*** type        : Set a Int => a Int
*** translation : singleton 2
```

The only way to resolve this overloading is to provide an explicit type declaration that tells what implementation we want for the set as follows:

```
? singleton 2 :: Tree Int
Leaf 2
(5 reductions, 16 cells)
```

So when it comes down to it, we cannot actually create a `Set` without explicitly stating what internal representation we want for it. It was the intent of the benchmark problem that a suitable internal representation would be chosen automatically on the basis of the type of the elements. This would be more in accordance with the spirit of an abstract data type that the user should not be concerned with its internal representation.

Apparently, Gofer's power for decision making is capable of diversifying the operations and implementation of operations on a single data structure. However it is incapable of making decisions about the data representation itself. What is possible in Gofer for example is to provide two different implementations of the `Set` functionality on a single data structure, for example a list. For `Eq` element types the list could just be an unsorted list whereas for `Ord` elements the list could be a sorted list instead. This is possible because the internal representation in both cases is the same, only the functions that act upon it are implemented differently.

Note that in this case, the type `Int` is an instance of both `Ord` and `Eq`[13]. This is not really the cause of the error however. The same error would still be reported even when using a type of element which is only an instance of `Eq` and not of `Ord`.

2.9.2 Iteration Benchmarks

Another important feature for a programming language is its capability to express iterations. Some programming languages offer primitive iteration constructs, others simply use recursion to get iterations as a special case. In this section we probe the expressiveness of the type language to express iteration or recursion.

The following benchmark requires a form of iteration or recursion to deal with a data structure that can be instantiated with different arity every time. Its arity is nevertheless static and known at compile time. In a way, this benchmark tests the type languages' capability to handle lists of types and counting with (positive) integers.

[13]Since the class `Eq` is declared a super class of `Ord` this will be the case for all instances of `Ord`

Benchmark 4 *Define a tuple-like container type, that contains a statically known number of elements of types a_0, a_1, $\ldots a_{n-1}$. Implement accessors* `project0`, `project1`, `project2` *etc. on this tuple type. These accessors return the first, second, etc. element from the tuple respectively.*

It would seem that this is an impossible task for the Gofer type system at first. For one thing, this task requires a counter to "compute" the names of the functions `project1`, `project2`, etc. Obviously Gofer's type language does not have integers in it so this seems impossible. Another problem is that there is no way to express recursion over standard Gofer tuples. A tuple of arity 3 has no connection with a tuple of arity 2 for example. Gofer does offer the possibility to express recursive type programs over types by means of instance declarations. It also has native tuple type constructors. Tuple types are very much like lists of types. Regrettably, Gofer's tuple types are not inductively defined: there is no relationship between tuples of arity $n+1$ and tuples of arity n. Therefore we cannot use native Gofer tuples to solve this benchmark problem since we cannot exploit recursion to define operations on arity $n+1$ tuples from the operations on arity n tuples.

Summarizing the above, we identify two basic problems with Gofer's type language that seem to make it impossible to express benchmark 4.

- The type language has no integers.
- The type language has no recursively defined lists.

Gofer's type language is nevertheless very expressive and we can get around these problems by using (abusing?) the type system in a creative way. We start by tackling the lack of integers by building our own "type numerals". With type numerals, we mean types t_0, t_1, t_2, etc. that can be used as representations for the the numbers 0, 1, 2, etc. respectively. This is accomplished fairly easily. We start with the number 0 and define the type `Zero` that will represent it as follows.

```
data Zero = Zero
```

Note that Gofer allows using the same name for the type and the constructor. In this case, `Zero` has two meanings. When it occurs in a type expression it denotes the type `Zero`. When it occurs in a regular Gofer expression on the other hand it denotes the zero-arity constructor `Zero`, which is in fact the only value of type `Zero`.

Now we will define a type constructor with which the rest of the positive integers greater than 0 can be inductively constructed by applying an `Inc` type constructor to the preceding type numeral. The type `Inc Zero` for example will represent 1 and the type `Inc (Inc Zero)` represents 2 etc. Again there is precisely one value for each of these type numerals which has the same name as the type itself.

```
data Inc a = Inc a
```

We will define a type class `Number` to group together all types representing numbers. This type class has no members[14] since we only use it to group the number types under a common denominator. The class `Number` serves as a kind of "meta type" for "numeric values" in the type language. Every type which is an instance of the class `Number` represents a positive integer.

[14]Do not confuse the word "member" with the word "instance".

```
class Number a
instance Number Zero                -- Zero is a number
instance Number a => Number (Inc a) -- A number+1 is also a number
```

The next thing we need is a representation for lists of types. As we explained, tuples seem natural candidates for this. Regrettably Gofer's native tuples are not usable because there is no inductive relationship between tuples of arity $n+1$ and tuples of arity n. Again we resort to the use of some `data` type constructors and a type class to define our own lists of types.

```
data Empty    = Empty     -- An empty List
data Cons a b = Cons a b  -- Consing a type to a list of types

class List l
instance List Empty                -- Empty is a list
instance List l => List (Cons a l) -- Consing to a list yields a list
```

Now we are almost done. We have all of the machinery we need. We have lists of types and we have positive integer numbers represented as types. We still need to define the projection functions `project0`, `project1`, etc. This means that we somehow need to incorporate a number into the name of the function. The "trick" here is to observe that in an overloaded function, it is as if the type of a function's argument virtually becomes part of the name of the function. Therefore, instead of defining functions `project0`, `project1`, etc. we will simply define an overloaded function `project`:

```
class (List t,Number idx) => Project t idx a where
  project :: idx -> t -> a
```

The type of the first parameter, `idx`, identifies which projection function is used. This is nearly the same as having the type part of the name since there is only one value of every type numeral. Instead of writing `project0` for example, we simply have to write `project Zero` and instead of writing `project1` we will have to write `project (Inc Zero)`. Note that the function `project Zero` and `project(Inc Zero)` are truly different projection functions, and will effectively have a different return type for tuples with different types of elements at position 0 and 1.

Let us now take a look at how we can define projection functions for all kinds of tuples inductively. This is rather intuitive. We first define the "`project0`" function on all tuples with at least one element in them as follows.

```
instance List l => Project (Cons a l) Zero a where
  project Zero (Cons x _) = x
```

Next we provide an inductive definition for projection on an index greater than zero.

```
instance Project t i a => Project (Cons b t) (Inc i) a where
  project (Inc i) (Cons _ t) = project i t
```

This definition states that a projection to index `i+1` is accomplished by invoking the projection to index `i` on the remainder of the tuple after stripping of the first element.

Except for the rather verbose representation of numbers and lists of types, this solution is exactly what benchmark 4 requested. There are however a number of problems with this solution. Because of the way we used multi-parameter type classes we run a very high risk of running into problems with "unresolved overloading" and "ambiguous type" errors. These can sometimes but not always be solved by providing explicit type declarations. Consider the following example where we try to project a "pair" onto its element at position 0.

```
? project Zero (Cons 'a' (Cons 1 Empty))

ERROR: Unresolved overloading
*** type        : Project (Cons Char (Cons Int Empty)) Zero a => a
*** translation : project Zero (Cons 'a' (Cons 1 Empty))
```

This is the same old problem we encountered already several times before. The type system will not notice that there is only one possible candidate for the type variable `a`. If we tell the type system explicitly what type we are expecting then it carries out this example without complaining:

```
? project Zero (Cons 'a' (Cons 1 Empty)) :: Char
'a'
(2 reductions, 13 cells)
```

It is not always possible to solve this kind of problems by means of an explicit type declaration however. Consider for example the following function which is intended to be a projection function for nested tuples. It works by composing two projection functions with one another by means of the function-composition operator ".".

```
double_project :: (Project tt i1 t, Project t i2 a) => i1 -> i2 -> tt -> a
double_project i1 i2 = project i1 . project i2

*** ambiguous type : (Project a b c, Project c d e) => d -> b -> a -> e
*** assigned to    : double_project
```

This problem cannot really be solved by explicit type declarations. We cannot disambiguate the intermediate type `t` without losing genericity. Another idea might be to try and solve the problem by using constructor classes instead of type classes. This solved the problems when we encountered them before in section 2.7.2. Regrettably this is also impossible. We would need to define some kind of tuple constructor that takes a list of types as argument and builds a tuple type from this. This simply cannot be defined because type constructors cannot make decisions based on the kind of argument they get. Therefore they cannot decide whether the argument type is an empty or non-empty list, or even whether it is a list at all.

2.10 Summary

In this chapter we have taken a close look at the type-language of the functional programming language Gofer. We consider this to be a type system that comes close to our ideal of having a true programming language as part of the type language. The introductory sections gave an overview of Gofer and its type language. We discussed the type language gradually, introducing new features in a way that illustrates how an increasingly more expressive type language increases the flexibility for dealing with various forms of polymorphism. As the type language becomes more complex, it starts to resemble a combination of a logic programming language and a functional programming language with which static types can be manipulated as data at compile time.

In section 2.9 we have explored the potential of Gofer's type language as a programming language to manipulate types at compile time. We did this by means of some benchmark programming assignments which were carried out in Gofer. The results of these benchmarks are summarized in figure 2.7.

Decision-making benchmarks	
Benchmark 1: Conditional implementation of a function `contains` on a list.	Can be elegantly expressed
Benchmark 2: Abstract implementation of `contains` for `Enumerable` collection types.	Can be elegantly expressed.
Benchmark 3: Implementation of a `Set` abstract data type the internal representation of which depends on the type of elements stored.	Can almost be expressed as wanted. The only defect in the solution is that the user of the set abstract data type will have to indicate the correct internal representation upon creating a set. Gofer's type language *cannot* support making decisions affecting the representation of a data type. It *can* make decisions affecting how and whether some functionality is implemented for a specific data structure.

Iteration / recursion benchmarks	
Benchmark 4: Generic implementation of projection function on tuples of different arities.	We succeeded eventually after a lot of effort. The solution feels like "walking on the edge of a volcano", just within the boundaries of the expressiveness of the type language, using it in ways it was not intended to be used: • We had to define our own type numerals. • We had to create our own lists of types because native tuple types could not be used for this purpose. • We used overloading to simulate function names containing a counter. • When we started using the projection functions, ambiguous type errors caused problems which could not always be solved.

Figure 2.7: Summary of the benchmark results for Gofer

The overall conclusion is that Gofer's type language is remarkably expressive[15]. The

[15]There is a strong suspicion that Gofer's type language is in fact theoretically Turing Complete in the

examples we gave in section 1.6 to illustrate problems in existing parametric-type proposals for Java were solved quite naturally by Gofer's type language. Nevertheless, it clearly was not intended to be a true programming language. An excellent illustration of this is the solution for benchmark 4, the implementation of generic tuples with different arity. We were able to express the benchmark program *eventually* with a great deal of effort. It is clear however that we are very near to the boundary of the expressiveness of Gofer's type language. Clearly the type language was not designed to be a real programming language because, for example, we had to implement our own integers. Nobody expects to have to do this in a real programming language. We also had to define our own "list of types" representation because the native tuple types could not be used for this purpose, even though they seem to be natural candidates. Finally, ambiguous-type errors and problems with unresolved overloading are hiding behind every corner when we start using the tuples we have defined. Because of this we probably would only be able to use them in very limited ways .

Note that the criticism of Gofer's type language given in this chapter should be seen in its proper context. We mainly want to illustrate the point of our thesis that decidability and unambiguity of a type system is not an absolute rule, and that undecidable and potentially ambiguous typing schemes can also be useful. The usefulness of such type systems would be the power of the type language as a programming language. The goals of the designers of Gofer however were completely different from ours. They did not design their type language as a programming language, for maximal expressiveness. Instead a number of strongly limiting design constraints were adopted as prerequisites. Given this, it is remarkable how expressive a type system they were able to design and implement in an efficient way. The power of Gofer's type language comes mostly from the mechanism of qualified types, which introduces a kind of logic programming language into the type system. Despite the limitations imposed by Gofer, the power of this approach is apparent and was the major motivating force behind the choice of a logic meta language for our system.

sense that any "program" that computes a boolean result, can be translated into a Gofer program the type correctness of which corresponds exactly to the computed truth value.

Chapter 3

Java and Parametric Types

3.1 Introduction

The purpose of this chapter is to provide an overview of existing parametric types proposals for Java, and to argue that they do not offer adequate support for type-oriented programming. Concretely, we will discuss the proposals of Odersky and Wadler (Pizza) [OW97], the proposal of Agesen, Freund and Mitchell [AFM97], the proposal of Myers, Bank and Liskov [MBL97], and Bruce's proposal [Bru97].

In chapter 1 we already briefly discussed how extending Java with parametric types introduces a rudimentary functional language into the type system. Therefore introducing parametric types is a step towards type-oriented meta programming for object-oriented languages. For this reason it is interesting to make a thorough study of existing Java parametric type extensions. We will consider these proposals mostly from a user's point of view because we are primarily interested in their expressiveness and not as much in their implementation.

In section 3.2 we will give a general overview and a comparison between existing proposals, pointing out the similarities and differences between them. For completeness we will also discuss another approach to genericity, known as *virtual types* [Tho97] in section 3.4. Virtual types are an alternative to parametric types. Both approaches have their merits and shortcomings as pointed out by Bruce et al. who integrate both in a single proposal [BOW98].

The rest of the chapter, from section 3.5 on will mainly be spent arguing that the functional language Gofer, discussed elaborately in chapter 2, comes much closer to our ideal of having a true programming language for a type language. In this respect, current parametric types proposals for Java cannot compete with Gofer. We will argue about this using the same benchmark programming assignments we also used to assess the expressive power of Gofer's type language. The benchmark results will clearly demonstrate that current Java parametric type extension come nowhere near the expressive power of Gofer's type language. Except for one exception, none of the benchmark problems can adequately be expressed in any of the existing Java parametric types proposals. Gofer did not get a perfect score either, nevertheless it did allow the expression of several of the benchmarks quite elegantly.

3.2 Parametric and Bounded Parametric Polymorphism

In order to have a common syntax with which to express example code and discuss differences between existing proposals, we present a fictional Java extension, PJava, which offers parametric and bounded parametric types. We purposefully do not give precise specifications of this fictional language extension. We are just going to use it informally, as a uniform syntactic notation which will allow us to accentuate the semantic differences of existing proposals without getting caught up in discussions about unimportant syntactic differences.

PJava adopts as a lexical convention that type parameters and type variables start with a "?". This makes them easy to recognize. Other proposals do not follow this lexical convention, instead type variables are distinguished from other identifiers by the way they are used. Syntactic differences such as identifying type variables by means of a "?" are not really important. The reason we adopt this lexical convention is that it will result in a more obvious syntactic correlation of PJava programs and the emulation of bounded parametric polymorphism in TyRuBa which we will present in section 7.3.

3.2.1 Parametric Classes and Interfaces

The syntax of a PJava parametric class or interface declaration is similar to the Java class or interface declaration syntax, with the difference that an additional type-parameter list is appended after the name of a class or interface. The parameter list is delimited by angular brackets "<" and ">".

Parametric types are very useful to support the use of generic data-structures. Every Java programmer knows painfully well that using generic data structures, such as for example stacks and lists, in Java programs usually implies writing bundles of dynamically-checked typecasts. The reason for this is that in Java the only way to define generic data structures is to use the type `Object` as element type. This will allow all types of elements to be stored in the data structure. However, as a result the static type of the stored elements is lost. Therefore it is often needed to use a typecast to reassert the forgotten static type. The solution to this problem is the use of a parametric class `Stack<?a>`. In this, the symbol `?a` is a type variable denoting a type parameter. Thus, `Stack` is not really a class, but is more like a function accepting a type `?a` as argument and creating a class `Stack<?a>` from it. Note the similarity to a type constructor in Gofer. Below is an example of a parametric `Stack` class declaration.

```
class Stack<?a> {
  private ?a[] elements = new ...;
  private int top = 0;

  public ?a pop() {...}
  public void push(?a e) {...}

  public boolean empty() {...}
}
```

Because the type of a stack now explicitly states the type of the elements stored in them, a typecast will no longer be necessary when retrieving for example `String` elements from a `Stack<String>`.

```
  Stack<String> stack = new Stack<String>();
  String s;

  while (...more strings?...) {
    s = ...get a string...;
    stack.push(s);
  }

  while (!stack.empty()) {
    s = stack.pop();  // NO Type Cast!
    ... do something with s ...
  }
```

3.2.2 Bounded Parametric Polymorphism

The `Stack` class is a relatively easy example because it can be instantiated with any type of elements. No messages are ever sent to elements, they are simply stored and retrieved. The situation is more complex when we have to deal with data structures which also send messages to their elements. As an example of this consider the implementation of a hashtable[1]. A hashtable class is parameterized by two types. One type for the keys and another for the values associated with the keys. The implementation of a hashtable requires computing a hash value for a key. This will be accomplished by sending a key the message `hashCode`. An implication of this is that not just about any type may be used as the key type for a hashtable, only a type that supports the message `hashCode` should be allowed. To be able to express this, PJava's syntax is extended, allowing constraints on type variables to be expressed in a `where` clause, added to a class or interface declaration.

As an example we present the outline of a `Hashtable` implementation in PJava below.

```
interface Hashable {
  int hashCode();
}

class Hashtable<?k,?v>
where ?k implements Hashable
{
  public void put(?k k,?v v) { ... k.hashCode() ... }
  public ?v get(?k k) { ... }
}
```

A *bounded* type parameter is constrained by a condition in the `where` clause. In the above example the `?k` parameter is constrained by the condition that it must implement the interface `Hashable`. Other "predicates" for constraining type variables are possible. We will not go deeper into this right now. The main semantic difference between parametric-type proposals is precisely the kind of conditions that can be expressed in order to constrain type parameters. We will discuss the kinds of constraints that are expressible when we discuss individual proposals in the following section. A summary of the different type-constraining predicates offered by different proposals can be found at the end of this section in figure 3.4.

The idea of where clause syntax comes from Myers, Banks and Liskov's proposal [MBL97]. Other proposals typically attach a condition to a type variable in the type-argument list directly. We chose to adopt a where-clause-like syntax because it is more general and because

[1]Example taken from [AFM97].

it corresponds more closely to the emulation of bounded polymorphism we will present in section 7.3.

3.3 Concrete Parametric-Type Proposals

In this section we will have a look at the differences and similarities between individual proposals. We will discuss them mostly from a user's point of view, and avoid the details of their implementation as much as possible. Some of the semantic differences however stem from the difference in the underlying implementation. For this reason we will briefly discuss the two main categories into which implementation strategies can roughly be divided.

3.3.1 Pizza and the Agesen-Freund-Mitchell proposal

We start with the proposal of Agesen, Freund and Mitchell (from now on referred to as AFM) [AFM97] and the proposal of Odersky and Wadler (Pizza) [OW97]. We treat these together because they are highly similar. They mainly differ in the way they are implemented while their semantics is very similar. There are some differences in flexibility, mostly dictated by implementation-related concerns.

As an example we present the outline of a parameterized `PriorityQueue` class below. This example was taken verbatim from the AFM paper and then translated into PJava syntax.

```
interface Comparable<?I> {
  boolean lessThan(?I);
}

class PriorityQueue<?T>
where ?T implements Comparable<?T>
{
  ?T queue[];
  void insert(?T t) {
    ...
    if (t.lessThan(queue[i])) ...
    ...
  }
  ?T remove() { ... }
  ...
}
```

The kind of type constraints that can be expressed to constrain type variables in both proposals are of the form of an extends or an implements clause. These both constrain the type variable in a similar way. An "`X extends C`" constraint restricts `X` to be the class `C` or a subclass thereof. A constraint "`X implements I`" restricts `X` to a class or interface, implementing or extending the interface `I` respectively. In other words, both of these constraints correspond to the Java subtype relationship between classes and interfaces.

The differences between Pizza and AFM are mainly due to the different implementation strategies. Pizza is implemented using a *homogeneous* strategy whereas AFM is implemented using a *heterogeneous* strategy. The terminology of heterogeneous versus homogeneous implementation was introduced by Odersky and Wadler [OW97]. We briefly explain the difference between the two.

Homogeneous versus heterogeneous implementation

A homogeneous and a heterogeneous implementation differ in the way code is shared between different instantiations of a parametric class. A homogeneous implementation uses shared code for different instantiations whereas a heterogeneous implementation duplicates the code each time a parametric class is instantiated. As an illustration, we have listed in figure 3.1 and figure 3.2 a homogeneous and a heterogeneous translation of the parametric stack example into Java. Note the type cast which is inserted by the homogeneous translation.

```
/** Shared code for all instances of Stack<?a> */
class Stack {
  private Object[] elements = ...;
  private int top = 0;

  public Object pop() {...}
  public void push(Object e) {...}

  public boolean empty() {...}
}

/** Code using the Stack performs type casts */
  Stack stack = new Stack();
  String s;

  while (...more strings?...) {
    s = ...get a string...;
    stack.push(s);
  }

  while (!stack.empty()) {
    s = (String)stack.pop();  // Type Cast Inserted
    ... do something with s ...
  }
```

Figure 3.1: Homogeneous translation of the Stack example

An implementation using a homogeneous strategy tends to be more restrictive than a heterogeneous translation with respect to where type parameters may be used. The reason for this is that in order to be able to share the exact same code between different instantiations of a parametric type, some restrictions must be imposed. Agesen [AFM97] provides an example that can be handled by AFM but not by Pizza, illustrating this kind of restriction. The given example is a *mixin* class, i.e. a class which uses a type parameter in an `extends` clause. This class can be instantiated to add functionality to some base class passed to it as a parameter. This usage of a type variable is allowed in AFM but not in Pizza. The reason for this is that

```
/** Specific code instantiated for Stack<String> */
class Stack_String {
  private String[] elements = ...;
  private int top = 0;

  public String pop() {...}
  public void push(String e) {...}

  public boolean empty() {...}
}

/** Code using the Stack */
  Stack_String stack = new Stack_String();
  String s;

  while (...more strings?...) {
    s = ...get a string...;
    stack.push(s);
  }

  while (!stack.empty()) {
    s = stack.pop();  // No Type Cast Inserted
    ... do something with s ...
  }
```

Figure 3.2: Heterogeneous translation of the Stack example

Pizza's homogeneous translation implies that a single class file[2] must be shared between all instantiations of a class. Because a Java class file specifies the class file containing the super class from which a class inherits, this implies that the super class must be a known class and cannot be a type parameter. A declaration of the kind

```
class C<?x> extends ?x { ... }
```

will therefore not be allowed by Pizza, but is allowed in AFM.

The above is an example that shows how a heterogeneous translation seems to be more interesting for obtaining a more expressive type system. It is however not all that clear cut which one is best, because a homogeneous implementation also has advantages. Most importantly, it allows parameterized methods. A parameterized method is a method which uses a type variable that is *not* a parameter of the class in which it occurs. An example of this is a `zip` method which is implemented on a parametric `List<?a>` class. The zip method takes as argument a `List<?b>` and constructs a new list by pairing the elements of the two lists element by element[3].

```
public class Pair<?a,?b> {
  public ?a fst;
  public ?b snd;
}

public class List<?a> {
  ...
  public List<Pair<?a,?b>> zip(List<?b> b) { ... }
  ...
}
```

This is an example that is expressible in Pizza but not in AFM. To understand why, take a look at the homogeneous translation of the example.

```
public class Pair {
  public Object fst;
  public Object snd;
}

public class List {
  ...
  public List zip(List b) { ... }
  ...
}
```

The important thing to notice is the type of the `zip` method. The homogeneous translation removes type variable and replaces the types `List<?b>` and `List<Pair<?a,?b>>` by `List`. The class `List` is the homogeneous translation for `List<?a>` and will be used for every instance of

[2]A class file is a binary format for storing compiled classes for the Java Virtual Machine.

[3]The `zip` example was borrowed from [BOW98]

this parametric class. Similarly, the `zip` method does not represent one method but is used as the implementation of an infinite number of methods, one method for each possible value of the type variable `?b`. Because a homogeneous translation replaces all instances of `List<?b>` by `List`, it is able to reuse the same fysical method declaration for each one of the virtually infinite number of `zip` method instantiations. This makes the implementation of a parametric method like `zip` relatively easy. A heterogeneous implementation on the other hand does not do this kind of code sharing and therefore it should in principle include code for an infinite number of `zip` methods. Clearly that is not possible. This does not mean that implementing parametric methods is entirely impossible in a heterogeneous setting, it is however certainly much more complicated. A possible implementation could for example determine which of the infinite number of methods are actually used and only generate code for these. These complications in implementing parametric methods in a heterogeneous setting is probably one of the reasons why AFM does not support them.

3.3.2 Bruce's proposal: LOOMJava

Bruce [Bru97] also proposes a Java extension offering parametric types. Essentially his proposal is a transplantation of the LOOM type system [BPF97] into Java. We will therefore refer to Bruce's proposal as LOOMJava from here on.

As far as parametric types are concerned, LOOMJava is very similar to Pizza and AFM. There are however a number of other extensions and changes to the type system that increase its flexibility to deal with with *binary methods*[BCC+95].

- A `ThisType` construct.
- Use of `@` to signal an *exact* type.
- Java subtyping is replaced by *matching*.

We briefly discuss the extensions and changes to the Java type system. For a more elaborate discussion we refer to [Bru97].

ThisType, matching and exact types

`ThisType` provides for a kind of "late binding of self" in the type system. This is useful for typing methods such as for example the `clone` method. In Java this method cannot be adequately typed and is declared as returning `Object`. The following kind of code is therefore very frequently seen in Java programs:

```
SomeVerySpecificClass x = (SomeVerySpecificClass)x.clone(); //Type Cast!
```

In reality the clone method always returns a value of exactly the same type as the object you send it to. In LOOMJava this would be declared as follows:

```
interface Clonable {
  @ThisType clone();
}
```

The introduction of `ThisType` poses a complication with respect to subtyping and inheritance. If a class (or interface) using `ThisType` is extended, the subclass is not usually a subtype[4] of the extended class. The reason for this is that the type `ThisType` changes when moving from a superclass to a subclass. We provide a classical example to illustrate this:

```
class Float {
  float value;

  Float(float v) { value=v; }

  @ThisType add(@ThisType a) {
    @ThisType result = this.clone();
    result.value+=a.value;
  }
}

class Complex extends Float {
  float imag;

  Complex(float v,float i) { super(v); imag=i; }

  @ThisType add(@ThisType a) {
     @ThisType result = super.add(a);
     result.imag += a.imag;
  }
}
```

In the class `Float` the method `add` has signature[5]:

```
@Float add(@Float);
```

In the class `Complex` the signature becomes:

```
@Complex add(@Complex);
```

Therefore the `Complex` class is not a subtype of the `Float` class because it overrides the `add` method covariantly. Covariant overriding means that the argument of the overridden method is changed into a subtype of the former argument type. A consequence of covariant overriding is that the subclass cannot simply be used in a context where an instance of the superclass is expected and thus is not a subtype of the superclass. To illustrate this we present the following example where problems arise because an object of type `Complex` is stored in a variable of type `Float`.

[4]A *subtype* of a type T is a type T' such that a value of type T' can be substituted for a value of type T, in any context, without introducing type errors.

[5]This is not entirely correct because Bruce explicitly states that `ThisType` refers to the interface of the class and not to the class itself. For the sake of obtaining simpler example code and argumentation, we chose to ignore this fact.

```
Float c = new Complex(1,-1);
Float f = new Float(1);
Float sum = c.add(f); //ERROR
```

The call of `add` in this example should not be allowed because the complex `c` should only accept another complex as argument. The conclusion is that `Complex` is not a subtype of `Float`.

It would seem that because `Complex` is no longer a subtype of `Float` the assignment to `c` above should be disallowed. However, LOOMJava solves the problem differently. The subtyping relationship is weakened to an alternative relationship called *matching*. Whether a type *matches* another type corresponds to `implements` and `extends` between classes and interfaces. The class `Complex` matches `Float` for example. Practically speaking, the relationship remains the same, but the name for it changes because technically speaking, the term "subtype" is no longer correct in the presence of the `ThisType` feature. The assignment above is thus still allowed. Instead, in order to keep type-safeness, it is no longer allowed to send binary methods to `c`. Objects to which binary methods are sent must be declared with an exact type. So the following program excerpt for example would be acceptable:

```
@Complex c1 = new Complex(1,-1);
@Complex c2 = new Complex(2,2);
@Complex sum = c1.add(c2); //OK
```

Parametric and Bounded Parametric Types

Except for the fact that instead of subtyping, matching is used to constrain type parameters in LOOMJava, the mechanism of bounded parametric polymorphism is essentially the same as in AFM and Pizza. Type parameters can be constrained by a predicate of the form

```
X extends I
```

which signifies that `X` *matches* `I` rather than that `X` is a subtype of `I`. It is interesting to note that for programs which do not use `ThisType` or `@`, matching and Java subtyping are completely identical.

3.3.3 Myers, Bank and Liskov's proposal

The proposals discussed up until now—Pizza, AFM and LOOMJava—are highly similar. They all allow restricting type parameters by an `extends` or `implements` clause. Myers, Banks and Liskov's proposal [MBL97] offers a different mechanism. Type parameters are constrained by stating a *where clause*[6] which restricts a type variable to types which support a given set of method and constructor signatures. We will therefore refer to this proposal as WhereJava from here on.

As a first simple example, we reconsider the `PriorityQueue` class, this time expressed with the kind of type-constraint supported by WhereJava.

[6]When Myers et. al. are talking about where clauses they implicitly assume signature-conformance type constraints. We also use where clause syntax in PJava for other kinds of type constraints to obtain a uniform syntactic notation.

```
class PriorityQueue<?T>
where ?T { boolean lessThan(?T); }
{
  ?T queue[];
  void insert(?T t) {
    ...
    if (t.lessThan(queue[i])) ...
    ...
  }
  ?T remove() { ... }
  ...
}
```

WhereJava's type constraining predicates are generally more flexible than those of the other proposals. Whether a type satisfies a list of signatures is not determined by an exact match, but also allows for contravariant method types. Another advantage is that it is possible to restrict types not only on the basis of the methods they support but also on the basis of the constructors they support.

WhereJava has one other interesting extension. It allows where clauses not only to be attached to classes and interfaces, but also to individual methods. A method with an individual where clause is a conditional method and it is only allowed to be called on a particular instance of the parametric type in case the constraint in the where clause is satisfied. This enables the implementation of benchmark problem 1 as listed in figure 3.3.

```
class Array<?El> {
  ?El[] contents;

  Array<?El>(int size) { contents = new ?El[size]; }

  ?El at(int i) { return contents[i]; }
  void atPut(int i, ?El e) { contents[i]=e; }
  int length() { return contents.length; }

  boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equals(at(i++));
    return found;
  }
  where ?El { boolean equals(?El); }
}
```

Figure 3.3: Conditional implementation of `contains` in Where(P)Java

3.3.4 Summary

From a user's point of view existing parametric types proposals for Java mainly differ in the kind of type-constraints they allow for bounded parametric types. A short overview

Proposal	Predicate	Meaning

Pizza/AFM	`?T extends ?C`
	`?T implements ?I`
		?T is a *subtype* of class `?C`
		?T is a *subtype* of interface `?I`

LOOMJava	`?T extends ?I`	?T *matches* `?I`.
WhereJava	`?T` {*signatures*}	?T *supports* all methods and constructors the signatures of which are listed between `{}`.

Figure 3.4: Type-constraint predicates available in different proposals

can be found in figure 3.4. Apart from the difference in the predicates that can be used to constrain types, there are also some other differences in flexibility which seem to stem from the underlying implementation strategy. It is not clear which proposal is "best". Let us summarize and compare them quickly.

The constraints expressed in WhereJava seem to be the most flexible. Advocates of the other proposals point out however that it introduces the notion of signature conformance into Java. This seems to overlap to some extent with the already existing notion of interfaces. Therefore the orthogonality of its design might be criticized. The other proposals integrate their constraint predicates more neatly with existing Java interfaces and the Java subtype relationships, but are less flexible.

Pizza and AFM both use the Java subtype relationship to constrain type variables. There are differences between the two because of their differing implementation strategies. Which one is best is hard to say. Pizza supports parametric methods and AFM does not. At the same time however, Pizza is more restrictive with respect to where type variables are allowed to occur.

We have not enough detailed knowledge about or experience with LOOMJava and its implementation to compare it with AFM or Pizza on finer points. Potentially, given either an adequate heterogeneous or homogeneous implementation its expressive power could at least equal that of either AFM or Pizza. Additionally it has some extra features to support the typing of binary methods, and can thus be seen as an extension of either of these proposals (which proposal depends on which implementation strategy is chosen).

3.4 Virtual Types and Related

3.4.1 Thorup's Virtual Types Proposal

The idea of virtual types is based on Beta's[MMPN93] notion of virtual class patterns. Virtual types were first proposed as an alternative generic types mechanism for Java by Thorup [Tho97]. It introduces the possibility to declare types as attributes of classes. Such a type attribute is called a *virtual type* because it can be specialized in a subclass. Instead of instantiating a parametric type, one therefore uses subclassing to concretize generic types. As an example, consider the following definition of a generic pair class using virtual types.

```
class Pair {

  /** Virtual types A and B for the components of the Pair */
  typedef A as Object;
  typedef B as Object;
  A car;
  B cdr;

  public A getCar() { return car; }
  public B getCdr() { return cdr; }
}
```

More concrete or specific pairs can be obtained by subclassing this class and overriding the types A and/or B.

```
class StringPair {
  typedef A as String; //Override
  typedef B as String; //Override
}
```

3.4.2 Bruce's Alternative to Virtual Types

It is argued by Bruce et al. [BOW98] that both parametric types *and* virtual types have their merits and defects and are in a way complementary. They present several examples to illustrate this. Some examples are more easily expressed with parametric types, others are better handled by virtual types. Typically the examples which work well with one proposal can also be expressed by the other, but often involving great difficulty and complications which make the resulting code virtually unusable in practice.

Typical examples which work well with parametric types but not with virtual types, are generic collections, such as lists, stacks, etc. Examples which work well with virtual types but not with parametric types typically involve families of types. For more details and specific examples we refer to [BOW98].

Because the proposals appear to be complementary, Bruce et al. [BOW98] present a proposal which combines the strengths of both systems. Basically their proposal comes down to a generalization of Bruce's former proposal (LOOMJava) [Bru97] we discussed in section 3.3.2. The notion of `ThisType` is generalized so that a group of classes or interfaces may assign names to refer to each other's "`ThisType`".

The declaration of such a "`ThisType`" name is similar to a virtual-type declaration. This proposal also adopts the other extensions proposed as part of LOOMJava: parametric types, exact types, and matching. This results in a statically type safe alternative to Thorup's virtual types which require runtime type checks.

We will not elaborate on this any further and refer to [Tho97, Bru97] for details. In section 11.10.2 we will come back to the issue of virtual types and how it relates to our approach.

3.5 Expressiveness of Parametric Java Proposals

We want to asses the expressiveness of the existing parametric-types proposals with respect to the benchmark problems from chapter 2. These benchmarks were aimed at probing the type-language's capability to facilitate type-oriented meta programming. We will present example code in the uniform syntax of PJava. We have not given precise specifications of

PJava. As explained before this is intentional. We merely use PJava syntax to be able to express all example code using a uniform syntactic notation. We will be careful to provide sufficient explanation to clarify the relationship between PJava example code and existing parametric-types proposals.

In the following subsections we will discuss the benchmark problems one by one. As a reminder we will repeat each benchmark programming assignment at the start of the subsection that discusses it.

3.5.1 Conditional Interface Implementation

Benchmark 1 Implement a function `contains` on a `List` (or `Array`) data structure that searches the list for the presence of an element by means of an equality test on elements. If the elements in the list do not support the equality test simply omit the function or method declaration but do not prohibit the instantiation or use of the data structure altogether.

In the introduction (chapter 1) this example was used to illustrate that most parametric types proposals' type languages have inadequate decision making capabilities.

In this particular example we would want to make the implementation of the inclusion of the `contains` method conditional, depending on whether the type parameter supports testing for equality. Most parametric type systems do not provide a way to do this. WhereJava is an exception because it allows where clauses to be attached to individual methods as was noted earlier. Therefore, benchmark 1 can be expressed in WhereJava. We already presented the implementation of benchmark 1 in figure 3.3 as an example.

3.5.2 Positioning Abstract Code

In the previous section we have seen that WhereJava has a somewhat better decision-making capability than the other proposals because it allows attaching type constraints to individual methods. As a result the simplest decision making benchmark problem, benchmark 1 can be expressed. However, it will become clear in this section that decision making capabilities still remain limited and cannot deal adequately with the slightly more difficult benchmark problem 2.

Benchmark 2 Implement a function `contains` on all data structures which provide a way of enumerating their elements. If the elements do not support equality tests, then omit the declaration of `contains` but do not prohibit instantiation or use of the data structure altogether.

Benchmark problem 2 corresponds to the second motivating example from section 1.6.2. In that section we already briefly discussed the problems in expressing this benchmark. We will give a more detailed and structured discussion in this section. We will discuss three alternative attempts which all fall short of a good and complete solution. The first one is an attempt using abstract classes, the second one uses a mixin class, and the third one tries to exploit the decision making implicitly supported by attaching a where clause to an individual method. For each of these attempts we will discuss why they are inadequate or incomplete solutions to the benchmark problem at hand.

Abstract Classes

The most obvious place—in many proposals actually the only possible place—to put abstract code, is an abstract class. In such a solution we would provide the `contains` method as part of an abstract class and rely on concrete subclasses to provide the specific implementation for `Enumerable` functionality. The abstract class code from the introduction is repeated in figure 3.5.

```
interface Enumerable<?El> {
  Enumeration<?El> elements();
}
interface Enumeration<?El> {
  boolean hasMoreElements();
  ?El next();
}
abstract class Searchable< ?El >
where ?El implements Equality<?El>
implements Enumerable<?El>
{ boolean contains(?El e) {
    boolean found = false;
    Enumeration<?El> elems = elements();
    while (!found && (elems.hasMoreElements()))
      found = e.equals(elems.next());
    return found;
  }
}
```

Figure 3.5: Abstract Class Implementation of `Searchable`

If we consider how to integrate this kind of code into an already existing class hierarchy it quickly becomes clear why using an abstract class is not always an adequate way of dealing with abstract code. The `Enumerable` functionality could be spread out all over the class tree and be implemented by several otherwise unrelated classes. It is thus not always possible to find a suitable place, a common super class, where we can insert the abstract `Searchable` class. This might be impossible without a complete reorganization of the class-tree structure. The problem becomes exponentially worse when several pieces of abstract code have to be integrated into one and the same class tree structure.

Intuitively, we can argue that abstract code purposefully ignores implementation details of specific classes. As a result it is typically applicable to a wide variety of classes not necessarily linked through inheritance. This is completely contradictory to the fact that putting abstract code in an abstract class requires pinpointing a specific location for it in the class tree.

Mixin

A solution which does not tie the abstract code to a specific location in the class tree can be accomplished by using a so-called *mixin*. A mixin is a piece of functionality which can be “mixed into” a class. In some parametric types proposals—AFM to name one— this can be simulated by means of a parametric class which uses its type parameter for a super class. Such a mixin class is like a function which can be applied to any class, provided that it meets the required type constraints. The example mixin class from the introduction is repeated in figure 3.6.

```
class SearchableMixin<?Super>
where ?Super implements Enumerable<?El>,
      ?El implements Equality<?El>
extends ?Super
{ boolean contains(?El e) { ... } }
```

Figure 3.6: Mixin implementation of `Searchable`

This is a more natural and elegant way of dealing with the abstract implementation of the `contains` method because unlike an abstract class, it is not fixed to a specific position in the class tree. The mixin can be applied to any class which meets the required type constraints. These classes do not have to be related to one another through inheritance. Nevertheless, this is not a true solution to the benchmark problem. Even though a mixin class *can* be applied to the classes that meet the requirements, the extension is not added automatically. Instead the user must explicitly create a subclass by applying the mixin. For example, instead of using `Array<X>`, one must explicitly use `SearchableMixin<Array<X>>` when one needs an array which supports the contains method.

As already pointed out in section 1.6.2, another problem with this mixin-class approach is that it is nearly impossible to specialize the mixin for a specific class. For example a `Set` abstract data type class, represented internally with a hashtable would benefit from a more specific implementation of `contains` that makes use of the hashtable instead of enumerating the elements of the `Set`. It is possible to define several mixin classes, each one with a different implementation strategy, but providing an implementation for the same interface. The user must however decide for himself which is the correct mixin to apply for a specific type parameter. This decision cannot be made by the type system.

Using a "conditional method"

A last attempt uses a where clause attached to the `contains` method. We declare the `contains` method in WhereJava as follows:

```
abstract class Enumerable<?El>
  ...
  boolean contains(?El e) { ... }
  where ?El { boolean equals(?El) }
  ...
```

The only way to provide an implementation for a method is by declaring it in a class. Since we want the method to affect all classes which are `Enumerable`, this means we have to have a common parent for all of these classes. In other words, we need an abstract `Enumerable` class. The `Enumerable` functionality must thus be captured at a specific point in the class tree. This again raises the problems with abstract classes which were discussed earlier. The problems merely shift from the implementation of `Searchable` functionality to the implementation of `Enumerable` functionality.

Final Conclusion for Benchmark 2

Existing proposals fall short on this benchmark problem. The closest to a good solution is probably a mixin class. Mixin classes can only be expressed in some of the proposals. Even then, a mixin class is not truly a solution to the benchmark problem because it does not actually add the `contains` method to a class but only provides a way for the user to manually add the method by explicitly creating a subclass. Another problem with the mixin "solution" is that it cannot be specialized for specific classes.

3.5.3 Type-Dependent Internal Representation

Benchmark 3 Implement an abstract data type `Set` representing a set of elements of some type `a`. When implementing a set of element types that merely support equality tests use a simple list data structure which is searched sequentially. For the implementation of a set of ordered elements, comparable by means of `<`, use a binary search tree.

This benchmark requires another kind of decision making than the two former benchmark problems. It turned out that Gofer's type language falls short just a little bit to solve this problem completely. Apparently Gofer's type language can be used to make decisions about functionality and implementation of functionality onto a data structure, but it cannot be used to express decisions that affect the data structure itself.

The situation for parametric Java is similar. We can easily declare two different implementations for an abstract `Set` class.

```
abstract class Set<?El> {
   abstract void insert(?El e);
   abstract boolean contains(?El e);
}

class TreeSet<?El>
where ?El { boolean equals(?El);
            boolean lessThan(?El); }
extends Set<?El>
{
   private contents Tree<?El>
   void insert (?El e) { ... }
   boolean contains(?El e) { ... }
}

class ListSet extends Set<?El>
where ?El { boolean equals(?El); }
extends Set<?El>
{ ... }
```

Since `Set` is an abstract class it cannot be instantiated. Thus, the situation is exactly the same as in Gofer: we must explicitly state the name of the specific concrete implementation class upon creating a `Set`. For example:

```
Set<String> x = new TreeSet<String>
```

We could try to get around the problem by means of a *factory* method or something similar.

```
abstract class Set<?El> {
  ...
  /** Factory method */
  static Set<?El> make() {
    ...???implementation of factory method???...
  }
}
```

The question of how to implement such a factory method remains to be answered. The factory method should create an instance of the right concrete subclass depending on the type `?El`. The only way to make this kind of decision based on the static type `?El` is to provide two implementations of the method and constrain each of them by an appropriate where clause as follows:

```
abstract class Set<?El> {
  ...
  static  Set<?El> make() {
     return new ListSet<?El>
  }
  where ?El { boolean equals(?El); }

  static  Set<?El> make() {
     return new TreeSet<?El>
  }
  where ?El { boolean lessThan(?El); }

}
```

Since this attaches a type constraint directly to a method the only proposal that might possibly support this kind of programming is WhereJava. None of the other proposals allows attaching constraints to individual methods. This mechanism in WhereJava can however only be used to define *optional* methods, and it is not allowed to provide two alternative method implementations this way. Code as presented above is rejected by WhereJava because it has two declarations for the method `make`. Note that this cannot be considered as a kind of overloading because both versions of the method have the same argument-type list.

3.5.4 Iteration or Recursion

Benchmark 4 Define a tuple-like container type, that contains a statically known number of elements of types a_0, a_1, $\ldots a_{n-1}$. Implement accessors `project0`, `project1`, `project2` etc. on this tuple type. These accessors return the first, second, etc. element from the tuple respectively.

This problem is too difficult to be expressed by any of the proposals. Let's try to express it and see how far we get. Before starting, note that we will have to give conditional implementations of the `project` methods. Therefore WhereJava is the only proposal which potentially could be used to express this problem. The other proposals do not offer the possibility to declare conditional methods. We therefore limit our discussion of this benchmark to WhereJava alone.

As in Gofer, we are faced with the problems that we have no numbers or lists in the type language. In Gofer we were able to implement these ourselves. We can do the same thing in PJava. For example, we can simulate type numerals by means of a class `Zero` and a parametric type `Inc`. We will need to be able to express constraints that limit type variables to be a number or a list. In WhereJava we only have signature constraints. We therefore identify numbers by providing a dummy method, the presence of which "tags" the class as being a type numeral.

```
class Zero {

  /** any class with this method is a number. The implementation of the
    method itself is irrelevant since it merely serves as a ``signature
    tag'' to recognize type numerals */
  public void numberTag() {}
}

class Inc<?x>
where ?x { void numberTag(); }
{
  void numberTag()
}
```

Similarly we can simulate lists of types:

```
class Empty
{
  void listTag() {}
}

class Pair<?a,?b>
where ?b { void listTag(); }
{
  ?a car;
  ?b cdr;

  ?a project(Zero dummy) { return car; }
}
```

In Gofer we used overloading to "simulate" the index which is part of the projection function's name. We do the same thing here, instead of defining a method `project0` for example, we add a dummy parameter of type `Zero` to the projection function. Overloading on the dummy parameter determines the index of the projection function.

This is as far as the similarity between Gofer and WhereJava goes however. The final step in the Gofer solution was the inductive definition of the projection functions for indexes greater than 0. This cannot be expressed in WhereJava because it lacks the computational power offered by Gofer's qualified type system. Inserting the following conditional method into the `Pair` class is the best we can do.

```
class Pair<?a,?b>
...
  ?return project(Inc<?pos> dummy) {
     cdr.project(new ?pos());
  }
  where ?b { ?return project(?pos); }
}
```

Unfortunately this will not work in WhereJava because the type variables `?return` and `?pos` are not parameters of the `Pair` class and therefore the above is a parametric method, a feature which is not supported by WhereJava.

3.6 Conclusion

In this chapter we have given an overview of existing parametric types proposals for the language Java. We used the benchmark programming assignments from chapter 2 to asses the capability of these proposals for type-oriented programming. The results of the benchmarks are summarized in figure 3.7.

	Gofer	AFM	Pizza	WhereJava
Benchmark 1	Yes	No	No	Yes
Benchmark 2	Yes	No	No	No
Benchmark 3	Almost	Almost	Almost	Almost
Benchmark 4	Yes, but ...	No	No	No

Figure 3.7: Benchmark results for Java parametric-types proposals

The overall conclusion is that existing parametric types proposals perform poorly on the benchmark programming assignments. Only WhereJava is able to express one out of four benchmarks adequately due to its *conditional method* feature. The underlying reason for the failure to express the benchmark problems is because the type languages of the proposals are not designed as programming languages and have numerous implicit and explicit restrictions on what kind of type constraining predicates can be expressed, and where type variables are allowed to be used. These restrictions stem from theoretical concerns such as decidability of typing, and also from implementation related concerns such as performance of the typing algorithm and the implementation in terms of the Java virtual machine.

In comparison to Gofer, the situation is clearly worse, as is indicated by the benchmark results: Gofer does at least as good for any of the benchmark problems, as any parametric types proposal. The most challenging of the lot is benchmark 4 which requires counting with numbers and manipulating lists of arbitrary length. Even though Gofer does not have either type numerals nor recursively defined type lists, it was possible to overcome this problem and simulate them by making explicit use of the implicit computational power offered by Gofer's qualified types. Existing Java parametric-types proposals come nowhere near this.

Chapter 4

Type-Oriented Meta Programming

4.1 Introduction

One of the central research hypotheses (hypothesis 1) of this dissertation states that there is unused and mostly unexplored potential in actively using static type information. Some examples in the introduction (section 1.6) already hinted at some possible applications of "active type manipulation". Programs expressed in a type programming language, may actively use static type information. These type "meta" programs are executed not as part of the actual "base" program, but instead are run as part of the compilation and type checking process, at compile time, with respect to the base language program. We call this programming technique "type-oriented meta programming". The goal of this chapter is to define exactly what we mean by type-oriented meta programming.

Many different languages and systems are referred to in the literature as meta-programming systems, meta languages or meta systems [MN87]. To avoid confusion we want to clarify what we understand under the terms *meta programming*, *meta program*, *base program*, etc. Therefore this chapter starts by introducing some terminology with respect to meta programming. After introducing the necessary terminology we give a concise definition of "type-oriented meta programming".

4.2 Meta-programming terminology

A program is a specification of a computational system that manipulates representations of entities from some "universe of discourse". For example, an address book application is a computational system that reasons about a universe of discourse which contains persons, names, phone numbers, addresses, etc. Consequently a computational system implementing an address book application manipulates representations for persons, names, etc.

Definition 1 *A* program *is a formal, executable specification of a computational system.*

Definition 2 *The* universe of discourse *of a program is the collection of concepts and entities which the computational system specified by the program can reason about.*

The program is expressed in a formalism that can be interpreted automatically in order to obtain the computational system it specifies. This formalism is called a "programming language".

Definition 3 *A* programming language *is a formalism that can be interpreted in an automatic manner in order to obtain the computational system specified by a program written in it.*

Programs can be constructed to reason about almost anything imaginable. It just boils down to defining representations of the entities or concepts one wants the program to reason about in terms of the data structures that are built into the programming language. Consequently, programs can be constructed that reason about other programs. Examples of such programs are compilers, type checkers, interpreters, code generators, etc.

Definition 4 *A program, the universe of discourse of which contains programs, is called a* meta program *or a* meta-level program*. The programs in the universe of discourse are called* base programs *or* base-level programs

In principle one can write meta programs in any general-purpose programming language. This involves defining and implementing representations of the programs in the universe of discourse in terms of the data structures offered by the general programming language. Apart from general programming languages there are also domain-specific programming languages that are tuned for specific application domains, i.e. with a bias towards certain types of universes of discourse.

Definition 5 *A* meta language *is a domain-specific programming language specifically tuned for specifying meta programs.*

In the sense of the above definition, core TyRuBa (see chapter 6) is a meta language for Java because it has specific features that facilitate manipulating pieces of Java programs. Typically meta languages have native data structures specifically intended for representing parts of programs or information about programs in the universe of discourse. Usually (but not necessarily) these data structures are tuned for representing information about programs written in a specific language called the *base language*.

Definition 6 *The* base language *for a given meta language is the language (if any) for which the meta language is specifically tuned.*

Not all objects in the universe of discourse need to be represented explicitly by objects in the computational system (specified by the program). Whenever an object or concept has an explicit representation we say that it is *reified*.

Definition 7 *An entity or concept in the universe of discourse that has an explicit representation in the computational system represented by the program is said to be* reified *by the program.*

4.3 Reflection

Even though we have explicitly chosen to avoid reflection and the complications that come with it we want to say a few words about reflection, if only to emphasize that we are *not* proposing to use reflection.

Given the definitions above we can say that a reflective program is a special kind of meta program the universe of discourse of which contains (aspects of) its own computational system. Following the treatment of [Smi82] a reflective system has a "causally connected self representation". This means that a program has access to some kind of data structure which represents (reifies) its computational system or aspects thereof. This can be inspected or it can be acted upon. "Causally connected" means that acting upon the self representation directly affects the computational system (this is sometimes called absorption). For a more detailed explanation of this terminology and theory we refer to [Smi82, Ste94b].

The self-referential nature of reflective systems makes them inherently very complicated both theoretically and with respect to implementation. Issues such as reflective overlap, meta-stability, infinite towers etc. need to be considered [Smi82, Smi84, Mae87, WF88, KdRB91, Ste94b, DVS95]. The complications with reflection mainly have one common cause: its self-referential nature creates confusion between what is "meta" and what is "base". Sometimes what is "meta" can be "base" at the same time and vice versa. A "simple" meta system does not have this problem and has a clean separation of meta level and base level thus avoiding the theoretical and practical problems related to reflection.

4.4 Type-Oriented Meta Programming

Finally we have enough terminology to state exactly what we mean by "type-oriented meta programming". As the phrase says, it is a meta-programming technique, this means that it is used in meta programs that reason about base programs. The wording "type-oriented" signifies that type information about base-level programs is reified and manipulated explicitly by the meta program.

Definition 8 Type-oriented meta programming *is a meta-programming technique relying on reified type information about base-level programs.*

4.5 Summary

In this chapter we have established some terminology and stated concisely and precisely what we mean by "type-oriented meta programming".

Chapter 5

Logic Meta Programming

5.1 Introduction

In this chapter we present the idea of Logic Meta Programming independently of the choice of a particular base language. It could in principle be applied to any programming language or even to any other structured form of textual data such as for example **`html`** [Gra97] or LaTeX [1].

The main idea of logic meta programming is describing base-language programs by means of logic programs. The central concept around which everything revolves is a mapping which associates every base-language program with a set of logic propositions which describe its properties in sufficient detail to be called a "representation" of it. A logic program is nothing more than a sophisticated way to specify a set of logic propositions. Therefore it follows that logic programs can be used to specify base language programs indirectly. This situation is depicted in figure 5.1.

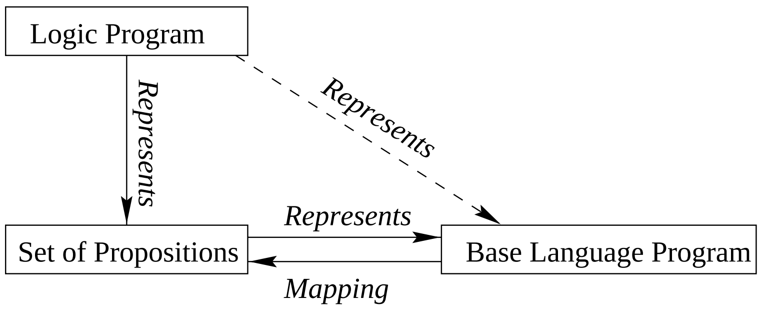

Figure 5.1: The role of the Mapping

This chapter outlines the typical architecture of a logic meta programming system based on this idea and explains what is expected of the representational mapping with respect to the chosen base language. The choice of a representational mapping scheme is an important design decision since it determines how useful and how well suited the resulting system will be for a particular purpose.

[1] An example of this is figure 9.7. The LaTeX source code to produce this figure was generated with the help of our logic meta programming system.

In the next chapter we will apply the ideas presented in this chapter to a particular base-language: Java. This will allow us to build a system, TyRuBa. The TyRuBa system will be used as medium for experimentation in the rest of this dissertation. TyRuBa will allow us to provide solid arguments to support our thesis. It will allow us to reify static type information about Java base-language programs and achieve type-oriented logic meta programming. Thus we will be able to illustrate the immense and currently unharvested potential of active type manipulation.

5.2 Base Programs as Sets of Propositions

The central concept around which everything revolves is a mapping of base-language programs onto sets of logic propositions that represent them. If the representation scheme is well chosen, the possibilities opened up can be immense. A logic programming language is just an expressive way to denote sets of logic propositions so we can then use the full power of a logic programming language to describe the structure of base-language programs indirectly represented as logic propositions.

Definition 9 *The symbol* $\mathbb{P}$ *denotes the* set of all base language programs*.*

Definition 10 *The symbol* $\mathbb{S}$ *denotes the* set of all sets of logic propositions*.*

Definition 11 *A* mapping *is a function* $M : \mathbb{P} \rightarrow \mathbb{S}$ *that associates a set of logic propositions* $S = M(P)$ *with any base language program* P*.*

Not just any function M mapping base programs onto sets of propositions can rightfully be called a *representational mapping.* The fact that we call a set S of logic propositions a *representation* of a base-language program implies that the base-language program can be reconstructed from the set of propositions. However the representation does not have to be syntactically complete in the sense that the reconstruction of the base-language program may be syntactically different from the original program as long as both base-language programs have the same semantics.

Definition 12 *A representation scheme* M *mapping base language programs onto sets of logic propositions is called* syntactically complete *iff* $\forall P, P' \in \mathbb{P} : M(P) = M(P') \Rightarrow P = P'$*. Where* $=$ *denotes the relation "... is syntactically identical to ...".*

Definition 13 *A representation scheme* M *mapping base language programs onto sets of logic propositions is called* semantically complete *iff* $\forall P, P' \in \mathbb{P} : M(P) = M(P') \Rightarrow P \equiv P'$*. Where* $\equiv$ *denotes the relation "... is semantically equivalent with ...".*

Note that any syntactically complete mapping is automatically also semantically complete since it is trivially true that programs with identical syntax also have the same semantics.

Definition 14 *A mapping* M *is called a* representational mapping *if* M *is semantically complete.*

Definition 14 implies that a representational mapping need not necessarily be syntactically complete but is free to "ignore" syntactic properties of a program as long as this does not alter the program's semantics. It is for example allowed to ignore comments.

5.3 Choosing the Representational Mapping

The choice of the representational mapping is an important part of the system's design since this mapping defines what aspects of the base-level program are made explicit and can be reasoned about by the meta program. The representational mapping should at least be semantically complete. It is not desirable that it should be syntactically complete. To see that this is the case consider a trivial mapping which represents any base-language program simply with a single proposition of the form:

```
program("...program text...").
```

Clearly this mapping is syntactically complete since it contains the entire text of the program as a string. It is also very obvious that this representation is at the same time almost completely useless for any purpose. It only regards a program as a string and consequently does not reveal anything interesting about it. With this mapping "meta programs" merely amount to string manipulations.

A useful mapping describes the structure of the program in more detail. Preferably the mapping is based on the base language's semantics rather than its syntax since programmers are more interested in reasoning about the semantics of their program than about the syntax of their programs. An aspect which is not relevant for the intended semantics of a program should therefore not be represented if this can be avoided. For example, in a `C` program the order in which functions are declared is not relevant for the meaning of the program. Therefore it is not a good choice to represent a `C` program with a *list* of function declarations. A list is a kind of collection of elements in which the order of the elements is considered relevant. Thus with a list representation, programs with distinct orderings of function declarations will have distinct representations. It is better to represent the `C` program as a *set* of individual assertions, one assertion per function declaration. This is closer to the semantics of the `C` program because the order of elements in a set is considered irrelevant.

With respect to reflecting the program's semantics rather than its syntax as much as possible, the trivial mapping is the worst possible choice. It only depends on the syntax of the program and reveals nothing more than the text of the program itself.

If at some point the syntax of a program or part of it is to be represented, this is best done in a structured hierarchical form, modeling the parse tree of the program rather than as a string. Also in this respect, the trivial mapping is the worst possible choice. It represents a program syntactically but doesn't even reveal the syntactic structure of the program.

The mapping also depends on the aspects we are interested in for the particular application we have in mind. In this dissertation we are interested in "type-oriented programming", therefore it will be important to make types and type relations explicit in the representation of the base-language program.

5.4 Architecture of a Logic Meta-Programming System

The make up of a logic meta programming system is depicted in figure 5.2. The heart of the system is a logic inference engine and database which stores the logic meta program. A logic program can be regarded as representing a set of deducible propositions. A proposition is in this set if and only if it can be derived from the facts and rules in the logic program. This virtual set of logic propositions can be consulted by *querying* the logic system. The interface of

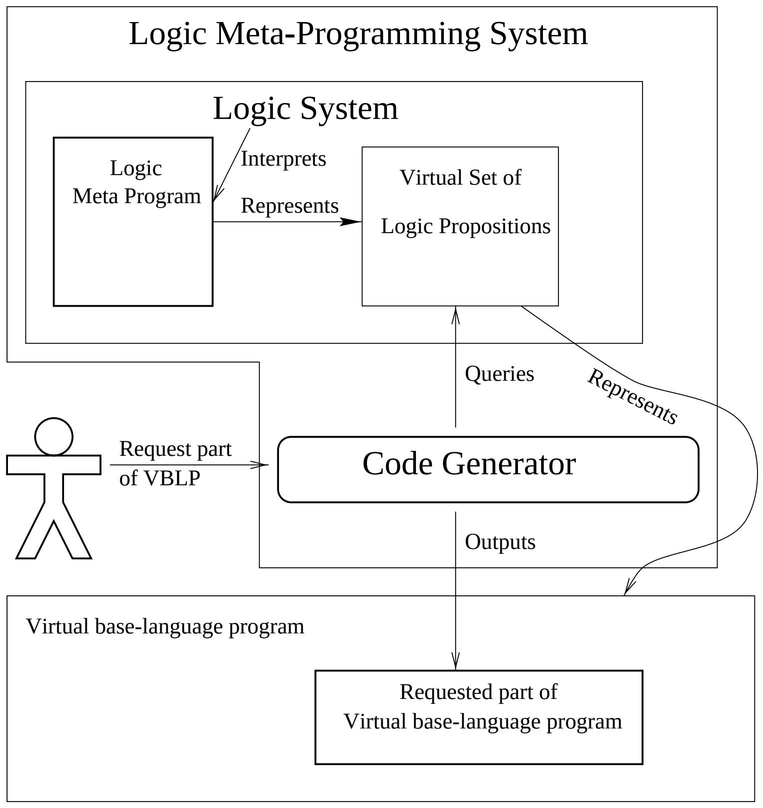
Logic Meta-Programming System
Logic System
Logic Meta Program
Interprets
Represents
Virtual Set of Logic Propositions
Queries
Represents
Request part of VBLP
Code Generator
Outputs
Virtual base-language program
Requested part of Virtual base-language program

Figure 5.2: Architecture of Logic Meta-Programming System

the system with its "user" is the *code generator* which outputs base language code. The code generator roughly corresponds to the inverse of the representational mapping. The following section elaborates on the exact relationship between the code generator and the mapping.

5.4.1 The Code Generator

The user asks the code generator to extract parts of the "virtual" base-language program represented by the meta program. The logic database contains all the necessary information to construct the base program so the code generator need only "ask" for it by making the right queries and then outputting the retrieved information in a suitable form in the syntax of the base language. This will become clearer when we discuss the concrete system TyRuBa for base-language Java in chapter 6.

The rationale behind the fact that specific parts of the virtual base-language program are requested from the code generator rather than having a code generator that outputs the entire program in one go is that the virtual program might possibly be infinitely large. This is due to the power of the logic paradigm which is able to express infinite sets of propositions. Suppose a `C` program is represented as a set of propositions asserting the presence of function declarations in the source file. It would be possible to declare an infinitely large program with infinitely many function declarations in it. However, trying to compile such a program by feeding it directly to an ordinary `C` compiler is impossible. The program would take an infinite time to compile. This does not necessarily mean that the program is useless altogether. Maybe it constitutes a library of an infinite number of (related) functions. It is perfectly reasonable to extract only the finite number of functions from the library that are actually called in the body of the main program using the library. So even though the virtual program might be infinite, it makes sense to be able to extract specific parts of its source code.

Disregarding the fact that the base program is accessible in parts let us now investigate more closely the properties of the "reverse" mapping implemented by the code generator.

Definition 15 *A* code generator *is a partial function $G : \mathbb{S} \rightarrow \mathbb{P}$ mapping sets of propositions onto base language programs.*

Note that a code generator is a *partial* function. That is, it does not necessarily define a valid base-language program for every possible set of propositions. This corresponds to the fact that some sets of propositions do not represent base-language programs.

A meaningful code generator is related to the representational mapping. The minimum requirement for the code generator is to be *consistent* with the representational mapping.

Definition 16 *A code generator G is called* consistent *with a representational mapping M iff $\forall P \in \mathbb{P}, \forall S \in \mathbb{S} : G(S) = P \Rightarrow M(P) = S$.*

Note also that the representational mapping is not an injective function so there might be two syntactically distinct programs P and P' which have the same representation $S = M(P) = M(P')$. Hence it is nonsense to require that $M(P) = S \Rightarrow G(S) = P$. Clearly this condition cannot be satisfied when two distinct programs P and P' exist for which $S = M(P) = M(P')$. Therefore the condition can only be satisfied for a syntactically complete representational mapping. However we already explained that syntactic completeness is usually not a desirable property.

Consistency with the representational mapping is a minimum requirement for a code generator to be meaningful. Therefore, from now on whenever we speak of a code generator we assume it to be consistent with the representational mapping. Note however that it is not specified what a code generator should do with a set of propositions that do not represent a base language program, i.e. a set S for which there is no P such that $M(P) = S$. This choice is left to the designer or implementor of the meta-programming system. It is also not completely specified what the code generator should return when several base level programs have the same representation. In principle any of the programs may be returned by the code generator in this case.

5.5 Summary

In this chapter we defined the concept of Logic Meta Programming and some related terminology. The central concept around which everything revolves is a *representational mapping* which defines how base level programs are represented as sets of logic propositions. We gave a schematic overview of the typical architecture of a logic meta programming system. This consists out of a logic inference engine and data base containing the meta program. The interface to the user is the *code generator* which is able to query the logic database and interpret the results in order to generate base-level code *consistent* with the *representational mapping*. An important aspect of the design of a logic meta-programming system is the choice of the representational mapping. The mapping is required to be *semantically* complete but *syntactic* completeness is not required. In fact, it is desirable to model the mapping after the semantics of base-language programs and explicitly ignore syntactic aspects of the program that do not affect the semantics.

Chapter 6

TyRuBa

6.1 Introduction

In this chapter we apply the principle of logic meta programming as outlined in chapter 5 with Java as base language. The system we present in this chapter is called TyRuBa which is an acronym for "Type Rule Base". The name is mainly historic as the core TyRuBa system has no specific features that make it focus on types. The system presented in this dissertation is a general system that lends itself to logic meta programming in general and is not preoccupied with types. The core of TyRuBa is a Prolog-like [SS94, CM81] language with a quoting mechanism that allows pieces of Java code to be treated as terms in logic programs. Similarly an unquoting mechanism allows logic terms to appear as part of these quoted pieces of Java code. The TyRuBa core system described thus far is not yet a logic meta-programming system as defined in chapter 5. It is just a particular variation of Prolog that is meant to allow more or less easy manipulation of Java source code fragments. To achieve logic meta-programming we will specify a representational mapping and add a consistent code generator to the system. The choice of the representational mapping and a corresponding code generator are important parts of the system design. They are also the most difficult part of the design and it is not even sure that there can be such a thing as "the best" representational mapping. What is "best" depends on what the system will be used for. Logic meta programming still being a very experimental technique it would be a bad idea to hardcode a highly specific representational mapping and code generator into the core TyRuBa system. We therefore chose a general mapping much like the trivial mapping given as an example in chapter 5. It was explained that this mapping is not very useful by itself, however, it will allow us to implement a more specific code generator in TyRuBa itself as an extra layer of rules that specify how the more specific representational mapping relates to the general mapping. This allows easy experimentation with different representational mappings and code generators, simply by providing different initialization files.

The rest of this chapter presents the core TyRuBa language and the general representational mapping that is hardcoded into the core TyRuBa code generator. At the end of the chapter we show an example initialization file that illustrates the implementation of a finer-grained code generator on top of the hardcoded code generator.

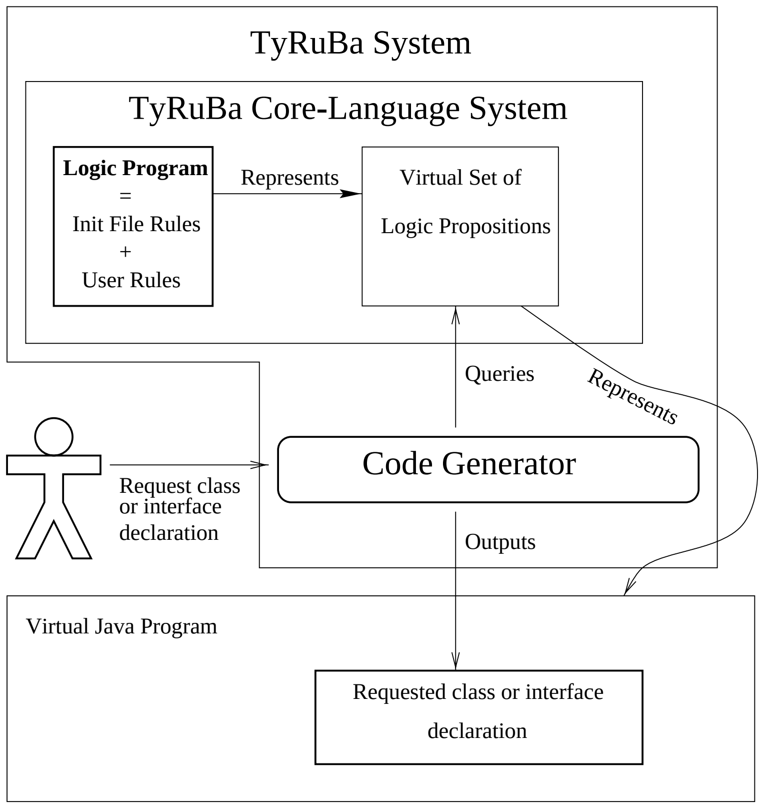
TyRuBa System
TyRuBa Core-Language System
Logic Program = Init File Rules + User Rules
Represents
Virtual Set of Logic Propositions
Queries
Represents
Code Generator
Request class or interface declaration
Outputs
Virtual Java Program
Requested class or interface declaration

Figure 6.1: Architecture of TyRuBa

6.2 History of TyRuBa

Before presenting the actual system, we first give some background information about the history of the TyRuBa system. This will help the reader in putting the system and its design into its proper context. The system certainly has many points upon which it can be criticized. However, it must be kept in mind that it was only designed and implemented as a medium for experimentation. The main purpose of it is to enable the exploration of the technique of type-oriented logic meta programming, and thus to provide evidence that this technique has considerable potential. Indirectly this evidence also supports our thesis, because the potential of type systems which sacrifice non-ambiguity and decidability of typing for expressiveness lies in the potential offered by a type-language which itself is designed as a true programming language. Such a type language opens up the possibility to do type-oriented meta-programming.

We wanted to do all this with minimal effort and come to the point of experimentation and validation as soon as possible. Therefore, all was sacrificed in order to obtain more expressive power for the type-manipulation language. Other aspects, including type-checking and syntactic issues were ignored as much as possible. A considerable research effort and implementation work will therefore still be needed in order to obtain a system which can truly be used in practice. A more elaborate sketch of TyRuBa's history, and some speculations about how it can be made more usable in practice will be given in chapter 11. We now only present a general discussion.

Originally it was our intent to create something which looks more like Java/Pizza, and which hides the logic system from the user under a layer of surface syntax. The first version of TyRuBa therefore looked much more like a traditional language and tried to "cover up" the underlying logic engine under some Java like syntactic extensions. As a consequence the system also had a very specific representational mapping hardcoded into it. The syntactic layer itself and the hardcoded mapping soon proved to be major obstructions. It was often felt that the syntactic layer provided too narrow an interface to the underlying logic language and thus was an obstruction in using the full potential of logic meta programming. The fixed representational mapping also was a considerable restriction. It soon became apparent that the chosen mapping was too coarse grained. Since it had been hardwired into the system's implementation and into the design of the surface syntax it was not easy to change or refine the mapping. Eventually it was decided to, at least temporarily, abandon the idea of a surface syntax and directly express meta programs in the logic language itself. The resulting system is presented in the remainder of this chapter.

6.3 The TyRuBa Language

Basically the TyRuBa language is a simplified Prolog with a few modifications that allow pieces of quoted Java code to occur as terms in TyRuBa programs. We will now take a closer look at the structure and the different elements of a TyRuBa program and give some examples illustrating their use. A summary of the TyRuBa grammar in EBNF notation is given in figure 6.10 at the end of this chapter.

6.3.1 Programs

A TyRuBa program is a list of interspersed logic rules, facts, queries and directives.

```
TyRuBaFile       ::= ( Rule | Fact | Query | IncludeDirective |
                       GenerateDirective | VerbatimDirective )*
```

A typical example of a TyRuBa "main program" file is displayed in figure 6.2. This particular example only consists of directives. The actual rules and facts of the logic program are included from another file called `"Array.rub"`.

```
#verbatim {package aRuBa.tyRuBa.examples;}

#include "Array.rub"

#generate Array<String>

#verbatim {

public class ArrayTest {

  static final int size = 3;

  public static void main(String[] args) {
    Array<String> box = new Array<String>(size);
    for (int i=0;i<size;i++) {
      box.setElementAt("("+i+")",i);
    }
    for (int i=0;i<size;i++) {
      System.out.print(box.elementAt(i));
    }
    System.out.println();
  }
}

}
```

Figure 6.2: Example "main" file

The rules in the program are stored in a database. Note that when queries are executed, the rules in the database will be considered in the opposite order from that in which they occur in the TyRuBa source file. We have established this order of precedence for rules because it conforms to the intuitive feeling that more recent rules are more important.

6.3.2 Rules, Facts and Queries

```
Query            ::= ":-" Expression "."
```

The queries (or "goals") are not really part of the TyRuBa program since they will not affect the generated Java code in any way. When a query is executed its solutions are dumped onto the standard error device. This is useful mainly for debugging purposes.

```
Rule             ::= Predicate ":-" Expression "."
Fact             ::= Predicate "."
```

Rules consist of a *conclusion* term followed by a "`:-`" followed by a *condition* expression. If a rule's condition is always true, then it may be omitted. Such a rule is called a fact. The semantics and syntax of rules and facts is the same as in Prolog so we will not elaborate any further on them.

6.3.3 Directives

Directives instruct the TyRuBa system to perform a special action.

```
IncludeDirective  ::= "#include" <STRING_LITERAL>
VerbatimDirective ::= "#verbatim" Term
GenerateDirective ::= "#generate" Term
```

An `include` directive works like an include directive in `C` [KR88]. The string literal is interpreted as a file name and the contents of the corresponding file is read and substituted for the include directive.

The `generate` and `verbatim` directives are related to Java code generation and will be discussed in section 6.4.

6.3.4 Terms

```
Term              ::= Variable | Constant | CompoundTerm
                    | QuotedCode | List
```

There are some superficial syntactic differences with Prolog for variables, constants and compound terms. There is also a special *QuotedCode* term for including pieces of Java as data in logic programs. We will now discuss the different kinds of TyRuBa terms one by one.

6.3.5 Variables and Constants

The lexical rules for variables and constants in TyRuBa are different from Prolog. In Prolog variables are identifiers that start with a capital letter. Identifiers starting with a lower case letter are constants. This would be confusing in TyRuBa because of the mix with Java. Java is case sensitive and allows variables and names of classes and interfaces etc. to start with upper case as well as with lower case letters. In order to avoid ambiguity TyRuBa therefore lexically distinguishes variables from constants by a leading "?" rather than by an uppercase letter. Some examples of variables and constants are listed in figure 6.3.

Variables	`?x`, `?Foo`, `?bar23`, `?code`
Constants	`x`, `X`, `Foo55`, `1`, `1234`

Figure 6.3: Some examples of TyRuBa variables and constants

6.3.6 Compound Terms

```
CompoundTerm    ::= Constant "<" TermList ">"
```

Compound terms in TyRuBa are written with "`<` and "`>`" rather than with "`(`" and "`)`" as in Prolog. This avoids confusion with function calls when terms appear as part of quoted Java code. It also has the advantage of making the syntax of predicates and terms distinct from one another. In regular Prolog these two are written in exactly the same manner even though they have different semantic roles[1]. Some examples of TyRuBa compound terms are:

```
Array<String>
Bar<Foo,1>
triplet<?first,?second,?third>
Deeply<Nested<compound<term,?x>>>
```

6.3.7 Lists

```
List            ::= "[" ListRest
ListRest        ::= "]"
                  |  Term ListCdr
ListCdr         ::= "," Term ListCdr
                  | "|" Term "]"
                  | "]"
```

Lists and pairs in the standard Prolog notation are supported in TyRuBa. A `[]` denotes the empty list and the notation `[X | Y]` denotes a pair. The pair `[X | Y]` is semantically equivalent to (and internally represented by) a compound term with a special qualifier as follows: `_pair_(X,Y)`. The notation `[a,b,c]` as shorthand for `[a | [b | [c | [] ] ] ]` is also supported. Since this is exactly as in regular Prolog we need not elaborate on this any further.

6.3.8 Quoted Java Code

TyRuBa provides a special kind of compound term that represents a piece of "quoted" Java code.

```
QuotedCode      ::= "{" (QuotedElement)* "}"
QuotedElement   ::= Variable
                  | CompoundTerm
                  | "{" (QuotedElement)* "}"
                  | "@" Term
                  | JavaTokenNotBraces
JavaTokenNotBraces ::= Identifier
                     | IntegerLiteral
                     | StringLiteral
                     | ...
                     | "," | "(" | ")" | ";"
                     | ...
```

[1]In Prolog this may create confusion for the programmer but also allows meta-circular code that treats Prolog expressions as data (i.e. terms). We thought this to be of minor importance in the context of this dissertation since we explicitly decided to avoid the theoretical and practical complications of reflection and meta-circularity (section 4.3).

In the version of TyRuBa presented and used in this text the quoting mechanism is very rudimentary. Basically a quoted Java term is nothing more than a kind of string starting after a "{" and ending just before the next balanced "}". Unlike a string it is not composed of characters but of Java tokens, variables, constant terms and compound terms. The Java tokens are treated as special name constants whose name is equal to the printed representation of the token. The following example of a quoted Java term illustrates all kinds of quoted elements:

```
{ void foo() {
    Array<?El> contents  = new Array<?El>[5];
    ?El        anElement = contents.elementAt(1);
  }
}
```

In the above example we see a compound term "`Array<?El>`". We find several name constants "`contents`", "`new`", "`anElement`", ... There are two integer literals `5` and `1`. The remainder of the tokens such as "`=`", "`.`" and "`(`" are Java tokens treated as name constants with "strange names". Note that a quoted code block may contain "`{`" and "`}`" tokens, as long as these are properly balanced. Let it be clear that a nested "`{`" or "`}`" is treated just as any other token and does *not* introduce a nested quoted code block.

The meaning of a quoted Java term in the context of a TyRuBa program is derived directly from its internal representation. A quoted Java term corresponds to a compound term of arity 1 with a special qualifier. As its subterm it contains a TyRuBa list of quoted elements. The example given above is internally represented by the following compound term[2]:

```
_{}_([void, foo, (, ), {, Array<?El>, contents, =, ... ])
```

The "`@`" is an "unquoting" symbol and may be used to explicitly include any TyRuBa term as part of the quoted code. Variables and compound terms may be inserted without an explicit unquote symbol, they are unquoted implicitly. The implicit unquoting of variables and compound terms mostly makes the explicit unquoting with "`@`" obsolete. However it is left in the language for unquoting lists.

6.3.9 Expressions

```
Expression      ::= Disjunction
Disjunction     ::= Conjunction ( ";" Conjunction )*
Conjunction     ::= SimpleExpression ("," SimpleExpression)*
SimpleExpression::= Predicate | Arithmetic
                  | "(" Expression ")"
```

Expressions in TyRuBa are similar to regular Prolog expressions. Conjunction (and) is expressed with a "`,`" and disjunction (or) with a "`;`".

[2]This is not actually correct TyRuBa syntax but is given only as an illustration.

6.3.10 Predicates

```
Predicate       ::= Constant [ "(" TermList ")" ]
```

Predicates in TyRuBa are the same as in Prolog. A predicate is the simplest type of expression from which more complicated expressions are composed by means of disjunctions and conjunctions. Evaluating a predicate is just as in Prolog. Unify with the head of a rule to determine whether it is applicable and then evaluate the body of the applicable rule. We will not explain this process in detail since it is pretty much the same as in Prolog. We refer to [CM81, SS94, Llo88, Roy94, ASS96] for elaborate background material about implementation-related and theoretical issues regarding Prolog-like languages and their evaluation.

TyRuBa offers a number of "meta" predicates[3] which as a convention have qualifier names written entirely with capitals. Each of these special predicates have their own method of evaluation and can be used to control various aspects of the evaluation and backtracking process of the system. Some of these can be found in regular Prolog also. We will now briefly explain these special predicates one by one. In order to clarify the explanations we will give a few simple examples which are evaluated with the "facts" shown in figure 6.4 present in the database of the TyRuBa system.

```
person(Kim,Mens).
person(Tom,Mens).
person(Wolfgang,DeMeuter).
person(Theo,DHondt).
mathematician(Kim,Mens).
mathematician(Tom,Mens).
```

Figure 6.4: A Simple TyRuBa Database

The FIRST meta predicate

FIRST(*expression*)

The *expression* is evaluated and only the first solution is retained, while all the remaining solutions are discarded. This corresponds roughly[4] to a "cut" in Prolog.

```
:- person(?fn,?ln).
#SOLUTION : person(Theo,DHondt)
#SOLUTION : person(Wolfgang,DeMeuter)
#SOLUTION : person(Tom,Mens)
#SOLUTION : person(Kim,Mens)

:- FIRST(person(?fn,?ln))
#SOLUTION : FIRST(person(Theo,DHondt))
```

Note the order of the solutions. It is exactly the reverse of the order in which the facts were listed. This of course is because of the reverse order in which TyRuBa considers rules and facts.

[3]These are called "meta" predicates because they fall outside of first order logic and are "about" terms or expressions.

[4]In Prolog we could define FIRST using a cut as follows: `first(E) :- call(E), !.`

The `FINDALL` predicate

`FINDALL(`*`expression, term, variable`*`)`

This is exactly the same as the standard Prolog `findall` meta predicate. It will find all instances of the *term* for which the *expression* is true and collect all of these instances into a list that will be bound to the *variable*.

```
:- FINDALL(person(?fn,?ln),?ln,?lns)
#SOLUTION : FINDALL(person(?fn,?ln),?ln,[ DHondt, DeMeuter, Mens, Mens ])
```

The `NODUP` predicate

In the context of code generation it is often useful or needed to eliminate duplicate or partially duplicate solutions to a query. Examples of using `NODUP` in the implementation of code generators can be found throughout the remainder of this dissertation. In section 7.2.2, it is used to eliminate duplicate names from a generated `extends` or `implements` clause for example.

`NODUP(`*`term, expression`*`)`

Execution of the above NODUP predicate is the same as evaluating the expression and eliminating from its solution all "duplicates". A solution is considered to be a "duplicate" solution if the instantiation of the term relative to this solution is identical to an instantiation of the same term relative to an earlier solution. The following example uses `NODUP` to remove duplicate lastnames.

```
:- NODUP(?ln,person(?fn,?ln))
#SOLUTION : NODUP(DHondt,person(Theo,DHondt))
#SOLUTION : NODUP(DeMeuter,person(Wolfgang,DeMeuter))
#SOLUTION : NODUP(Mens,person(Tom,Mens))
```

Again, the reversed order in which rules are considered explains why Tom Mens is preferred over Kim Mens by the TyRuBa system.

A word of caution is in order here. The `NODUP` predicate does not function properly when the filtered solutions still contain unbound variables. This not really a problem for its intended use: to filter duplicates from pieces of generated code. Unbound variables are not supposed to occur in generated code anyway (see section 6.4.3).

The `NOT` predicate

`NOT(`*`expression`*`)`

The NOT predicate fails whenever its argument *expression* succeeds (i.e. has at least one solution) under the current context of variable bindings.

```
:- person(?fn,?ln),NOT(mathematician(?fn,?ln))
#SOLUTION : person(Theo,DHondt),NOT(mathematician(Theo,DHondt))
#SOLUTION : person(Wolfgang,DeMeuter),NOT(mathematician(Wolfgang,DeMeuter))
```

The `BOUND` predicate

`BOUND(`*`variable`*`)`

The BOUND predicate verifies whether *`variable`* is bound. It fails otherwise.

Arithmetic

TyRuBa offers some predicates to do simple integer arithmetic and comparison. They are summarized in figure 6.5.

TyRuBa expression	Mathematical relationship
`+(?a,?b,?s)`	$s = a + b$
`*(?a,?b,?p)`	$p = ab$
`<(?a,?b)`	$a < b$

Figure 6.5: Special predicates for integer arithmetic and comparison

The predicates "`+`" and "`*`" for simple integer arithmetic only work if at least two of the three arguments have integer values at the moment the predicate is evaluated. If the third one is also bound it will be verified whether the value is correct. In case the third one is still unbound, it will be bound to the correct value.

The comparison predicate "`<`" functions as a filter and only works when both arguments are bound to integers at the moment the predicate is evaluated. In this case it is verified whether the first argument is smaller than the second and the "`<`" predicate succeeds or fails accordingly.

Note that a comparison for equality is not strictly needed since this can be done by means of the implicit equality test in the unification algorithm when a variable occurs more than once in a rule or query. The following fact that defines an `equal` predicate is provided as part of one of the TyRuBa initialization files.

```
equal(?x,?x).
```

6.4 Code Generation

What we have described in the preceding sections, is basically just a simplified Prolog system with some superficial syntactic differences and with an extension for including quoted pieces of Java code. This does not yet constitute a meta-programming system for Java as outlined in chapter 5. We still need a *code generator* that is consistent with some representational mapping. We are going to introduce a coarse-grained general-purpose representational mapping and a consistent code generator in this section. This code generator is hardcoded in the TyRuBa system. However, we can experiment with more specific and more fine-grained representational mappings and code generators by implementing a code generator on top of the hardcoded code generator in TyRuBa itself. We will show one example of this in order to demonstrate how it can be done.

6.4.1 The Hardcoded Representational Mapping

When looking at Java programs, the most coarse-grained entities in them are class declarations and interface declarations. The hardcoded representational mapping does not split up these entities but just regards them as one large chunk of quoted Java code. The presence of a class declaration with name `X` in the base program (to be generated) is thus represented with a proposition:

```
generate(X,{class X extends ... implements ... {...}}).
```

Similarly for an interface declaration:

```
generate(X,{interface X extends ... {...}}).
```

The name of the predicate `generate` is chosen thus because the user usually implements it himself in some initialization file in order to specify a code generator for the representational mapping he wants to adopt.

According to the observations from chapter 5 the representational mapping presented here is not a very suitable one. The classes and interfaces are just treated as big string like entities. No semantic information is available. However it is not the intention to use this mapping directly. Hardcoding this mapping into the system merely establishes a convenient interface that allows the user to implement his own representational mapping on top of it. Given the experimental stage we are in, it would be a bad idea to hardcode a highly specific code generator into the system directly. Thus, the hardcoded, coarse-grained, general-purpose code generator allows easy experimentation with different representational mappings and code generators simply by swapping initialization files.

6.4.2 Directives for the code generator

The code generator is driven by directives in the TyRuBa source files. There are two directives which address the code generator, the `verbatim` and the `generate` directive. We now briefly discuss their function.

The `generate` directive

This directive invokes the code generator and requests it to generate the part of the source file (i.e. a class or an interface declaration) corresponding to the `name` given by its argument. The example in figure 6.2 uses the directive

```
#generate Array<String>
```

to instruct the code generator to dump the definition of the class `Array<String>` onto the output file.

All the code generator does is invoke the TyRuBa core system and evaluate a query:

```
:- generate(Array<String>,?code).
```

The solutions to the query will bind the "generated code" to the `?code` variable. It is possible that the meta program does not provide an entity with the specified name. In this case the query has no result and the code generator will report an error message.

It is also possible that the meta program is ambiguous and that the query has more than one solution. In this case the code generator will disregard all but the first solution. Knowing that rules are tried in the reverse order of which they were read, this can be used to obtain a primitive kind of "overriding" of older class or interface definitions with newer ones.

The `verbatim` directive

This is nothing but a shortcut to allow easy inclusion of some specific verbatim pieces of Java code. The argument of the verbatim directive is dumped directly onto the generated file. This shortcut falls out of the conceptual model of representational mapping and code generator but is nevertheless often useful. In figure 6.2 a `verbatim` directive is used twice. The first directive adds a `package` declaration to the start of the generated Java file. The second one adds a "main" class which constitutes a small test application which uses the generated `Array<String>` class.

6.4.3 The Meaning of Terms in Java Code

Consider the (hardcoded) representational mapping. If we take an existing Java program and look at the set of propositions which represent it we will find only propositions of the form

```
generate(?entityName,?quotedJavaTerm).
```

In place of the variable `?entityName` there will be a Java identifier and in place of `?quotedJavaTerm` there will be chunk of quoted Java code which does not contain any TyRuBa variables or compound terms, but only Java tokens. This is so because the pieces of quoted Java code correspond with class and interface declarations taken from the Java program. Since Java programs do not contain TyRuBa variables and compound terms neither do the quoted code blocks.

However, the code generator works exactly in the opposite direction than does the (theoretical) representational mapping. The code generator starts from a logic program, i.e. a "virtual" set of logic propositions and from this it generates Java code. The question therefore arises what meaning the code generator should assign to compound terms, variables, etc. when they occur in some of the propositions. The simplest idea is to simply forbid this kind of use and issue an error message or something like this. However, the system can be made considerably more flexible and expressive by giving some sensible meaning to compound terms, variables etc. We now explain the choices that were made in this respect.

Compound Terms

In the TyRuBa code generator, compound terms (which do not contain any unbound variables, either directly or nested inside one of their subterms) are treated as Java identifiers. Thus meta programs may use structured names for classes and interfaces. This facilitates, for example, the simulation of parametric classes and interfaces (section 7.3). All that is required to be able to use compound terms as special Java identifiers is a translation scheme which maps compound terms onto valid Java identifiers. The translation scheme in the TyRuBa code generator simply takes the compound term's textual representation and replaces all occurrences of "`<`", "`>`" and "`,`" by "`_L`", "`_R`" and "`_C`" respectively. Some examples of compound terms and what they look like in generated Java code are displayed in figure 6.6.

Variables

An error is generated when unbound variables occur in pieces of quoted Java code output by the code generator. The reason for this is that in general, it is hard or maybe even impossible to assign a reasonable substitute for a free variable in the generated code. It might be

TyRuBa term	In generated Java code
`Array<String>`	`Array_LString_R`
`triplet<1,2,3>`	`triplet_L1_C2_C3_R`
`deeply<nested<term>>`	`deeply_Lnested_Lterm_R_R`

Figure 6.6: Examples of compound terms and their Java "mangled" name.

reasonable to replace variables by the identifier `Object` in case the variable occurs in a place where a class name is expected. However this is just a particular case. If the variable occurs anywhere else it is far less obvious to find a reasonable substitute. To complicate things further the conception of quoted Java code as a sequence of tokens without any further syntactic or semantic information makes it impossible to determine the nature of the context in which a variable occurs[5].

List

Output generated for a list occurring in a quoted-code block is produced by generating output for each element in the list one by one with white space in between each element.

Quoted Code Blocks

When the code generator encounters a nested quoted code block it treats it in the same way that it treats a list. That is, generate output for each element of the quoted code block with white space in between. Note that the "`{}`" are delimiters indicating the start and end of a quoted code block and are not considered to be part of the quoted code block's contents so therefore they are not printed. To make this a little clearer consider the piece of TyRuBa code in figure 6.7. It makes a piece of quoted code with two other pieces of quoted code as elements. Note that the delimiting "`{}`" of the nested blocks are not printed. Note also that this is not the same as an append of the two blocks. The nesting structure is retained. Although not visible at first sight, the constructed code block is *not* a list of tokens but really a list of two quoted code blocks. When looking more closely, this is visible from the amount of white space printed in between tokens.

```
blockpair(?b1,?b2,{?b1 ?b2}).
:- blockpair({Just a},{list of tokens},?pair).
#SOLUTION : blockpair({ Just a },{ list of tokens },{  Just a   list of tokens  })
```

Figure 6.7: Make a quoted code block of two elements

6.5 Refining the Code-Generator

Since the hardcoded code generator is not very useful by itself, we will need to implement a more fine-grained code generator. We can do this by providing an initialization file which

[5]The latter complication might be solved by a more structured representation of pieces of quoted Java code. We could then replace variables by something suitable (e.g. `Object`) in some situations.

implements it on top of the hardcoded code generator. In this section we will illustrate how to do this by giving an example.

The code generator we implement here as an example is only a little more fine grained than the hardcoded code generator. Consequently this code generator is not very useful either. The example merely aims at illustrating the method of implementing a code generator in TyRuBa on top of the existing hardcoded code generator. We do not want to go into the details of a more complicated code generator just yet.

6.5.1 Representational mapping

The hardcoded mapping does not make a distinction between a class and an interface declaration. Both are treated simply as big chunks of quoted Java code associated with the name of the defined entity by means of the `generate` predicate. A simple "refinement" of this representational mapping makes a distinction between a class declaration and an interface declaration and stores class declarations and interface declarations under different predicates in the TyRuBa logic database. Additionally it also asserts separately whether the class or interface is public or final with a proposition of the form `public(entityName)` or `final(entityName)` respectively.

The remainder of the class or interface declaration, i.e. from the `extends` clause on, is put into a proposition `class` or `interface` predicate respectively. This representational mapping is illustrated in figure 6.8.

6.5.2 Code Generator

We can implement a code generator that is consistent with this representational mapping by means of an initialization file that provides definitions for the `generate` predicate from the hardcoded mapping in terms of the predicates `public`, `final`, `class` and `interface` as shown in figure 6.9.

6.6 Summary

In this chapter we presented the TyRuBa system, a concrete system that allows logic meta programming for Java. The system offers a core consisting of a simple Prolog-like logic language with an extension that allows pieces of Java code to be included as terms in logic programs. The syntax of TyRuBa is summarized in figure 6.10.

The core TyRuBa system has a large-grained general-purpose code generator hardcoded into it. The large-grained code generator is not very useful by itself but provides a convenient interface to "plug in" custom made code generators implemented in TyRuBa itself in an initialization file. In the next chapters we will make elaborate use of this and propose and implement more fine-grained mappings and code generators whenever the need for them arises.

```
public interface anInterface
extends ...
{ ... }

final class aFinalClass extends ...
implements ...
{  ... }

public class aPublicClass ...
class anotherClass ...
```

```
public(anInterface).
interface(anInterface,{
  extends ...
  { ... }
}).

final(aFinalClass)
class(aFinalClass,{
  extends ...
  implements ...
  {  ... }
}).

public(aPublicClass).
class(aPublicClass,{...}).

class(anotherClass,{...}).
```

Figure 6.8: A Java program (on top) and its representation.

```
generate(?class,{?public ?final class ?class ?rest}) :-
  gen_public(?class,?public),
  gen_final(?class,?final),
  class(?class,?rest).

generate(?itf,{?public ?final interface ?itf ?rest}) :-
  gen_public(?itf,?public),
  gen_final(?itf,?final),
  interface(?itf,?rest).

gen_public(?entity,{}) :- NOT(public(?entity)).
gen_public(?entity,{public}) :- public(?entity).

gen_final(?entity,{}) :- NOT(final(?entity)).
gen_final(?entity,{final}) :- final(?entity).
```

Figure 6.9: Sample implementation of a code generator in TyRuBa

```
TyRuBaFile      ::= ( Rule | Fact | Query | IncludeDirective |
                      GenerateDirective | VerbatimDirective )*
IncludeDirective::= "#include" <STRING_LITERAL>
VerbatimDirective::= "#verbatim" Term
GenerateDirective::= "#generate" Term
Rule            ::= Predicate ":-" Expression "."
Fact            ::= Predicate "."
Query           ::= ":-" Expression "."
Expression      ::= Disjunction
Disjunction     ::= Conjunction ( ";" Conjunction )*
Conjunction     ::= SimpleExpression ("," SimpleExpression)*
SimpleExpression::= Predicate | Arithmetic
                  | GreaterFilter | "(" Expression ")"
GreaterFilter   ::= "<" "(" Term "," Term ")"
Predicate       ::= Constant [ "(" TermList ")" ]
Term            ::= Variable | Constant | CompoundTerm
                  | QuotedCode | List
CompoundTerm    ::= Constant "<" TermList ">"
List            ::= "[" ListRest
ListRest        ::= "]"
                  |  Term ListCdr
ListCdr         ::= "," Term ListCdr
                  | "|" Term "]"
                  | "]"
Constant        ::= <IDENTIFIER>
                  | Literal
TermList        ::= [ Term ( "," Term )* ]
Variable        ::= <VARIABLE>
QuotedCode      ::= "{" (QuotedElement)* "}"
QuotedElement   ::= Variable
                  | CompoundTerm
                  | "{" (QuotedElement)* "}"
                  | "@" Term
                  | JavaTokenNotBraces
```

Figure 6.10: The Syntax of TyRuBa

Chapter 7

Type-Oriented Meta Programming in TyRuBa

7.1 Introduction

This chapter will show that TyRuBa can be used for type-oriented meta programming for Java. To accomplish this we propose a representational mapping that reifies base-level types and relationships between base-level types. A corresponding code generator consistent with this representational mapping is implemented by means of a suitable TyRuBa initialization file.

The mapping and code-generator in this chapter are "minimalistic" in the sense that we have kept them as coarse grained as possible while still allowing type-oriented programming. Classes and interfaces are reified as atomic units in the sense that the body of a class or interface declaration is treated as a big chunk of quoted Java code which is not split up into smaller elements. However, the extends and implements relationships between classes and interfaces *is* reified. Subsequent chapters will present applications of type-oriented meta programming in TyRuBa which typically require a more fine-grained view of the base-level program. We will introduce more fine-grained representational mappings and implement refinements of the code generator when the need arises.

At the end of this chapter we will give a tentative example that illustrates the potential power of type-oriented meta programming in TyRuBa. This example is specifically chosen because it makes full use of the fact that the logic meta language is an unrestricted, fully Turing-complete programming language.

7.2 Reifying Type Information

The first thing we need in order to accomplish *type-oriented* meta programming is to reify base-level type information into meta programs. This can be accomplished by means of a suitable representational mapping and corresponding code generator that makes types explicit. In this section we propose a representational mapping and code generator which do just this.

7.2.1 Representational Mapping

In a Java program a (user defined) type is either a class or an interface. The relationships between types are class extension, interface extension and interface implementation. For the time being this is all we are interested in, and we consider this to be the absolute minimum of reified information needed to do type-oriented programming for Java.

The presence of a class declaration in the base-level program is represented by means of a proposition of the form:

`class_(`*`className`*`,{`*`classBody`*`}).`

This "asserts" that there is a class declaration for the class ***className*** with a class declaration body given by ***classBody***. A similar proposition asserts the presence of an interface declaration:

`interface_(`*`interfaceName`*`,{`*`interfaceBody`*`}).`

Whenever an interface name appears in the implements clause of a class declaration this is asserted by a proposition of the form:

`implements_(`*`className`*`,`*`interfaceName`*`)`

Extends clauses are treated similarly. Whenever a type (class or interface) appears in the extends clause of another type this is asserted by a proposition of the form:

`extends_(`*`subTypeName`*`,`*`superTypeName`*`)`

Note that we have adopted the naming convention of ending predicate names with a "_" if they have an assigned meaning in this representational mapping. We will use different naming conventions for the refined representational mappings given in subsequent chapters thereby making it implicitly clear what representational mapping scheme is being used.

A schematic Java program and the corresponding set of logic propositions are given in figure 7.1.

7.2.2 Code Generator

We now show the implementation of a code generator that is consistent with the representational mapping given above. As we explained in section 6.5 we can implement this code generator in TyRuBa itself, on top of the hard-coded code generator. The entire implementation of the code generator is given in figure 7.2. The code generator is implemented as an extra layer on top of the hard-coded code generator by implementing the `generate` predicate in terms of the predicates of the representational mapping. Code for classes and interfaces are both generated in more or less the same way. We will take a closer look at one of them, for example the generation of code for a class:

```
generate(?class,{
class ?class
?extendsclause
?implementsclause
{ ?body }
}) :- class_(?class,?body),
      generate_extendsclause(?class,?extendsclause),
      generate_implementsclause(?class,?implementsclause).
```

The extends clause, the implements clause and the body of the class are computed by the condition of the above rule. The body of the class is simply looked up in the logic

```
interface interface1
extends superInterface1,superInterface2
{ ...A... }

interface interface2 { ...B... }

class aClass extends aSuperclass
implements interface1, interface2 {
  /** The body of aClass */
  ...C...
}
```

```
interface_(interface1,{...A...}).
extends_(interface1,superInterface1).
extends_(interface1,superInterface2).

interface_(interface2,{...B...}).

class_(aClass,{
  /** The body of aClass */
  ...C...}).
extends_(aClass,aSuperClass).
implements_(aClass,interface1).
implements_(aClass,interface2).
```

Figure 7.1: A Java Program (on top) and its representation (below)

```
generate(?class,{
class ?class
?extendsclause
?implementsclause
{ ?body }
}) :- class_(?class,?body),
      generate_extendsclause(?class,?extendsclause),
      generate_implementsclause(?class,?implementsclause).

generate(?itf,{
interface ?itf
?extendsclause
{ ?body
}
}) :- interface_(?itf,?body),
      generate_extendsclause(?itf,?extendsclause).

generate_extendsclause(?x,?extendsclause) :-
  FINDALL(NODUP(?extended,extends_(?x,?extended)),
          ?extended,?extendslist),
  JavaClause(extends,?extendslist,?extendsclause).

generate_implementsclause(?x,?implementsclause) :-
  FINDALL(NODUP(?itf,implements_(?x,?itf)),
          ?itf,?implementslist),
  JavaClause(implements,?implementslist,?implementsclause).

/***** Auxiliary predicates for constructing Java syntax */

/*Create an implements or extends clause (?symbol indicates which)
  Example 1:
   JavaClause(extends, [a,b,c], ?result)
   ==> JavaClause(extends,[a,b,c], {extends a,b,c})
  Example 2:
   JavaClause(extends, [], ?result)
   ==> JavaClause(extends, [], {})
*/
JavaClause(?symbol,[],{}).
JavaClause(?symbol,[?f|?r], {?symbol ?itflist})
 :- JavaCommaList([?f|?r],?itflist).

/*Construct a list of comma separated elements
  Example:
   JavaCommaList([a,b,c],?result)
   ==> JavaCommaList([a,b,c],{a,b,c})
*/
JavaCommaList([],{}).
JavaCommaList([?e1],{?e1}).
JavaCommaList([?e1,?e2 | ?er],{?e1,?cr}) :- JavaCommaList([?e2|?er],?cr).
```

Figure 7.2: A code-generator for type-oriented meta programming

database. The extends and implements clauses are computed by two auxiliary predicates, `generate_extendsclause` and `generate_implementsclause` respectively. These auxiliary predicates look up all type names that are in an `extends` or `implements` relationship with the class being generated. From this an extends or implements clause is constructed. For the precise definition of the auxiliary predicates see figure 7.2. Note the usage of `NODUP` to eliminate duplicate type names in the generated extends or implements clause.

7.3 Parametric and Bounded Parametric Types

As a first simple example of type-oriented programming we show how to simulate parametric and bounded parametric polymorphism. We explained in chapter 1 that parametric types can be seen as offering a very limited form of type-oriented programming. It is therefore reasonable to assume that TyRuBa, which we claim offers a capability for unrestricted type-oriented meta programming, should at least support parametric types.

The ability to express parametric types in TyRuBa is a straightforward result of being able to use logic terms in place of identifiers, more specifically in place of identifiers that represent type names. Reconsider the example of a parametric array class from section 1.6.1. This parametric array class can be expressed in TyRuBa as follows:

```
class_(Array<?El>,{
  private ?El[] contents;

  /** Construction */
  Array<?El>(int sz) { contents = new ?El[sz]; }

  /** Basic Array functionality */
  ?El at(int i) { return contents[i]; }
  void atPut(int i, ?El e) { contents[i]=e; }
  int length() { return contents.length; }
}).
```

Such a definition represents not one but several classes: one for each binding of the variables in the class name. In the given example, that is one `Array<?El>` class for each possible `?El` element type. We can also simulate bounded parametric polymorphism by imposing restrictions on the variables by using a rule with a condition. Figure 7.3 shows a translation of the example from figure 1.1 into TyRuBa. Note that if a class imposes a bound on one or more of its type parameters, then the rules asserting that the class implements an interface (or extends an other class) should also be guarded by the same condition.

Note that the `subtype` predicate is not defined by the representational mapping. It corresponds to the Java subtype relationship between classes and interfaces and is defined in TyRuBa itself. Figure 7.4 shows some rules which define `subtype` along with a few other useful predicates. These rules are provided as part of the initialization file which also defines the code generator and is included automatically into the TyRuBa rule base upon initialization.

With respect to the existing proposals for parametric types in Java [OW97, AFM97, MBL97], this emulation of bounded parametric polymorphism resembles most that from Pizza [OW97] and [AFM97]. The given implementation of `subtype` corresponds to the type restrictions with which type parameters in these two systems can be bounded. Pizza [OW97] and

```
interface_(Equality<?This>,{
  boolean equals(?This e);
}).

interface_(Searchable<?El>,{
  boolean contains(?El e);
}).

class_(Array<?El>,{
  ?El[] contents;

  /** Construction */
  Array<?El>(int sz) { contents = new ?El[sz]; }

  /** Basic Array functionality */
  ?El at(int i) { return contents[i]; }
  void atPut(int i, ?El e) { contents[i]=e; }
  int length() { return contents.length; }

  /** Searchable Interface */
  boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equals(at(i++));
    return found;
  }

}) :- subtype(?El,Equality<?El>).

implements_(Array<?El>,Searchable<?El>) :- subtype(?El,Equality<?El>).
```

Figure 7.3: A “bounded” parametric `Array` class in TyRuBa

```
/**** Predicates that deduce information from "_" facts */

/* Is there a class named ?cl */
class(?cl) :- class_(?cl,?body).

/* Is there an interface named ?itf */
interface(?itf) :- interface_(?itf,?body).

/* Does a class implement an interface (directly or indirectly) */
implements(?cl,?itf) :- implements_(?cl,?itf).
implements(?cl,?itf) :- BOUND(?itf),extends(?itf0,?itf),implements_(?cl,?itf0).
implements(?cl,?itf) :- BOUND(?cl),NOT(BOUND(?itf)),
                        implements_(?cl,?itf0),extends(?itf0,?itf).
implements(?cl,?itf) :- NOT(BOUND(?itf)),NOT(BOUND(?itf)),
                        extends(?itf0,?itf),implements_(?cl,?itf0).
implements(?cl,?itf) :- extends_(?cl,?super),implements(?super,?itf).

/* Does ?cl extend ?super (?cl class or interface) */
extends(?cl,?super) :- extends_(?cl,?super).
extends(?cl,?super) :- extends_(?cl,?super0),extends(?super0,?super).

/* Is ?cl a subclass of ?super */
subclass(?cl,?super):-extends(?cl,?super),class(?cl).

/* Is ?t1 a subtype of ?t2 according to Java type rules */
subtype(?t1,?t2) :- extends(?t1,?t2);implements(?t1,?t2).
subtype(?t,?t) :- class(?t);interface(?t).
```

Figure 7.4: Some rules deducing useful type information

[AFM97] are very similar to each other but because of its heterogeneous implementation, [AFM97] is somewhat more flexible. The terminology “heterogeneous” and “homogeneous” was introduced in [OW97]. We briefly explained the difference between homogeneouos and heterogeneous implementation in section 3.3.1. A heterogeneous implementation reinstantiates the code of a parametric class every time it is used whereas a homogeneous implementation shares the code. In order to be able to share the code between all instantiations of a parametric class more restrictions must be imposed. TyRuBa fits into the “heterogeneous” category since it generates separate Java source code for every instantiation of a class.

7.4 Fully Exploiting Turing Equivalence

TyRuBa is a fully Turing-equivalent meta language for manipulating pieces of Java code together with their type properties. This allows for sophisticated forms of generic programming where code for a certain class or interface can be generated by an arbitrary computation specified in Prolog. This section gives an example illustrating this. The example is a parametric class for a multidimensional array data structure where the dimension of the array is a parameter of the class. This example is interesting because it recurses over an integer representing the dimension of the array. An iteration like this leads to potential infinite loops when there is no adequate condition that makes it terminate after a finite number of steps. Hence this example can not be expressed in a system which guarantees that “compile-time type programs” always terminate.

We start with the most trivial case which ends the recursion, namely an array with dimension equal to zero:

```
class_(MArray<0,?El>, {
  private ?El contents;

  ?El elementAt() { return contents; };
  void setElementAt(?El el) { contents = el; }
}).
```

Before giving the recursive class that specifies how an array with n dimensions can be specified in terms of an array of $n-1$ dimensions, let us first have a look at how the specific example of a three-dimensional array can be implemented in terms of a two-dimensional array in figure 7.5. This will help to understand the recursive TyRuBa code we will give afterwards.

As can be seen, the implementation of a three-dimensional array in terms of a two dimensional one is pretty much as expected. It stores the array as a Java array of two dimensional arrays. Its constructor and accessor methods have an extra parameter to use for the Java array and pass the rest of the arguments on to the respective two dimensional constructor or method.

We now present the recursive[1] part of the implementation of `MArray` part of the implementation in figure 7.6. The condition of the rule restricts it to be applicable only when the dimensionality of the array is 1 or greater than 1. This restriction is responsible for the recursion ending when zero is reached. The condition of the rule also computes bindings

[1]The implementation of a three dimensional MArray was only given as a clarifying example and is subsumed by the recursive code

```
class_(MArray<3,?El>, {
  private MArray<2,?El>[] contents;

  MArray<3,?El> (int size3, int size2, int size1) {
     contents = new MArray<2,?El> [size3];
     for (int i=0;i<size3;i++) {
         contents[i] = new MArray<2,?El> (size2,size1);
     }
  }
  ?El elementAt (int index3,int index2,int index1) {
     return contents[index3].elementAt(index2,index1);
  }
  void setElementAt (?El el,int index3,int index2,int index1)
  {
     contents[index3].setElementAt(el, index2, index1);
  }
}).
```

Figure 7.5: A three-dimensional array

for the variables `?CFormals` and `?atFormals` by means of some auxiliary predicates. The definition of the auxiliary predicates used in this example are listed in figure 7.7. Note that the definition of `JavaCommaList` was given before in figure 7.2 so it is not repeated here.

One remark about the names of the formals in the generated formals list. The name of a formal is given by a compound term `a<?name,?counter>`. Remember that the hard-coded code generator has a name hashing scheme which allows compound terms (with no unbound variables) to be used as identifiers. Up to now we have just used this feature for simulating parametric type names, however we can just as well use it for names of variables and such. So this is just a way of making a list of formals with distinct names by including a counter as part of the name. The names of the formals will not be literally the same as those given in the three-dimensional array implementation but they are similar in spirit. The names will just be a little bit messier, for example "`a_Lindex_C3_R`" instead of just "`index3`".

7.5 Summary

In this chapter we showed how TyRuBa can be used for type-oriented logic meta programming. This is accomplished by implementing a code generator that is consistent with a representational mapping that reifies types (class and interface names) and the relationships between them (extends and implements). The representational mapping and code generator presented in this chapter are kept as coarse as possible while still enabling a minimal capability for type-oriented meta programming. We will present more fine-grained refinements in subsequent chapters as the need arises.

The coarse grained mapping already suffices to emulate parametric and bounded parametric types similar to existing proposals for adding parametric types to Java [OW97, AFM97]. The emulation of parametric polymorphism in TyRuBa is a straightforward consequence of using logic variables in logic terms representing type names. The kind of constraints that can be used to impose bounds on the type parameters depends on the information that is

```
class_(MArray<?Dim,?El>, {
  private MArray<?DDim,?El>[] contents;

  MArray<?Dim,?El> (?CFormals) {
     contents = new MArray<?DDim,?El>[?CFirst];
     for (int i=0;i<?CFirst;i++) {
         contents[i] = new MArray<?DDim,?El> (?CRest);
     }
  }

  ?El elementAt (?atFormals) {
     return contents[?atFirst].elementAt (?atRest);
  }

  void setElementAt (?El el, ?atFormals)
  {
     contents[?atFirst].setElementAt (?elAtRest );
  }
})
:- >(?Dim,0), +(?DDim,1,?Dim),
   JavaFormals(?Dim, int, size, ?CFormals0),
   JavaRemoveTypes(?CFormals0,[?CFirst | ?CRest0]),
   JavaFormals(?Dim, int, index, ?atFormals0),
   JavaRemoveTypes(?atFormals0, [?atFirst | ?atRest0]),
   JavaCommaList(?CFormals0,?CFormals),
   JavaCommaList(?CRest0,?CRest),
   JavaCommaList(?atFormals0,?atFormals),
   JavaCommaList(?atRest0,?atRest),
   JavaCommaList([el | ?atRest0],?elAtRest).
```

Figure 7.6: A multidimensional array class in TyRuBa

```
/*Generate a list of formals with type {?Type a<?Name,?counter>}*/
JavaFormals(0, ?Type, ?Name, []).
JavaFormals(?Dim, ?Type, ?Name, [{?Type a<?Name,?Dim>} | ?RestFormals])
 :- >(?Dim,0), +(?DDim,1,?Dim),
     JavaFormals(?DDim, ?Type, ?Name, ?RestFormals).

/*Take a list constructed in JavaFormals and remove the types*/
JavaRemoveTypes([],[]).
JavaRemoveTypes([{?T ?I} | ?RT] , [?I | ?R]) :- JavaRemoveTypes(?RT,?R).
```

Figure 7.7: Auxiliary predicates used in MArray

reified by the representational mapping. The mapping presented in this chapter is very coarse grained. It is however possible to define and implement more fine-grained mappings that reify more information, and we will effectively do this in subsequent chapters.

At the end of this chapter we have given a tentative example illustrating the potential power of having a Turing-complete meta language to manipulate types and the code that implements them. This example illustrates the potential use of recursion in type-manipulating meta programs.

Chapter 8

Benchmarking TyRuBa's Expressiveness

8.1 Introduction

We have explained in previous chapters what is meant by type-oriented logic meta programming and how the TyRuBa system provides a means to this end in the context of Java. In this chapter we will take another look at the benchmark programming assignments of section 2.9. We used these benchmarks as a means to explore and assess the expressiveness of Gofer's type language as a programming language.

Benchmark problems 1 and 2 correspond roughly to the motivating examples from section 1.6, illustrating the loss of expressiveness due to the lack of a suitable decision making feature in typical Java parametric types proposals. Gofer's type language is considerably more sophisticated and even though it was not designed as such it comes very close to a programming language. Because of this, benchmark problems 1 and 2 were easily solved in Gofer. When we push the boundary some more however we soon reach the limits of its expressiveness. Benchmark problem 3 could only be partially solved. The solution to benchmark 4 was very complicated and only achieved using the type language in ways it was clearly not intended to be used. As a result, the solution was easily shown to have some problems due to ambiguous type errors which make it only of very limited use in practice.

The type languages in traditional type systems have a lot of restrictions built into them to ensure termination, avoid ambiguity and generally keep the type checking or inferencing algorithm tractable. As a consequence of these restrictions the type language is not designed as a programming language and has only a very limited capability for supporting type-oriented meta programming. In contrast, TyRuBa offers the possibility for type-oriented meta programming in an unrestricted (logic) meta language that is truly a programming language. In this chapter we will show what we gain in terms of expressiveness by implementing the benchmark problems from section 2.9 in TyRuBa. We will see that all of the programming assignments can be easily expressed. While studying the first example it will however become evident that TyRuBa with the coarse grained mapping proposed in chapter 7 is not sufficient to solve the problems. We need a more fine-grained mapping that splits up the class declaration into smaller parts. After defining and implementing the more fine-grained mapping all of the benchmark problems are easily solved.

8.2 Conditional Implementation Re-examined

Let us start by re-examining benchmark 1. In keeping with the motivating example from section 1.6 we will again implement arrays rather than lists. For easy reference we have repeated the example in figure 8.1. Rather than expressing the example in the "fictional" parametric extension of Java used in section 1.6, we express it in TyRuBa with the representational mapping from chapter 7 (i.e. the "_" mapping).

```
interface_(Equality<?This>, {
  boolean equals(?This e);
}).

interface_(Searchable<?El>, {
  boolean contains(?El e);
}).

implements_(Array<?El>,Searchable<?El>) :-
  implements(?El,Equality<?El>).

class_(Array<?El>, {
  ?El[] contents;

  /** Construction */
  Array<?El>(int size) { contents = new ?El[size]; }

  /** Basic Array functionality */
  ?El at(int i) { return contents[i]; }
  void atPut(int i, ?El e) { contents[i]=e; }
  int length() { return contents.length; }

  /** Searchable Interface */
  boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equals(at(i++));
    return found;
  }

}) :-  implements(?El,Equality<?El>).
```

Figure 8.1: A Parametric Array Class

Note the use of `implements` instead of `implements_` to constrain the `?El` parameter to be a class implementing the `Equality` interface. This is intentional, we purposefully refer to the `implements` predicate from the initialization file (figure 7.4). Using `implements_` instead would only signify element types which directly implement the equality interface but would "forget about" classes that implement it indirectly (i.e. inherit it or have it implicitly included via a wider interface).

Despite being written in TyRuBa, the code in figure 8.1 still exhibits the same problem as the code previously given: the type bound for element types is imposed on the array class as a whole. We cannot instantiate the `Array` class for elements not implementing equality. We could "solve" this problem by providing an extra declaration of `Array` without the implementation of `Searchable` as shown in figure 8.2. This "solution" is inadequate because it duplicates the code for basic array functionality.

```
/** Arrays of elements NOT supporting Equality tests */
class_(Array<?El>,{
  /** Basic Array functionality */
  ...
}) :- NOT(implements(?El,Equality<?El>)).

/** Arrays of elements that DO support Equality tests */
implements_(Array<?El>,Searchable<?El>)
  :- implements(?El,Equality<?El>).

class_(Array<?El>, {
  /** Basic Array functionality */
  ...
  /** Searchable Interface */
  ...
}) :- implements(?El,Equality<?El>).
```

Figure 8.2: Two alternative, mutually exclusive declarations of `Array`

8.3 More Fine-Grained Reification of Classes

In order to really solve the problem we must treat the body of a class declaration not as a single monolithic block of code, but chop it up into smaller pieces. Thus, restrictions can be imposed on every piece individually. For the purpose of this example (and of the other benchmark problems treated later on) it will suffice to chop the class up into pieces responsible for implementing interfaces, on an interface by interface basis. This is accomplished by implementing a more fine grained refinement of the code generator which is consistent with the following representational mapping.

8.3.1 Representational Mapping

The basic idea of the representation scheme is that a class body is chopped up into pieces providing the implementations for interfaces[1]. Every one of these pieces[2] is put into a proposition of the form:

implements_I(*className*,*interfaceName*,{...}).

[1]We implicitly assume that interfaces do not overlap. This is not a serious restriction because it is always possible to create an extra interface for the shared part of two overlapping interfaces and add this new interface to their extends clause. Note that this only requires editing the interface declarations themselves, but does not have any impact on the classes which implement the interfaces.

[2]Code implementing an interface which is inherited from a superclass must not be declared in an `implements_I` clause for the subclass. Only code with occurs in the actual declaration of the class itself must be declared.

The remaining code in the class body is put into a proposition:

```
class_I(className,{...}).
```

Just like the _ mapping, the _I mapping reifies the `extends` and `implements` relationship which is declared between classes and interfaces. Whenever a type is listed in the `extends` clause of another type, this is asserted by a fact of the form:

```
extends_I(subtypeName,superTypeName).
```

Thus, the `extends_I` predicate is merely a renamed version of the `extends_` predicate.

The implements relationship is expressed in a similar way, by means of a fact of the form:

```
implements_I(className,interfaceName).
```

Note the "overloaded" usage of the name `implements_I`. When used with two arguments it states the occurrence of an interface name in a class's implements clause. When used with three arguments it also provides a piece of source code that must be inserted into the class's body and actually provides the needed method declarations.

A schematic Java program and the corresponding set of logic propositions are given in figure 8.3. Note the "`_I`" at the end of predicate names. The predicates of the representational mapping introduced in this chapter all follow this naming convention. The "`I`" stands for "Interface", indicating the representational mapping which considers classes at *interface granularity*.

8.3.2 Code Generator

We now present a code generator consistent with the representational mapping given above. Its implementation is given in figure 8.4. It is implemented on top of the "`_`" code generator in a way similar to how this itself was implemented on top of the hard-coded code generator. The implementation of the code-generator is thus just a set of rules that establishes how the "`_`" predicates are deduced from the "`_I`" predicates. The most complicated of these rules is the one for `class_`. It finds all pieces of a class's body which implement interfaces on that class. From this, duplicate interface names are eliminated by using `NODUP`. Finally, all of the thus obtained pieces of class body are pasted together with the class's basic class body.

8.4 Solving the Benchmark Problems

Because TyRuBa, with the proposed representational mapping, constitutes an environment where a real programming language is available for manipulating Java interface types and pieces of Java code that implement them, the possibilities are virtually unlimited. One can write logic programs to implement classes, define interfaces, implement interfaces on classes etc. These programs may infer different implementations or interfaces depending on a type parameter. The extra expressive power this offers is more than sufficient to easily express all of the benchmark problems.

```
interface interface1
extends superInterface1,superInterface2
{ ... }

interface interface2 { ... }

class aClass extends aSuperclass
implements interface1, interface2 {
  /** Some code specific to aClass */
  ...
  /** Some code implementing interface1 */
  ...
  /** Some code implementing interface2 */
  ...
}
```

```
interface_I(interface1,{...}).
extends_I(interface1,superInterface1).
extends_I(interface1,superInterface2).

interface_I(interface2,{...}).

class_I(aClass,{
  /** Some code specific to aClass*/
  ...
}).
extends_I(aClass,aSuperClass).
implements_I(aClass,interface1,{
  /** Some code implementing interface1 */
  ...}).
implements_I(aClass,interface2,{
  /** Some code implementing interface2 */
  ...}).
```

Figure 8.3: A Java Program (on top) and its representation as a set of propositions

```
class_(?class,{?basics ?interfaces}) :-
  class_I(?class,?basics),
  FINDALL(NODUP(?itf,implements_I(?class,?itf,?how)),
          ?how,?interfaces).

implements_(?cl,?itf) :- implements_I(?cl,?itf).
extends_(?a,?b) :- extends_I(?a,?b).
interface_(?itf,?body) :- interface_I(?itf,?body).

implements_I(?cl,?itf) :- implements_I(?cl,?itf,?body).
```

Figure 8.4: A Code Generator splitting up classes into “interface implementations”

8.4.1 Conditional Interface Implementation

Benchmark 1 corresponds to the "Conditional Interface Implementation Problem" from section 1.6.1. The problem comes down to the fact that sometimes we are not satisfied with simply putting type constraints on an entire class but want finer control on an interface by interface basis. As an example we presented a parametric `Array` class (figure 8.1) which implements the `Searchable` interface. In this particular example we would want to make the implementation of the `Searchable` interface conditional, depending on whether the type parameter provides an implementation for the `Equality` interface. Typical parametric type systems do not provide a way to do this. They usually do offer a mechanism of *bounded* parametric polymorphism which allows constraining the instantiation of a parametric class by imposing a "bound" on type parameters. However this is an all or nothing situation. The class can either be instantiated, or it cannot be instantiated. A situation where in some cases the class is instantiated but without an implementation for a certain interface is impossible.

In TyRuBa, with a representational mapping which divides classes into parts implementing interfaces, we may impose restrictions on every separate piece of the class body. We may say for every individual part on what condition it may be included into the actual class declaration body that will appear in the generated Java program. This is a very natural way of providing "conditional interface implementations". Figure 8.5 shows TyRuBa code for an `Array` class with a conditional implementation of the `Searchable` interface.

```
class_I(Array<?El>,{
  private ?El[] contents;

  /** Construction */
  Array<?El>(int size) { contents = new ?El[size]; }

  /** Basic Array functionality */
  ?El elementAt(int i) { return contents[i]; }
  void setElementAt(?El e,int i) { contents[i]=e; }
  int length() { return contents.length; }
}).

implements_I(Array<?El>,Searchable<?El>,{
  public boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equals(elementAt(i++));
    return found;
  }
}) :- implements(?El,Equality<?El>).
```

Figure 8.5: Array with conditional implementation of `Searchable`

8.4.2 Positioning Abstract Code

Benchmark problem 2 corresponds to the second motivating example from section 1.6.2. In that section we discussed that it is not always adequate to put abstract code into abstract classes. The reason is that it is not always easy or possible to find a suitable place in the class tree for the abstract class. The problem becomes exponentially worse when several abstract classes are to coexist in the same class library. A nicer solution we also discussed is using a mixin class but this requires explicit subclassing to add the abstract functionality to the class and makes it difficult to specialize the abstract functionality in specific classes. In TyRuBa it is very natural to write an abstract implementation for an interface completely separate from the class hierarchy and declare to which classes it should be applied by means of an arbitrary logic expression. Figure 8.6 shows a logic rule that implements the searchable interface on any collection class that implements the `Enumerable` interface and has elements comparable for `Equality`.

```
/* Abstract implementation of Searchable on top of Enumerable */
implements_I(?X,Searchable<?El>,{
 public boolean contains(?El e) {
    boolean found = false;
    Enumeration<?El> elems = this.elements();
    while (!found && (elems.hasMoreElements()))
      found = e.equals(elems.nextElement());
    return found;
 }
}):- implements(?X,Enumerable<?El>),
     implements(?El,Equality<?El>).

/* Something which contains elements can usually enumerate them */
interface_I(Enumerable<?El>,{
  Enumeration<?El> elements();
}).

/* TyRuBa-ized version of Java.util.Enumeration */
interface_I(Enumeration<?El>,{
  boolean hasMoreElements();
  ?El nextElement();
}).
```

Figure 8.6: An Abstract implementation of `Searchable` in TyRuBa

8.4.3 Type-Dependent Internal Representation

Benchmark problem 3 was the implementation of an abstract data type `Set`, the internal representation of which depends on the type of elements that will be stored in it. This requires another kind of decision making than the two former benchmark problems. It turned out that Gofer's type language falls short just an inch to solve this problem completely. Apparently Gofer's type language can be used to make decisions about functionality and implementation of functionality onto a data structure, but it cannot be used to express decisions that affect the data structure itself.

Now let us have a look at the TyRuBa solution. We start by defining an interface for the functionality of sets. Basically one can do two things with a set: `insert` elements into it, or ask whether it `contains` a particular element:

```
interface_I(SetInterface<?El>,{
   void insert(?El e);
   boolean contains(?El e);
}).
```

Before going on we first implement the classes that will be used to represent lists and trees respectively. These can be found in figure 8.7. They are just two simple classes with a few instance variables and a constructor to initialize them.

```
class_I(List<?El>,{
  ?El first;
  List<?El> rest;

  List<?El>(?El f,List<?El> r) {first=f;rest=r;}
}).

class_I(Tree<?El>,{
  ?El elem;
  Tree<?El> left;
  Tree<?El> right;

  Tree<?El>(?El e,Tree<?El> l,Tree<?El> r) {
     elem=e; left=l; right=r;
  }
}).
```

Figure 8.7: `Tree` and `List` internal representation classes

Next, we will declare the concrete implementations of this interface onto a class `Set<?El>`. We will have two declarations of this class, with two different internal representations.

```
/** If the elements implement Equality use this */
class_I(Set<?El>,{
   private List<?El> representation = null;
}) :- implements(?El,Equality<?El>).

/** If the elements implement Ordered use this instead */
class_I(Set<?El>,{
   private Tree<?El> representation = null;
}) :- implements(?El,Ordered<?El>).
```

Note that the order in which these declarations are given matters. The last declared one will take precedence over the other. This is important because usually a class that implements `Ordered` will also implement `Equality`. By giving the `Tree` representation after the `List` representation we ensure that a `Tree` representation will be chosen for sets with elements that implement both `Ordered` and `Equality` at the same time.

We still have to declare how these internal representations can be used to implement the `SetInterface`. We start with the implementation of `List` represented sets, listed in figure 8.8. The implementation is pretty straightforward and does not need a lot of explanation. The

```
implements_I(Set<?El>,SetInterface<?El>,{

   public void insert(?El e) {
     if (!contains(e)) {
       representation = new List<?El>(e,representation);
     }
   }

   public boolean contains(?El e) {
     return listContains(representation,e);
   }

   private static boolean listContains(List<?El> l,?El e) {
     return (l!=null) && (
       l.first.equals(e) || listContains(l.rest,e)
     );
   }

}) :- implements(?El,Equality<?El>).
```

Figure 8.8: Implementing `SetInterface` for `List` representations

`contains` method calls an auxiliary function that traverses the list sequentially. The `insert` method first checks whether the element is not already in the list. If it is not, then it is added to the front of the list.

The implementation of the `SetInterface` for sets with an internal `Tree` representation is listed in figure 8.9. We will not explain this code in detail since it is just a simple and straightforward implementation of the insert and lookup algorithms on binary search trees.

Again, it is important that the `Tree` implementation is given last, after the `List` implementation to make sure that the `Tree` implementation will be chosen in case element types implement both `Equality` and `Ordered`.

This concludes the solution for benchmark 3. The TyRuBa solution to the benchmark does not have the problem found in the Gofer solution. Whenever code is generated for a class `Set<?El>` the appropriate implementation with either a list or a tree representation will be chosen automatically depending on the type of elements. The user of the set abstract data type does not have to be concerned with the internal representation at all.

8.4.4 Iteration or Recursion

Benchmark problem 4 was an implementation of tuples which contain a fixed number of elements. Each element possibly of a different static type. The arity of the tuple may vary but is known at compile time. We were able to express this problem eventually in Gofer, but we were clearly nearing the edge of the type language's expressiveness. We had to define our own numerals and "list of type" representations. The Gofer solution was not entirely satisfactory because it was not very usable in practice because of ambiguous-type related errors hiding around the corner.

We will now express the same programming benchmark in TyRuBa. We will do so in two different ways. The first solution is nearly the same as the Gofer solution. The difference is

```
implements_I(Set<?El>,SetInterface<?El>,{

   public void insert(?El e) {
     representation = insertTree(representation,e);
   }

   private static Tree<?El> insertTree(Tree<?El> t,?El e) {
     if (t==null)
        return new Tree<?El>(e,null,null);
     else {
        int comparison = t.elem.compareTo(e);
        if (comparison<0)
          t.left = insertTree(t.left,e);
        else if (comparison>0)
          t.right = insertTree(t.right,e);
        return t;
     }
   }

   public boolean contains(?El e) {
     return treeContains(representation,e);
   }

   private static boolean treeContains(Tree<?El> t,?El e) {
     if (t==null)
       return false;
     else {
        int comparison = t.elem.compareTo(e);
        if (comparison<0) /*Less*/
          return treeContains(t.left,e);
        else if (comparison>0) /*Greater*/
          return treeContains(t.right,e);
        else /*Equal*/
          return true;
     }
   }

}) :- implements(?El,Ordered<?El>).
```

Figure 8.9: Implementing `SetInterface` for `Tree` representations

that we do not have to create our own numbers and lists. These are provided by the TyRuBa logic programming language. This solution also has no problems with the kind of errors that render the Gofer solution nearly useless in practice. TyRuBa does not perform the stringent ambiguity checks that Gofer does.

We will also present another solution. This will make more explicit use of TyRuBa's capability as a programming language. The second solution will have a more efficient, non-recursive representation of tuples and a more natural way of constructing tuples. To accomplish this we will have to use more of TyRuBa's capabilities as a true programming language.

Recursive Tuples

Let us start with the recursive representation of tuples which is the simplest of both. We will define a tuple by means of a parametric class, the parameter of which is a list of types corresponding to the respective types of the elements stored in the tuple. We begin by defining tuples of arity zero. These are extremely simple since they do not contain anything.

```
/** 0 arity Tuple */
class_I(Tuple<[]>,{
}).
```

Next we will declare tuples of arity greater than zero inductively. A tuple of arity $n+1$ has an instance variable for storing the first element and another instance variable that contains a nested tuple of arity n with the remaining elements.

```
/** Inductive definition of Tuple of arity greater than 0 */
class_I(Tuple<[?T|?R]>,{
  private ?T first;
  private Tuple<?R> rest;

  Tuple<[?T|?R]>(?T f,Tuple<?R> r) {
    first = f;
    rest  = r;
  }
}) :- class(Tuple<?R>).
```

The `Project` type class from the Gofer solution (section 2.9.2) corresponds to the following interface declaration:

```
interface_I(Project<?index,?El>,{
  ?El project<?index>();
}).
```

Finally we declare the implementation of this interface onto tuples by means of one base case for projecting to index zero and one inductive case for indexes greater than zero. Again this is nearly identical to the Gofer solution.

```
implements_I(Tuple<[?El|?R]>,Project<0,?El>,{
  public ?El project<0>() { return first; };
}).

implements_I(Tuple<[?F|?R]>,Project<?i,?El>,{
  public ?El project<?i>() { return rest.project<?iMinus1>();}
}) :- implements(Tuple<?R>,Project<?iMinus1,?El>),
      +(?iMinus1,1,?i).
```

This solves the benchmark problem quite nicely. The solution is a little bit simpler and more intuitive than the Gofer solution because we did not have to invent our own representation for numbers and lists of types. Remember that the Gofer solution had a very restricted usability because when using the tuples we get problems with "ambiguous type" related errors. Because TyRuBa imposes none of the strict "ambiguous type" related restrictions Gofer imposes, this is not a problem either.

The solution with recursively represented tuples is not optimal. We can do better in TyRuBa if we make more elaborate use of its power as a programming language than we do now. Creating a tuple is not practical, it is far too verbose because it requires creating a nested tuple object. The following example which creates a tuple `(1,"Hello",'f')` clearly illustrates the problem:

```
Tuple<[int,String,char]> triplet =
    new Tuple<[int,String,char]>(1,
      new Tuple<[String,char]>("Hello",
        new Tuple<[char]>('f',
          new Tuple<[]>())));
```

The performance characteristics of recursive tuples are also not as good as we would like. A "flat" representation where a tuple of arity n is represented by an object with n instance variables would be better. A recursive representation clearly consumes more space because of all the internal tuple objects. It also consumes more time because the projection function onto index $n+1$ has to invoke another projection function recursively n times to arrive at the correct nesting level whereas in a flat representation the correct instance variable could simply be accessed directly.

Non-recursive, flat tuples

The problems of the solution in the previous section are due to the fact that we "translated" the tuple implementation too directly from the Gofer solution presented in section 2.9.2. In Gofer we had no choice but to implement the tuples recursively because that is the only way we can offer generic support for tuples of different arities.

In TyRuBa this is not true. We can implement the tuples any way we like because we have the full power of a programming language at our disposal to generate the required implementation. We may therefore just as well choose a non-recursive tuple representation.

Before continuing and presenting the TyRuBa code for flat tuples let us have a preview of what the generated code will look like eventually. As an example consider the code generated for the tuple class `Tuple<[int,String,char]>` listed in figure 8.10. We have applied some minor cosmetic changes to make the code more readable: we "unmangled"[3] identifier names,

[3]Name mangling of terms was explained in section 6.4.3. As a quick reminder note that the term `element<String,1>` for example corresponds to the "mangled" Java identifier `element_LString_C1_R`.

```
class Tuple<[int,String,char]>
implements  Project<2,char> ,  Project<1,String> , Project<0,int>
{
  /** Constructor */
  Tuple<[int,String,char]> (int element<int,0>,
                            String element<String,1>,
                            char element<char,2>    )
  {
    this . element<int,0> = element<int,0> ;
    this . element<String,1> = element<String,1> ;
    this . element<char,2> = element<char,2> ;
  }

  /** interface Project<2,char> */
  private char element<char,2> ;
  public char project<2> ( ) { return element<char,2> ; } ;

  /** interface Project<1,String> */
  private String element<String,1> ;
  public String project<1> ( ) { return element<String,1> ; } ;

  /** interface Project<0,int> */
  private int element<int,0> ;
  public int project<0> ( ) { return element<int,0> ; } ;
}
```

Figure 8.10: Example code generated for a “flat” 3-tuple

cleaned up the indentation and added some comments.

The representation of a 3-tuple is an object with three instance variables to hold the elements of the tuple. The names of these instance variables have the form `element<?Type,?idx>`. The `?idx` is the index of the element in the tuple. When invoking the projection function `project<1>` for example, this will simply return the value of the instance variable with `?idx` equal to 1 which in this particular example is `element<String,1>`. Strictly speaking it is not really necessary to have the type of the element part of the instance variable's name. This was merely convenient because it simplifies the TyRuBa program by making it easy to tell the type of an instance variable from its name.

Now let us have a look at the TyRuBa program which implements flat tuple representations. We start by implementing an auxiliary predicate `tupleElementNames` to compute the names of the instance variables of the tuple class.

```
tupleElementNames(?T,?N) :- tupleElementNames(0,?T,?N).
tupleElementNames(?idx,[],[]).
tupleElementNames(?idx,[?T|?TR],[element<?T,?idx>|?R]) :-
  +(?idx,1,?idxPlus1),
  tupleElementNames(?idxPlus1,?TR,?R).
```

When this predicate is invoked with a list of types in `?T` it will compute a list of the corresponding instance variable names in `?N` as the following example shows:

```
:- tupleElementNames([ int, String, char ],?N).
#SOLUTION : tupleElementNames([ int, String, char ],
   [ element<int,0>, element<String,1>, element<char,2> ])
```

This predicate will be used to compute the names of the instance variables and also for the names of the formal arguments of the tuple's constructor function. Below is the declaration of the tuple class with its constructor. The instance variables are not yet provided in the basic class functionality but will be provided with the implementation of the respective `Project` interfaces later.

```
class_I(Tuple<?TypeList>,{
  /** Constructor */
  Tuple<?TypeList>(?constructorFormals) {
    ?constructorBody
  }
}) :- tupleElementNames(?TypeList,?formalsNamesList),
      JavaFormals(?TypeList,?formalsNamesList,?constructorFormals),
      tupleConstructorBody(?formalsNamesList,?constructorBody).
```

The formals of the constructor are generated by means of a predefined auxiliary predicate `JavaFormals` that pairs up a list of type names and a list of formal argument names. The body of the constructor is computed by the auxiliary predicate `tupleConstructorBody` which is given below:

```
tupleConstructorBody([],{}).
tupleConstructorBody([?i | ?r],{this.?i=?i; ?cr}) :-
```

```
   tupleConstructorBody(?r,?cr).

interface_I(Project<?index,?El>,{
  ?El project<?index>();
}).
```

This predicate generates a list of assignment statements of the form `this.?i=?i;` for every instance variable name `?i`.

Finally we implement the projection interfaces on this class. The respective instance variables are defined together with each interface implementation.

```
implements_I(Tuple<?TypeList>,Project<?idx,?El>,{
  private ?El element<?El,?idx>;

  public ?El project<?idx>() { return element<?El,?idx>; };
}) :- tupleElementNames(?TypeList,?elements),
      element(element<?El,?idx>,?elements).
```

The predefined predicate `element` is true when the first argument is an element of the list in the second argument. The result of the query `element(element<?El,?idx>,?elements)` will thus find the instance variable for the correct index `?idx` in the list of instance variables `?elements` and bind the name of its type to `?El`.

This solution is somewhat more complicated than the recursive tuple solution. We needed a number of auxiliary predicates to iterate over a list and to generate names of instance variables with a counter. What this extra effort gets us however is a more efficient representation for tuples. The tuples are also more easily created by means of a constructor function of which the arity corresponds to the arity of the tuple. A tuple of arity three for example is created as follows:

```
Tuple<[int,String,char]> test= new Tuple<[int,String,char]>(1,"Hi",'f');
```

The point of all this is that if the more straightforward solution is not completely satisfactory, we can always resort to the power of TyRuBa as a programming language. At the cost of some extra (meta) programming we can then generate exactly what we want.

8.5 Summary

In this chapter we re-examined the benchmark programming tasks from section 2.9. We were able to solve them in TyRuBa in an elegant and natural way. In order to solve the problems, we have introduced a representational mapping which looks at a class body as being composed of pieces of source code responsible for implementing Java interfaces.

An overview of the benchmark problems and a short discussion of the solutions in TyRuBa can be found in figure 8.11. The overall result is that TyRuBa solves all of the problems quite naturally. This illustrates that it is indeed worthwhile to consider designing a type-language as a true programming language, purposefully ignoring concerns such as decidability and ambiguity in order to augment the expressive power of the type language tremendously. This is worthwhile since it considerably improves the power of the language for writing generic code,

Decision-making benchmarks	
Benchmark 1	Conditional implementation of a function `contains` on an array: Can be elegantly expressed
Benchmark 2	Abstract implementation of `contains` for `Enumerable` collection types: Can be elegantly expressed: A logic expression can be used to characterize which classes an abstract implementation is applicable to. This allows the abstract code to be expressed more independently of the actual class tree structure.
Benchmark 3	Implementation of a `Set` abstract data type the internal representation of which depends on the type of elements stored: Can be elegantly expressed. Two different implementations of a single class can be provided. Either one is chosen dependent on the truth value of logic expressions.

Iteration / recursion benchmarks	
Benchmark 4	Can be elegantly expressed. Integers and lists are native data types of the type programming language. We have shown two solutions. The first one was straightforward and represented tuples recursively. This is not an optimal solution because creating tuples is too verbose and because the representation is too time and space consuming. A second more efficient solution represented tuples non-recursively. This solution could be expressed by means of some extra (meta) programming to generate the names for the instance variables and the formal arguments and the body of the constructor method.

Figure 8.11: Summary of the benchmark results for TyRuBa

as is clearly illustrated by the benchmark solutions. We think this is of particular importance for object-oriented languages and type systems since these are especially concerned with genericity and polymorphism.

Chapter 9

Case Study

9.1 Introduction

This chapter presents an example which illustrates the use of type-oriented logic meta programming to obtain a high degree of genericity. The example makes very sophisticated use of the logic meta language to manipulate and construct types and generate concrete implementations for the types. The user of a framework only implements part of the class hierarchy. If he respects some conventions that are prescribed by the framework implementor, then another part of the class hierarchy will be generated automatically. This style of programming is very powerful and enables sophisticated forms of genericity.

We will illustrate the technique in this chapter with a simple framework that implements a hierarchy of classes and interfaces for representing the expressions of a calculator. The framework will provide a number of expression classes and interfaces, and a factory class to build expressions. The framework instantiator only implements classes representing values. The rest of the classes and interfaces will be derived from this by the framework.

Before presenting the actual example we first introduce a representational mapping, a code generator and a number of predicates that reify information about methods, instance variables and constructors in classes and interfaces. These will be used throughout the rest of the chapter in expressing the example.

9.2 Representational Mapping

We will not present the implementation of the code generator but only describe the representational mapping itself and the information that is derived from it by a number of predicates defined in initialization files. The exact implementation of these predicates will also be omitted, we only describe them and the kind of information they are intended to make accessible. The complete listing of the initialization files can be found in appendix B. All the predicates directly linked to the representational mapping in this chapter will follow the naming convention of ending with "`_M`".

The representational mapping we use in this chapter reifies the implements and extends relationship between classes and interfaces as did the mapping from chapter 7. The predicates `extends_M` and `implements_M` are just renamed versions of the predicates `extends_` and `implements_`. We briefly repeat their meaning here as a reminder. Whenever a type `?Super` occurs in an extends clause of a type `?Sub` this is asserted as follows:

```
extends_M(?Sub,?Super).
```

Implements clauses are handled similarly. Whenever an interface `?I` occurs in an implements clause of a class `?C` this is asserted as follows:

```
implements_M(?C,?I).
```

The "`_M`" mapping refines the "`_`" mapping. It also reifies information about individual members of classes and interfaces whereas the "`_`" mapping treated class and interface bodies as atomic units. Therefore predicates `class_M` and `interface_M` do not contain a class or interface body anymore. The `class_M` predicate simply states the presence of a class declaration in the generated source file as follows:

```
class_M(?C).
```

The presence of an interface `?I` is declared similarly:

```
interface_M(?I).
```

Classes are viewed as being composed of methods, instance variables and constructors. Interfaces are viewed as being composed of methods. The types of instance variables, the argument types of constructors and methods, and the return type of methods are reified. For simplicity we will not reify exceptions and tags such as private, public or protected since we won't need that information for the examples in this chapter.

Whenever a type `?T` (class or interface) declaration contains a method `?m` with return type `?R` and argument types `?A1`,`?A2`, . . . `?An` this is asserted as follows:

```
method_M(?T,?R,?m,[?A1,?A2,...,?An],{...declaration...}).
```

The Java code between the `{}` is the entire declaration as it will appear in the body of the class or interface. The following shows an example of a method declaration for an `add` method on a class called `Integer`.

```
class_M(Integer).
method_M(Integer,Integer,add,[Integer],{
 Integer add(Integer other) {
   return new Integer(this.value+other.value);
 }
}).
```

The corresponding declaration in Java can be found in figure 9.1.

Declarations for constructors are asserted in a similar way. When a class `?C` contains a declaration for a constructor with argument types `?A1`,`?A2`, . . . `?An` this is asserted as follows:

```
constructor_M(?C,[?A1,?A2,...,?An],{...declaration...}).
```

The constructor from figure 9.1, for example, is expressed as follows:

```
class Integer {

 int value = 0;

 Integer(int init) {
   value = init;
 }

 Integer add(Integer other) {
   return new Integer(this.value+other.value);
 }
}
```

Figure 9.1: Example Java class declaration

```
constructor_M(Integer,[int],{
 Integer(int init) {
   value = init;
 }
}).
```

The presence of a declaration for a variable of type `?T` with name `?n` and initializer `?i` in a class `?C` is asserted as follows:

```
var_M(?C,?T,?n,?i).
```

The variable declaration in figure 9.1, for example, is expressed as follows.

```
var_M(Integer,int,value,{=0}).
```

This concludes what we have to say about the representational mapping itself.

9.3 Derived Information

The information reified by the representational mapping is very basic, it only tells about things declared in classes or interfaces directly, but does not state the features they inherit from super types. Usually when meta programs need to know whether a type has a certain feature, such as a method or instance variable they do not care whether this feature is declared in the class itself or inherited from one of its super classes. In order to make this kind of information accessible, we implemented a number of predicates that deduce useful higher level information from the basic information reified by the representational mapping. We only list the predicates and describe their meaning. For their exact definition by means of TyRuBa rules we refer to appendix B which contains listings of the initialization files.

The predicate `extends` is the transitive closure of the `extends_M` predicate. Hence `extends(?A,?B)` is true if and only if there is a chain of types starting with `?A` and ending with `?B` such that every type in the chain appears in the extends clause of the preceding one.

The predicate `subtype` implements the Java subtype relationship between classes and interfaces. Which is, in other words, the transitive reflexive closure of the union of the `implements_M` and `extends_M` predicates.

The predicate `implements` is a restriction of `subtype` that is true only when the first argument is a class and the second an interface.

The predicates `feature` and `feature1` provide a uniform way of asking whether a class or interface has a "feature". A "feature" may be a method, a variable or a constructor. It is possible to ask whether a class or interface has a feature directly present (i.e. *not* via inheritance from a super type) by means of the predicate `feature1` as follows:

```
feature1(?type,?feature).
```

The predicate `feature` is like `feature1` but also takes inheritance into account. Note that not all features are inheritable from super types. For example, (non private) methods are inherited, but constructors are not. Features are described by TyRuBa terms as outlined in figure 9.2.

Term	Represents
`var<?T,?n>`	A variable of type `?T` with name `?n`.
`method<?R,?m,[?A1,...,?An]>`	A method `?m` with return type `?R` and argument types `?A1`, ...,`?An`
`constructor<[?A1,...,?An]>`	A constructor with argument types `?A1`, ...,`?An`

Figure 9.2: Representation of "features" as TyRuBa terms

Note the use of the following naming convention. When we use the number 1 as a postfix in names of predicates this usually signifies "directly" in some or other way. The name without the postfix is then obtained as a composition or transitive closure of some kind. In this case, `feature` is a composition of `feature1` with the transitive closure of the inheritance relationship on classes.

9.4 An Evaluator for Arithmetic Expressions

We will now start presenting the example of a generic expression evaluator. Rather than presenting the generic framework by itself we will start from the instantiator's point of view, with a concrete instance of the framework in mind: an evaluator of arithmetic expressions with integers and floating point numbers. Keeping the concrete instantiation in mind will make it easier to understand the framework code.

The instantiator implements two classes for representing integer values and floating point values respectively. The declarations of these can be found in figure 9.3. Note that the names of the operations on the classes are somewhat peculiarly written as terms of the form `op<?name>`. This naming convention is part of the contract between framework instantiator and developer. The framework will make use of this naming convention to identify what messages on values represent operations. Except for the names of the operations we have used Java syntax in these figures to make them more readable. It is assumed however that in reality they are written in TyRuBa using the code generator and representational mapping presented in section 9.2. From now on we will write explicit TyRuBa syntax when we are using logic rules and terms in such a way that the code cannot be expressed in Java directly. It will be implicitly assumed that everything is expressed in TyRuBa regardless of how it is presented in the figures.

```
class Integer {
 int value;

 public Integer(int init) {value=init;}

 public op<add>(Integer other) {
   return new Integer(value+other.value);
 }
 public op<sub> ...
 public op<mul>...
}

class Float {
  float value;

  public Float(float init) {value=init}
  public op<add>...
  ...
}
```

Figure 9.3: Instantiator's classes for Integer and Float values

In order for the framework to know that these classes represent values, the instantiator must "declare" this by stating the following facts:

```
value(Integer).
value(Float).
```

We can now start presenting the code of the framework. From the above declarations, given by the instantiator, the framework will derive the calculator expression class and interface hierarchy and the factory class. Let us see how this is accomplished by means of logic rules.

As there are two kinds of values there should also be two kinds of expressions which return the respective value type when evaluated. We will make all expressions of a certain type `?T` implement the interface `Exp<?T>` where `?T` should be a type of value. In TyRuBa:

```
interface_M(Exp<?T>) :- value(?T).
method_M(Exp<?T>,?T,eval,[],{
  ?T eval();
}).
```

So with the above instantiation code for the framework this will define two interfaces `Exp<Integer>` and `Exp<Float>`. We might anticipate extending the framework with more complicated expressions later and allow the user to assert what kind of arguments the `eval` method takes. This would for example facilitate an extension of the framework that allows variables in expressions and passes a dictionary with values for variables as an argument to `eval`. To keep this example simple we have not done so and assume that `eval` has no arguments.

The simplest types of expressions are constants. We implement constants by automatically implementing the `Exp<?T>` interface on values of type `?T`. The `eval` method for values simply returns the value itself.

```
implements_M(?T,Exp<?T>) :- value(?T).
method_M(?T,?T,eval,[],{
  public ?T eval() { return this; }
}) :- value(?T).
```

Next we declare the `Factory` class and to this we add methods for creating constants. We assume that a value has a constructor of one argument to initialize it. This allows us to declare the following:

```
class_M(Factory).
method_M(Factory,?T,q,[?Init],{
  static ?T q(?Init init) { return new ?T(init); }
}) :- value(?T),
      feature(?T,constructor<[?Init]>).
```

This declares a class `Factory` with two static methods with signatures `Integer q(int)` and `Float q(float)` for creating constants[1].

Now we will implement classes for compound expressions that apply operations to subexpressions of appropriate types. This will be a little more complicated. We will need to define an expression class for every operation that is defined on every value. We should consider an addition of integers to be a different kind of expression than an addition of floating point numbers so the types of the arguments must become part of the class's name. We can define the classes as follows:

```
class_M(Op<?name,[?A1 | ?Ar]>) :-
  value(?A1),
  feature(?A1,method<?R,op<?name>,?Ar>).
```

Note how we use a pair `[?A1 | ?Ar]` for the types of the arguments. The first argument `?A1` must be a type of value and the remaining arguments `?Ar` must be the arguments of an operation method on `?A1`. Thus, the above rule will declare (amongst others) classes with names `Op<add,[Integer,Integer]>` and `Op<mul,[Float,Float]>`. Note that it is convenient that we have adopted the naming convention that methods for operations on values have a name of the form `op<?x>`. This avoids confusing them with other methods on values such as `eval`.

So far we have only declared the existence of the classes for operation expressions. Now we still need to implement them. They will require instance variables for storing the subexpressions, a constructor for initializing them and an `eval` method for evaluating them. Before showing the TyRuBa code that generates this implementation for any operation with any number of subexpressions, let us first have a look at one particular operation class and

[1]The name of the method, q, stands for "quote".

see what the generated code looks like. Figure 9.4 shows the code generated for the class `Op<add,[Integer,Integer]>`, representing an addition of two integer expressions. For readability we have replaced the mangled names in the Java output by the TyRuBa terms for which they stand. We also indented the code properly and added a few comments. Except for these cosmetic changes the code is presented exactly as generated by the TyRuBa system.

```
class Op<add,[Integer,Integer]>   implements  Exp<Integer>
{
  /** Constructor */
  Op<add,[Integer,Integer]> (Exp<Integer> f<2>,Exp<Integer> f<1>) {
    this . f<2> = f<2> ;
    this . f<1> = f<1> ;
  }
  /** Evaluation */
  public Integer eval (  ) {
    return f<2> . eval (  ) . op<add> (  f<1> . eval (  )  ) ;
  }
  /** Subexpressions */
  Exp<Integer> f<1>  ;
  Exp<Integer> f<2>  ;
}
```

Figure 9.4: Generated code for class `Op<add,[Integer,Integer]>`

Note that the names of the instance variables and formal arguments of the constructor is a bit counter intuitive: they are numbered in reverse order, i.e. `f<1>` is the second subexpression whereas `f<2>` is the first subexpression. The reason for this is that the numbering is generated with a recursive rule that descends a list. It is easiest to start numbering at the end because of the way the list is traversed. Most complications in the generation of the code for the `Op<?name,?Args>` classes is that we want to accommodate an arbitrary number of `?Args`. The formals list for the constructor and the declaration of the instance variables will have to be computed from a list of types. The following auxiliary predicate calculates a list types of the form `Exp<?A`$_i$`>` from a list of types `?A`$_i$. We can use this predicate to calculate a list of types for the instance variables and the arguments of the constructor.

```
CAL_ExpTypes([],[]).
CAL_ExpTypes([?T | ?R],[Exp<?T> | ?ER]) :- CAL_ExpTypes(?R,?ER).
```

As part of one of the initialization files, some predicates for generating and manipulating lists of formals are provided. We have listed the predicates used in this chapter in figure 9.5. The complete initialization file can be found in appendix A. Using these predicates we can now declare the constructor method for an operation expression class as follows:

```
constructor_M(Op<?name,[?A1 | ?Ar]>,?Exp,{
  Op<?name,[?A1 | ?Ar]>(?formals) {
    ?initCode
  }
}) :- class_M(Op<?name,[?A1 | ?Ar]>),
      CAL_ExpTypes([?A1 | ?Ar],?Exp),
/* Generate JavaFormals from a list of types and argument names */
JavaFormals([],[],{}).
JavaFormals([?T|?R],[?n|?r],?formals) :-
   JavaCommaCons({?T ?n},?Rr,?formals),
   JavaFormals(?R,?r,?Rr).

/*Generate JavaFormals from type names only. Argument names are computed*/
JavaFormals(?T,?f) :-
  JavaGenNames(?T,?n),
  JavaFormals(?T,?n,?f).

/*Is a formal an element of a list of JavaFormals? */
JavaFormalsElement({?T ?n},?formals) :-
  JavaCommaCons({?T ?n},?rest,?formals).
JavaFormalsElement({?T ?n},?formals) :-
  JavaCommaCons({?TTT ?nnn},?rest,?formals),
  JavaFormalsElement({?T ?n},?rest).

/* Generate a list of numbered names [f<n>,...f<2>,f<1>] */
JavaGenNames([],[]).
JavaGenNames([?x1],[f<1>]).
JavaGenNames([?x1,?x2|?xs],[f<?c1>,f<?c2>|?rest]) :-
  JavaGenNames([?x2|?xs],[f<?c2>|?rest]),
  +(?c2,1,?c1).

/* "Consing" an element to a comma separated Java list */
JavaCommaCons(?x,{},{?x}).
JavaCommaCons(?x,{?y},{?x,?y}).
JavaCommaCons(?x,{?y,?z},{?x,@{?y,?z}}).

/*Construct a list of comma separated elements*/
JavaCommaList([],{}).
JavaCommaList([?e|?r],?c) :-
   JavaCommaCons(?e,?cr,?c),
   JavaCommaList(?r,?cr).
```

Figure 9.5: Auxiliary Predicates to generate Java code

```
      JavaFormals(?Exp,?formals),
      JavaFormals(?Exp,?names,?formals),
      CAL_initCode(?names,?initCode).
```

This defines a constructor method for every class representing an operation expression. The first call to `JavaFormals` will generate the list of formals for the constructor declaration. The second call is used in "reverse", to extract the generated names from the generated list of formals and bind it to the variable `?names`. This is then passed to the auxiliary predicate `CAL_initCode` to construct the body of the constructor which initializes the instance variables. We will give the instance variables the same name as the names of the formals of the constructor so the predicate `CAL_initCode` is defined as follows:

```
CAL_initCode([],{}).
CAL_initCode([?f|?r],{this.?f=?f; ?ir}) :- CAL_initCode(?r,?ir).
```

Next we define the instance variables of the operation class. For every formal argument of the constructor generated above, we have to define an instance variable of the exact same type and name. We can write this in TyRuBa as follows:

```
var_M(Op<?name,[?A1 | ?Ar]>,?type,?var,{}) :-
  class_M(Op<?name,[?A1 | ?Ar]>),
  CAL_ExpTypes([?A1 | ?Ar],?Exp),
  JavaFormals(?Exp,?formals),
  JavaFormalsElement({?type ?var},?formals).
```

The only thing that remains to be done is to define the evaluation method and declare that the operation class implements `Exp<?T>` of the appropriate type `?T`. The following TyRuBa rule finds out what the return type `?T` of the operation is and declares that the operation implements `Exp<?T>`.

```
implements_M(Op<?name,[?A1 | ?Ar]>,Exp<?T>) :-
      class_M(Op<?name,[?A1 | ?Ar]>),
      feature(?A1,method<?T,op<?name>,?Ar>).
```

Finally we declare the `eval` method for the operation expression.

```
method_M(Op<?name,[?A1 | ?Ar]>,?T,eval,[],{
  public ?T eval() {
    return ?a1.eval().op<?name>(?evalar);
  }
}) :- class_M(Op<?name,[?A1 | ?Ar]>),
      feature(?A1,method<?T,op<?name>,?Ar>),
      CAL_ExpTypes([?A1 | ?Ar],?Exp),
      JavaFormals(?Exp,?instVars),
      JavaFormals(?Exp,[?a1 | ?ar],?instVars),
      CAL_evalcals(?ar,?evalar).
```

It evaluates its first subexpression `?a1` and then calls the appropriate operation method `op<?name>` on the resulting value. The arguments passed to the operation come from evaluating every other subexpression. This list of evaluated subexpressions `?evalar` is computed by the auxiliary predicate `CAL_evalcals`.

```
CAL_evalcals([],{}).
CAL_evalcals([?a1],{?a1.eval()}).
CAL_evalcals([?a1 , ?a2 | ?ar],{?a1.eval(),?rest}) :-
   CAL_evalcals([?a2 | ?ar],?rest).
```

This concludes the declaration of the classes that represent operation expressions. However, we still have to add methods to the `Factory` class to make it easy to create operation expressions. We will use Java's overloading to relieve the user of having to mention the types of the arguments expressions (which are part of the class's name). For the class `Op<add,[Integer,Integer]>` for example the generated factory method will be as follows:

```
static public Op<add,[Integer,Integer]> add (Exp<Integer> f<2>,
                                             Exp<Integer> f<1>   )
{
  return new Op<add,[Integer,Integer]> (  f<2> , f<1>  ) ;
}
```

We may declare the factory methods for all operation expression with the following rule.

```
method_M(Factory,Op<?name,?A>,?name,?CA,{
  static public Op<?name,?A> ?name(?formals) {
     return new Op<?name,?A>(?actuals);
  }
}) :- class_M(Op<?name,?A>),
      feature(Op<?name,?A>,constructor<?CA>),
      JavaFormals(?CA,?formals),
      JavaFormals(?CA,?actualsL,?formals),
      JavaCommaList(?actualsL,?actuals).
```

This concludes our presentation of the framework for now. We can easily generate the needed classes and interfaces to be able to run the following small program which builds an expression, evaluates it and displays the result. The entire code of the framework and the instantiation code, complete with `#generate` directives etc. is listed in appendix B.

```
public class Calculator extends Factory {

  static public void main(String[] args) {
    Exp<Float> e = add(mul(q((float)5.0),q((float)3.3)),
       sub(q((float)4.0),q((float)5.0)));
    float i = e.eval().value;
    System.out.println(i);
  }

}
```

9.5 Adding Booleans

The calculator framework allows instantiation with different types of values in a very straightforward way. As an example we will show how values of type boolean can be added easily. All we really have to do as instantiator is implement a class `Boolean` with the operations we want on booleans, and declare the class `Boolean` to be a type of value for the expression framework.

```
class Boolean {
  boolean value;
  Boolean(boolean init) {value=init;}
  Boolean op<not>() { return new Boolean(!value); }
  Boolean op<or>(Boolean other) {
     return new Boolean(this.value||other.value);
  }
  ...
}
value(Boolean).
```

We can define an `ifThenElse` operation on these booleans that accepts two values of the same type and returns either value dependent on whether the boolean is true or false.

```
method_M(Boolean,?T,op<ifThenElse>,[?T,?T],{
  public ?T op<ifThenElse>(?T thn,?T els) {
    if (value)
       return thn;
    else
       return els;
  }
}) :- value(?T).
```

9.6 Adding Subtyping

We are now going to present an extension to the calculator framework that makes it easy for the instantiator to declare a subtype relationship that may exist between values such as `Integer` and `Float`. The framework should be able to use this to allow operations such as addition, multiplication and subtraction to "lift" an integer, turning it into a float when it happens to be used in a context where a float is expected. As instantiator, we can already accomplish roughly the same result in the existing framework merely by implementing operations of mixed integer and float types on integers and floats. This is tedious however and we have to repeat it for every operation and possibly valid combination of mixed types. The extension of the framework we will present will relieve the instantiator of this tedious labour.

9.6.1 Instantiation of the Framework

Again, we will explain the framework by starting from the instantiator's point of view. We only want the instantiator to go through a minimal effort to establish a subtype relationship between value types. A first idea is to make the subtype relationship of values in the calculator follow the Java subtype relationship between classes implementing values. However this seems impractical since it is for example not natural to make an `Integer` a subclass of `Float`. This would mean that an integer would contain a floating point value in its instance variables. Therefore we have opted to let the instantiator signify a subtype relationship by implementing a conversion method on a type of value which is a subtype of another. The method should have the following signature:

```
?To as<?To>();
```

For example, the framework instantiator can “declare” the subtype relationship which exists between `Float` and `Integer` by implementing an `as<Float>` method on class `Integer` as follows:

```
method_M(Integer,Float,as<Float>,[],{
  public Float as<Float>() { return new Float(value); }
}).
```

We don not wish to burden the framework instantiator with declaring every possible conversion between values. Only between values which have a “direct” link. The transitive (and reflexive) closure of type conversions will be taken into account automatically. For the sake of the example we will therefore introduce `Double`, a third type of value for high precision floating point numbers. The framework instantiator will only provide conversions from integers to floats and from floats to doubles. The conversion from integers to doubles does not have to be explicitly declared since it can be regarded as a composition of the two other conversions. Figure 9.6 gives an overview of all the code provided by the instantiator of the framework.

9.6.2 The Framework Code

We will now present the extensions to the framework that support subtyping in the calculator. Before plunging in and presenting the actual TyRuBa code in full detail, let us first have a look at the Java interface and class hierarchy we intend to generate. We have depicted it in figure 9.7. Because the picture would not fit on a page otherwise we have abbreviated the class names `Float`, `Double` and `Integer` to `F`, `D` and `I` respectively. Class names in the figure are written in **boldface** to distinguish them from interface names.

The classes for operation expressions, `Op<?n,[?V,?V]>`, are defined by the simple version of the framework presented in the previous sections. There is also one new kind of expression represented by the classes `TypeConv<?F,?T>` to do type conversions. The class `TypeConv<?F,?T>` is a wrapper class containing an expression of type `?F` turning it into an expression of type `?T`.

Type conversion expressions are meant to be created implicitly when needed. This will be supported by a smart “subtype aware” factory that accepts anything which is convertible to an expression of type `?T` whenever the simple factory accepts an `Exp<?T>`. The smart factory will apply the necessary type conversions before invoking the simple factory accepting only expressions of a specific type. The purpose of the `Convertible<?T>` interfaces is to provide a common supertype for everything which is directly or indirectly convertible to an expression of

```
class Double {
  double value;
  public Double(double init) { value = init; }
  public op<add>(Double other) { return new Double(this.value+other.value); }
  public op<mul> ...
  ...
}

class Float {
  float value;
  ...
  Double as<Double>() { return new Double(value); }
}

class Integer {
  int value;
  ...
  Float as<Float>() { return new Float(value); }
}

value(Integer).
value(Float).
value(Double).
```

Figure 9.6: Sample Calculator Instantiation Code with Subtyping

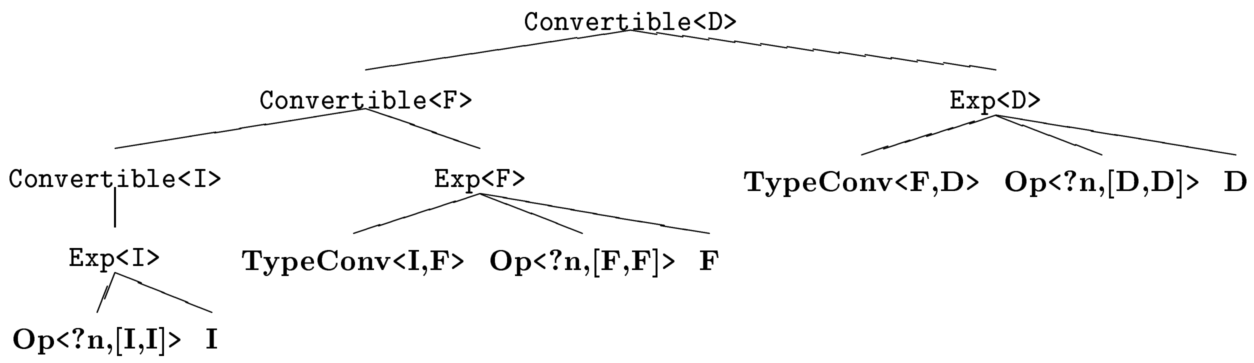

Figure 9.7: Calculator expression **class** and `interface` hierarchy

type `?T`. This will allow constructing the subtype aware factory by simply replacing argument types `Exp<?T>` by `Convertible<?T>`.

We now give a step by step overview and explanation of the TyRuBa code that defines the type hierarchy of figure 9.7.

Type conversion expressions

Whenever the instantiator provides a type conversion method `as<?T>` on values of type `?R`, we declare a class that represents an implicit type conversions from an expression of type `?R` to an expression of type `?T`. The name of this class is `TypeConv<?R,?T>`.

```
class_M(TypeConv<?R,?T>) :-
   value(?R),value(?T),
   feature(?R,method<?T,as<?T>,[]>).
```

This class is a wrapper class around an expression of type `?R`. We will store the "wrapped" expression in an instance variable named `from`.

```
var_M(TypeConv<?R,?T>,Exp<?R>,from,{}) :-  class_M(TypeConv<?R,?T>).
```

The constructor for a `TypeConv` expression has one argument which will be stored in the previously declared instance variable.

```
constructor_M(TypeConv<?R,?T>,[Exp<?R>],{
  public TypeConv<?R,?T>(Exp<?R> init) { from = init; };
}) :- class_M(TypeConv<?R,?T>).
```

An object of the class `TypeConv<?R,?T>` should behave as an expression of type `?T`. Therefore we declare that `TypeConv<?R,?T>` implements `Exp<?T>` and we provide a corresponding implementation for the `eval` method that invokes `eval` on the wrapped expression and converts the returned value to the correct type.

```
implements_M(TypeConv<?R,?T>,Exp<?T>) :- class_M(TypeConv<?R,?T>).
method_M(TypeConv<?R,?T>,?T,eval,(),{
  public ?T eval() {
    return from.eval().as<?T>();
  }
}) :- class_M(TypeConv<?R,?T>).
```

The `Convertible<?T>` interface

The next thing we will do is declare and implement the `Convertible<?T>` interface on all expressions which can be converted to expressions of type `?T`. We will do this in such a way that type conversions on expressions correspond to the transitive reflexive closure of the type conversions explicitly defined on values by the instantiator.

All expressions that are convertible to type `?T` will include a method that returns a converted expression of type `?T`. We therefore define the `Convertible<?T>` interface as follows:

```
interface Convertible<?T> {
  Exp<?T> cast<?T>();
}
```

Note that the conversion method is called `cast<?T>` instead of `as<?T>` to distinguish conversion of expressions from conversion of values. Values are also considered to be (self evaluating) expressions so both an `as<?T>` and a `cast<?T>` method may be defined on them. Therefore the distinct names are important to avoid a name collision.

Next we will implement this interface on the appropriate expression classes and insert it into the type tree as depicted in figure 9.7. Every expression of type `?T` is trivially convertible to its own type. This is reflected in the type hierarchy by making every `Exp<?T>` an extension of `Convertible<?T>`.

```
extends_M(Exp<?T>,Convertible<?T>) :- interface_M(Exp<?T>).
```

Of course, this declaration implies that we should also provide an implementation of `cast<?T>` on every expression of type `?T`. Converting an expression to its own type is simple: merely return the expression itself. We have to insert this trivial conversion method into every class representing an expression of type `?T`. In other words, we have to insert it into every class that implements `Exp<?T>`.

```
method_M(?E,Exp<?T>,cast<?T>,[],{
  public Exp<?T> cast<?T>() { return this; }
}) :- implements_M(?E,Exp<?T>).
```

Now we are going to define the transitive closure of type conversion. We can do this by observing that if there is conversion from values of type `?R` to values of type `?T` then every expression which is `Convertible<?R>` can also be regarded as `Convertible<?T>`. We solidify this observation into the interface hierarchy by declaring `Convertible<?R>` an extension of `Convertible<?T>`.

```
extends_M(Convertible<?R>,Convertible<?T>) :- class_M(TypeConv<?R,?T>).
```

Again we will have to provide a corresponding implementation for the `cast` method. Every class which implements `Convertible<?R>` now also implements `Convertible<?T>` and must have a `cast<?T>` method definition.

```
method_M(?E,Exp<?T>,cast<?T>,[],{
  public Exp<?T> cast<?T>() {
     return new TypeConv<?R,?T>(this.cast<?R>());
  }
}) :- implements1(?E,Convertible<?R>),
      class_M(TypeConv<?R,?T>).
```

A subtype aware factory

Finally, we will provide support for implicit type conversions by defining a new subtype aware factory class `SubtypeFactory`. `SubtypeFactory` will call the methods of `Factory`. The methods in `SubtypeFactory` are more or less identical to those of `Factory`, however, whenever the `Factory` method has an argument of type `Exp<?T>` the method in `SubtypeFactory` will have an argument of type `Convertible<?T>` instead. The code that declares and implements `SubtypeFactory` can be found in figure 9.8. Rather than explain this in detail we just illustrate what it does by showing what the generated code looks like. Figure 9.9 shows part of the generated code for the subtype aware factory. Every method in `Factory` gives rise to a corresponding method in `SubtypeFactory`. The `SubtypeFactory` method passes its arguments on to the `Factory` method after applying the appropriate type conversions.

```
class_M(SubtypeFactory).
method_M(SubtypeFactory,?R,?m,?A,{
  static ?R ?m(?formals) { return Factory.?m(?actuals); }
}) :- feature(Factory,method<?R,?m,?A0>),
      CAL_convertibleTypes(?A0,?A),
      JavaFormals(?A,?formals),
      JavaFormals(?A,?a,?formals),
      CAL_typeConversions(?a,?A0,?actualsL),
      JavaCommaList(?actualsL,?actuals).

CAL_convertibleTypes([],[]).
CAL_convertibleTypes([?A0|?A0r],[?A|?Ar]) :-
  CAL_convertibleType(?A0,?A),
  CAL_convertibleTypes(?A0r,?Ar).

CAL_convertibleType(Exp<?T>,Convertible<?T>).
CAL_convertibleType(?A,?A) :- NOT(equal(Exp<?T>,?A)).

CAL_typeConversions([],[],[]).
CAL_typeConversions([?a|?r],[?A|?R],[?Ca|?Cr]) :-
  CAL_typeConversion(?a,?A,?Ca),
  CAL_typeConversions(?r,?R,?Cr).

CAL_typeConversion(?a,Exp<?T>,{?a.cast<?T>()}).
CAL_typeConversion(?a,?A,?a) :- NOT(equal(Exp<?T>,?A)).
```

Figure 9.8: TyRuBa code for generating `SubtypeFactory`

9.7 Conclusion

In this chapter we have presented a sophisticated example which illustrates how type-oriented logic meta programming opens up tremendous possibilities for genericity. We showed how a class and interface hierarchy can be partially specified by a framework instantiator while the remainder is generated by the framework. In order to “complete the picture”, the framework may extend classes and interfaces provided by the instantiator and it may also define new classes and interfaces. The generated classes and interfaces in this example depend on the

```
class SubtypeFactory {
  static Op<add,[Double,Double]> add (Convertible<Double> f<2>  ,
                                      Convertible<Double> f<1>   )
  {
    return Factory.add( f<2>.cast<Double>() , f<1>.cast<Double>() );
  }

  static Op<add,[Float,Float]> add (Convertible<Float> f<2>  ,
                                    Convertible<Float> f<1>   )
  {
    return Factory.add( f<2>.cast<Float>() , f<1>.cast<Float>() );
  }
  static Op<add,[Integer,Integer]> add ...
  ...
  static Double  q (   double f<1>   ) { return Factory . q (  f<1>    ) ; }
  static Float   q (   float  f<1>   ) { return Factory . q (  f<1>    ) ; }
  static Integer q (   int    f_L1_R ) { return Factory . q (  f_L1_R  ) ; }
}
```

Figure 9.9: Generated code for the `SubtypeFactory` class

instantiator's classes in a non-trivial manner and the full power of a programming language is needed to capture the dependencies between them. Consequently the entire example relies heavily on the expressive power of TyRuBa as a real programming language to manipulate and consult type information in the base level program. Therefore it constitutes a significant piece of evidence which supports our thesis that the type manipulation language should not be restricted for reasons of avoiding ambiguity and non-termination of type checking.

Chapter 10

Aspect-Oriented Programming

10.1 Introduction

This chapter illustrates an interesting and powerful application of the TyRuBa system and type-oriented logic meta programming to support aspect-oriented programming [MLTK97, KLM$^+$97]. The notion of *aspect-oriented* programming is motivated by the observation that there are *aspects* of programs which defy the abstractive capabilities of general-purpose programming languages. These so called aspects cannot be neatly packaged into separate modules or components but cut across the entire program. This harms the readability and maintainability of the program seriously because its source code becomes a tangled mess of instructions and expressions that belong to the implementation of different aspects.

The example we are going to look at in this chapter is the aspect of synchronization. Multi-threaded Java applications can become seriously complicated because of synchronization code which ensures data integrity of data structures which can be accessed by several threads of execution simultaneously. The synchronization aspect is more or less orthogonal to the program's functionality. Nevertheless the synchronization code is spread all over the program thus making it completely unintelligible. Aspect-oriented programming proposes to solve the problem by separating aspect code from the base functionality of the program itself. The aspects are described separately by means of a special purpose *aspect language*. A so called *aspect weaver* generates the actual code, intertwining aspect code with basic functionality code.

As an illustration of the expressive power of TyRuBa we will show that it is well suited to support aspect-oriented programming. Following the treatment of Lopes [LK97] we will implement a subset of the aspect language, Cool, proposed by her to express the synchronization aspect of Java programs separately from their basic functionality. We do not intend to copy all of her work, this would lead us too far and take too much time and effort. Instead we will restrict ourselves to a simplified subset. What we want to draw attention to is the fundamental advantage TyRuBa has over a special-purpose aspect language. TyRuBa comprises a full-fledged logic programming language instead of a restricted special-purpose aspect language. This has a major fundamental advantage because the aspect declarations are represented as facts in a logic program. Aspect declarations therefore can be accessed by means of queries and declared indirectly by means of logic rules, thus enabling *aspect-oriented meta programming*. This increases the expressive power of the "aspect language" tremendously.

10.2 The Synchronization Problem

The problem in writing multi-threaded Java applications is that synchronization code ensuring data integrity tends to dominate the source code completely. As a result it becomes entangled and unmanageable. As an illustration of the problem, consider the implementation of a `BoundedStack` abstract data type which is given in figure 10.1. The figure just lists the "bare bones" version without synchronization code. This code is simple, straightforward and easy to read.

```
class BoundedStack  {
  static final int MAX = 10 ;
  int pos = 0 ;
  Object[] contents = new Object [ MAX ] ;

  public void print ( )  {
     System.out.print("[");
     for (int i=0 ; i<pos ; i++ ) {
        System.out.print(contents[i]+" ") ;
     }
     System.out.print("]");
  }

  public Object peek ( )  {
     return contents[pos];  }

  public Object pop ( )  {
     return contents[--pos];  }

  public void push (Object e)  {
     contents [pos++]=e ;  }

  public boolean empty ( )  {
     return pos == 0 ;  }

  public boolean full ( )  {
     return pos == MAX ;  }
}
```

Figure 10.1: The "bare bones" version of the class `BoundedStack`

The readability of the class `BoundedStack` with synchronization code added is a lot worse. It is even too complicated to fit comfortably onto a single page. Therefore we will only take a look at one of the methods in it. The other methods are messed up in a similar way. Figure 10.2 lists the declaration of the `peek` method, complete with synchronization code.

To implement synchronization at the granularity of methods, a number of counter instance variables will be added to the `BoundedStack` class. One such counter will be declared for each method. A counter instance variable will therefore have a name such as `BUSY_pop`, `BUSY_peek` etc. Code must be added to the start and end of each method to increment and decrement these counters. Also added to the start of the method is a "guard condition" which verifies whether the method may start executing. If the guard is not satisfied the method must wait

```
public Object peek ()  {
    synchronized ( this ) {
      while ( ! (  (  BUSY_pop == 0  ) &&  (  BUSY_push == 0  )   ) ) {
        try { wait ( ) ; }
        catch ( InterruptedException e ) { }
      }
      ++ BUSY_peek ;
    }
    try {
       return contents [ pos ] ;  }
    finally {
      synchronized ( this ) {
        -- BUSY_peek ;
        notifyAll ( ) ;
      }
    }
}
```

Figure 10.2: The `peek` method with synchronization code

for the guard to become true. The `peek` method for example waits until there are no more threads currently executing a `push` or a `pop` method. It is obvious from figure 10.2 that the synchronization code completely dominates the source code: almost all of the code in the figure is synchronization code. Aspect oriented programming solves this problem by providing a special purpose language, called an *aspect language*, with which the synchronization aspect can be described separately from the base functionality. A code generator, called an *aspect weaver* takes a base program without aspects and an aspect program and generates output code integrating both.

We are going to implement a simple subset of the Cool aspect language proposed by Lopes [LK97]. The Cool aspect language is used to specify the synchronization aspect of a Java base program. We are not going to copy the syntax exactly. Instead, we express the synchronization aspect by means of logic facts. The following facts for example declare which methods should not be called concurrently in the `BoundedStack` example.

```
selfExclusive(BoundedStack,push).
selfExclusive(BoundedStack,pop).
selfExclusive(BoundedStack,print).

mutuallyExclusive(BoundedStack,[push,pop,peek]).
mutuallyExclusive(BoundedStack,[push,pop,empty]).
mutuallyExclusive(BoundedStack,[push,pop,full]).
mutuallyExclusive(BoundedStack,[push,pop,print]).
```

Whenever there is a fact `selfExclusive(?c,?m)` this means that the method `?m` of class `?c` should not be started concurrently with itself. If there is a fact `mutuallyExclusive(?c,?methods)` then this means that no method in the list `?methods` may be started concurrently with any other method in the list. A method from a mutually exclusive list may however be started concurrently with itself unless it is declared `selfExclusive` as well. The `peek` method for example is allowed to be executed concurrently with itself, but not with `push` or `pop`. Note that by identifying methods by their names only, we implicitly assume that the base program does not use method overloading. The same simplifying assumption is made by Lopes also. It is not difficult to support overloading, but the example would then become more verbose because the types of the arguments would also have to be listed to identify a method.

Additional guards, other than those derived from the above synchronization declarations, may be added to a method by declaring a fact:

```
requires(?c,?m,?condition).
```

This means that the method `?m` in class `?c` may not be started unless the `?condition` expression evaluates to `true`. The following example declarations ensure that no elements are ever popped from an empty stack nor pushed onto a stack which is full.

```
requires(BoundedStack,push,{!full()}).
requires(BoundedStack,pop,{!empty()}).
```

All of the above facts together form the aspect program that describes the synchronization aspect of the `BoundedStack` class. For easy reference we have listed all of the aspect declarations together in figure 10.3.

```
selfExclusive(BoundedStack,push).
selfExclusive(BoundedStack,pop).
selfExclusive(BoundedStack,print).

mutuallyExclusive(BoundedStack,[push,pop,peek]).
mutuallyExclusive(BoundedStack,[push,pop,empty]).
mutuallyExclusive(BoundedStack,[push,pop,full]).
mutuallyExclusive(BoundedStack,[push,pop,print]).

requires(BoundedStack,push,{!full()}).
requires(BoundedStack,pop,{!empty()}).
```

Figure 10.3: The `BoundedStack` synchronization-aspect program

Declarations of facts `onEntry` and `onExit` can be used to specify synchronization related actions that have to be performed upon entry and exit of a method.

```
onEntry(?class,?method,?statements).
onExit(?class,?method,?statements).
```

The above were not used in the example but will be used indirectly to specify the actions that maintain the counter variables upon entry and exit of a method.

10.3 The Weaver: a Special Purpose TyRuBa Code Generator

The framework of logic meta-programming supported by the TyRuBa system is an excellent medium to implement the Cool aspect weaver. In this section we will explain how we go

```
/** The class BoundedStack */
class_(JCore,BoundedStack).
extends_(JCore,BoundedStack,Object).

/** Instance Variables */
var_(JCore,BoundedStack,int,MAX,{static final int MAX = 10;}).
var_(JCore,BoundedStack,{Object[]},contents,
  {Object[] contents = new Object[MAX];}).
var_(JCore,BoundedStack,int,pos,{int pos = 0;}).

constructor_(JCore,BoundedStack,[],{public BoundedStack()},{}).

method_(JCore,BoundedStack,boolean,full,[],{
   public boolean full()},{
      return pos==MAX;}).

method_(JCore,BoundedStack,boolean,empty,[],{
   public boolean empty()},{
      return pos==0;}).

method_(JCore,BoundedStack,void,push,[Object],{
   public void push(Object e)},{
     contents[pos++]=e; }).

method_(JCore,BoundedStack,Object,pop,[],{
   public Object pop()},{
     return contents[--pos]; }).

method_(JCore,BoundedStack,Object,peek,[],{
   public Object peek()},{
     return contents[pos]; }).

method_(JCore,BoundedStack,void,print,[],{
   public void print()},{
      System.out.print("[");
      for (int i=0;i<pos;i++) {
         System.out.print(contents[i]+" ");
      }
      System.out.print("]");
   }).
```

Figure 10.4: The `BoundedStack` “core” program

about implementing the aspect weaver. As an indication that TyRuBa is very well suited to the task at hand we may mention that it took us less then a day to implement the code generator to support the above aspect declarations.

10.3.1 Layers of Code-to-code Transformations

The first thing we did was take the implementation of the `_M` code generator[1] and refine it a little bit so that method declarations are separated into a method body and a method signature declaration. This makes it easy to "wrap" synchronization code around a method body. The following example declares the bare-bones `peek` method. The TyRuBa representation of the entire base functionality of the `BoundedStack` class is listed in figure 10.4.

```
method_(JCore,BoundedStack,Object,peek,[],
  {public Object peek()},   //signature
  {return contents[pos]; }  //body
).
```

Note that we no longer follow the convention of ending the code generation predicate names with `_X` where `X` is a letter to identify the code generator. Instead, the code generator is identified by an explicit qualifier symbol `JCore`[2] which is added as first argument to every code generation predicate. This facilitates implementing a series of code transformations layered one on top of the other. The final layer is qualified by the symbol `FI` and used to output the finally resulting Java code. The intermediate layers in between are never turned into actual Java code but simply remain represented by `method_(?layer,...)`, `var_(?layer,...)` etc. Because some code to code transformations mostly just copy the layer above and then make some changes to it, a trivial code transformation which simply copies one layer onto the next is provided with the implementation of the code generator. As an illustration we list the rules that define this trivial code to code transformation below. For a complete listing of all of the code generation and transformation rules we refer to appendix C.

```
class_(?dest,?cl) :- copyLayer(?source,?dest), class_(?source,?cl).
interface_(?dest,?cl) :- copyLayer(?source,?dest), interface_(?source,?cl).
implements_(?dest,?cl,?itf) :- copyLayer(?source,?dest),
     implements_(?source,?cl,?itf).
extends_(?dest,?cl,?itf) :- extends_(?source,?dest),
     implements_(?source,?cl,?itf).
var_(?dest,?class,?type,?name,?impl) :- copyLayer(?source,?dest),
   var_(?source,?class,?type,?name,?impl).
method_(?dest,?class,?retType,?name,?argTypes,?head,?body) :-
   copyLayer(?source,?dest),
   method_(?source,?class,?retType,?name,?argTypes,?head,?body).
constructor_(?dest,?class,?argTypes,?head,?body) :-
   copyLayer(?source,?dest),
   constructor_(?source,?class,?argTypes,?head,?body).
```

[1]The `_M` representational mapping reifies classes at member granularity: each individual method, constructor and variable is asserted by a separate proposition.

[2]Named after the JCore language from Lopes's system. In Lopes's system JCore is a simplified version of Java which is used to express the basic functionality without aspect code.

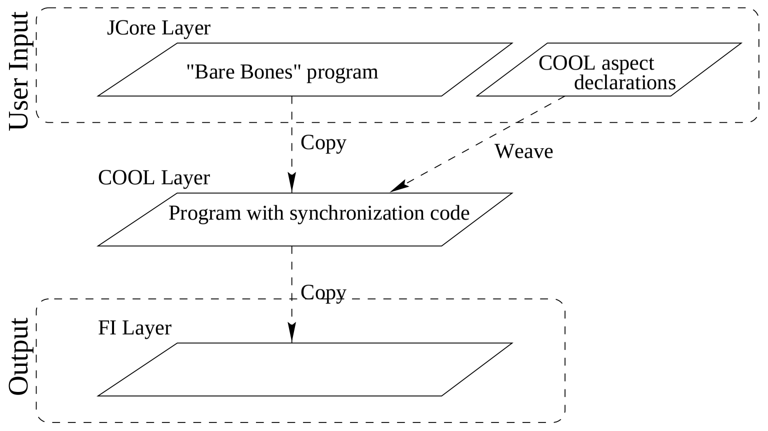

Figure 10.5: The Cool code generator

This implementation is pretty simple[3]. Every code-generation related fact from a layer `?dest` is defined as being true whenever the exact same fact is true in layer `?source`. As an extra condition every one of the rules is guarded by a predicate `copyLayer(?source,?dest)`. This means that we "trigger" a copying transformation from one layer to the next by declaring a fact such as `copyLayer(oneLayer,nextLayer)`.

The setup for implementing the Cool code generator consists of three layers of code on top of each other. The situation is depicted in figure 10.5. The top layer is the `JCore` layer. This is copied to the `COOL` layer. The Cool code generator adds some instance variables and wraps synchronization code around the bodies of the methods where needed. The `COOL` layer is then copied onto the `FI` layer to make this the finally generated output code. We might add more layers in between in order to implement additional aspect languages and weavers. The following two declarations roughly define the layering structure of the code generator.

```
/** First we copy all code from the JCore layer above: */
copyLayer(JCore,COOL).

/** For the time being the COOL layer is the final layer */
copyLayer(COOL,FI).
```

If no other rules or fact declarations are added then the above provides a code generator that copies the JCore program to the `COOL` layer and subsequently copies the code from the `COOL` layer to the `FI` layer. The code which appears on the `FI` layer will finally be used to generate actual output code.

10.3.2 Synchronization Aspect Code

The above `copyLayer` declarations only copy code without changing it. We still need rules that modify the code in the Cool layer to add synchronization aspect code to it.

Before continuing we note that part of the aspect language can be defined in terms of the more low-level features of the aspect language itself. The synchronization code for maintaining

[3]In a full Prolog implementation this could be expressed even shorter by using the "`..`" meta predicate

the counters could be added by means of `onEntry` and `onExit` declarations. Likewise, the conditions that consult the counters to verify whether a method may be started concurrently can be added by means of a `requires` declaration. We therefore first implement support for the more low-level aspect declarations `onExit`, `onEntry`, and `requires`. Afterwards we will implement support for `selfExclusive` and `mutuallyExclusive` easily in terms of the more low-level declarations.

Low-level aspect declarations

The core of the Cool code generator is very simple. Basically it merely adds some wrapper code around the body of a JCore method declaration. Below is the rule which adds wrapper code around a method in the `COOL` layer. The wrapper code should look familiar since it has roughly the same layout as the example synchronization code we presented for the `peek` method in figure 10.2.

```
/** Add wrapper code to COOL methods */
method_(COOL,?class,?Return,?name,?Args,?head,{
    synchronized (this) {
      while (!(?condition)) {
        try { wait(); }
        catch (InterruptedException COOLe) {}
      }
      ?atStart
    }
    try {?body}
    finally {
      synchronized(this) {
        ?atEnd
        notifyAll();
      }
    }
}) :- method_(JCore,?class,?Return,?name,?Args,?head,?body),
      COOL_allRequired(?class,?name,?condition),
      COOL_atStartStatements(?class,?name,?atStart),
      COOL_atEndStatements(?class,?name,?atEnd).
```

A number of auxiliary predicates compute the `?condition` expression and the `?atStart` and `?atEnd` statement lists to be inserted into the template wrapper code.

The auxiliary predicate `COOL_allRequired` collects all of the conditions declared by `requires` aspect declarations for a certain method. All of these are combined into a conjunction, i.e. a list of Java expressions combined together by means of the Java logical and operator “`&&`”.

```
COOL_allRequired(?class,?name,?exp) :-
  FINDALL(NODUP(?cond,requires(?class,?name,?cond)),
          ?cond,?conditions),
  JavaConjunction(?conditions,?exp).
```

The use of `NODUP` avoids duplicate conditions from being included more than once.

The `COOL_atStartStatements` and `COOL_atEndStatements` collect the `onEntry` and `onExit` statements respectively.

```
COOL_atStartStatements(?class,?name,?statements) :-
   FINDALL(onEntry(?class,?name,?stat),
           ?stat,?statements).

COOL_atEndStatements(?class,?name,?statements) :-
   FINDALL(onExit(?class,?name,?stat),
           ?stat,?statements).
```

Mutually exclusive and self exclusive declarations

The rules presented in the previous section implement the core of the Cool code generator which supports the more low-level aspect declarations that add synchronization statements and conditions to the synchronization wrapper code of a method. We can easily provide support for `selfExclusive` and `mutuallyExclusive` declarations in terms of the more low-level declarations.

The following declaration adds the condition that makes sure that a `selfExclusive` method is never started concurrently with itself[4].

```
requires(?class,?name,{COOLBUSY<?name> == 0}) :-
   selfExclusive(?class,?name).
```

This code makes use of a counter variable `COOLBUSY<?name>` which registers how many times a method has been entered. We still have to declare these variables and the `onEntry` and `onExit` code that increments and decrements the counter appropriately. We will come to these declarations later. First let us have a look at the rule that provides the entry condition for a `mutuallyExclusive` method, which also consults a counter variable.

```
requires(?class,?name,{COOLBUSY<?other> == 0}) :-
   mutuallyExclusive(?class,?names),
   element(?name,?names),
   element(?other,?names),
   NOT(equal(?name,?other)).
```

What this rule states is that a guard condition `COOLBUSY<?other> == 0` must be added to a method `?name` whenever `?name` and `?other` are two distinct methods occurring together in a single `mutuallyExclusive` list declaration.

Now, the only thing that remains to be done to complete the Cool code generator is add the counter variables and the administrative code that increments and decrements the counter upon entry and exit. Since the counters are used for verification of mutually exclusive as well as self exclusive conditions we provide a counter variable for every method which is self exclusive or an element of a mutually exclusive list.

[4]The implementation of our simplified code generator also prohibits recursive calls from the same thread. This is usually not the intention. In Lopes's work this is patched by using a more complicated `Lock` object instead of a simple `int` counter. The `Lock` object also records which thread is locking the object and allows calls from the same thread explicitly. We could easily support this more complicated locking strategy. All we need to change are the guard conditions and the declaration of the counter instance variables.

```
var_(COOL,?class,int,COOLBUSY<?name>,{
  private int COOLBUSY<?name> = 0;
}) :- NODUP([?class,?name],
        selfExclusive(?cl,?name);
        mutuallyExclusive(?cl,?mutList),element(?name,?mutList)).
```

The use of NODUP is not strictly necessary because the code generator ignores duplicates anyway. However, it makes the declaration of the onEntry and onExit code simpler. We can simply declare statements to increment and decrement the counters on the condition that a counter variable is declared in the class. Without using NODUP above this could result in the counter being incremented or decremented more than once[5].

```
/** Every method for which there is a COOLBUSY counter
    must get some onEntry and onExit statements to maintain
    the counter */
onEntry(?class,?name,{
        ++COOLBUSY<?name>;
}) :- feature1(COOL,?class,var<int,COOLBUSY<?name>>).
onExit(?class,?name,{
        --COOLBUSY<?name>;
}) :- feature1(COOL,?class,var<int,COOLBUSY<?name>>).
```

This concludes the implementation of our simplified version of the Cool aspect language and code generator. An excerpt from the resulting output code for BoundedStack is listed in figure 10.6.

10.4 Aspect-Oriented Meta Programming

There is a major fundamental advantage to using TyRuBa instead of a special purpose aspect language. The facts declaring the aspects are easily accessible from logic programs and it is possible to indirectly declare aspects by means of logic rules of arbitrary complexity. As a result we can do *aspect-oriented meta programming.* We can write logic programs that infer aspect declarations from other aspect declarations. In fact, we already made use of this potential in the implementation of the code generation for selfExclusive and mutuallyExclusive which was defined in terms of onEntry, onExit and requires.

That this is indeed a considerable advantage and increases the power of the aspect language tremendously will become clear in the following example. As we were experimenting with the BoundedStack example we were not entirely pleased with the way method locking strategies are expressed by means of mutuallyExclusive and selfExclusive declarations. As we reasoned about which methods should be declared selfExclusive we came to the conclusion that these were methods which somehow make a change to some state. Because they make a change to this state they should not be invoked together with any method which also modifies this state, this includes the method itself. Methods which do not modify a state but only inspect it can safely be invoked concurrently with themselves and each other. They should however *not* be invoked concurrently with other methods that modify the same state.

[5]Even this would not be a real problem since we only test for equality to 0.

```
class BoundedStack {

  public void print ( )  {
    synchronized ( this ) {
      while ( ! (  (  COOLBUSY_Lpush_R == 0  ) &&  (  COOLBUSY_Lpop_R == 0  )
               &&  (  COOLBUSY_Lprint_R == 0  )    ) ) {
        try { wait ( ) ; }
        catch ( InterruptedException COOLe ) { }
      }
      ++ COOLBUSY_Lprint_R ;
    }
    try {
      System . out . print ( "[" ) ;
      for ( int i = 0 ; i < pos ; i ++ ) {
         System . out . print ( contents [ i ] + " " ) ;
      }
      System . out . print ( "]" ) ;
    }
    finally {
      synchronized ( this ) {
        -- COOLBUSY_Lprint_R ;
        notifyAll ( ) ;
      }
    }
   }

   public Object peek ( )  {
    synchronized ( this ) {
      while ( ! (  (  COOLBUSY_Lpush_R == 0  )
               &&  (  COOLBUSY_Lpop_R == 0  ) ) ) {
        try { wait ( ) ; }
        catch ( InterruptedException COOLe ) { }
      }
      ++ COOLBUSY_Lpeek_R ;
    }
    try {
     return contents [ pos ] ;  }
    finally {
      synchronized ( this ) {
        -- COOLBUSY_Lpeek_R ;
        notifyAll ( ) ;
      }
    }
   }

   ...other method declarations...

   private int COOLBUSY_Lprint_R = 0 ;
   ...other counter variables...
   private int COOLBUSY_Lpeek_R = 0 ;

   int pos = 0 ;
   Object [ ] contents = new Object [ MAX ] ;
   static final int MAX = 10 ;
}
```

Figure 10.6: Code Generated for `BoundedStack`

This kind of reasoning, about which method modifies and/or inspects what state was the real rationale behind the `selfExclusive` and `mutuallyExclusive` declarations. It would therefore be better if this kind of information can be declared directly and explicitly. Instead of the previously given set of `xxxExclusive` declarations, we would like to write:

```
modifies(BoundedStack,push,this).
modifies(BoundedStack,pop,this).
inspects(BoundedStack,peek,this).
inspects(BoundedStack,empty,this).
inspects(BoundedStack,full,this).
modifies(BoundedStack,print,SystemOut).
inspects(BoundedStack,print,this).
```

The above should provide sufficient information to derive `selfExclusive` and `mutuallyExclusive` properties automatically. We will see that indeed it does and that we can define some simple rules that express how to do so.

The first rule simply states that a method is self exclusive if it modifies some state.

```
selfExclusive(?class,?method) :- modifies(?class,?method,?thing).
```

A method which inspects a state is mutually exclusive with all methods which modify the same state. This is captured by the following rule.

```
mutuallyExclusive(?class,[?inspector|?modifiers]) :-
   inspects(?class,?inspector,?thing),
   FINDALL( NODUP(?method,modifies(?class,?method,?thing)),
            ?method,
            ?modifiers ).
```

We need one more rule to say that all methods which modify the same state should be mutually exclusive with each other.

```
mutuallyExclusive(?class,?modifiers) :-
   NODUP(?thing,modifies(?class,?xxx,?thing)),
   FINDALL( NODUP(?method,modifies(?class,?method,?thing)),
            ?method,
            ?modifiers ).
```

For the `BoundedStack` example this last rule is superfluous. It only infers two redundant facts:

```
:- mutuallyExclusive(BoundedStack,?x).
****START Solutions*****
#SOLUTION : mutuallyExclusive(BoundedStack,[ print ])
#SOLUTION : mutuallyExclusive(BoundedStack,[ pop, push ])
...
```

The first one states that the `print` method is mutually exclusive with itself. This does not make a lot of sense since a mutually exclusive declaration is only useful when it states at least two methods. The declaration doesn't harm though, it just will not result in any code being generated. The second one states that `pop` and `push` are mutually exclusive. This one is also redundant because it is already implied by other inferred facts:

```
:- mutuallyExclusive(BoundedStack,?x).
****START Solutions*****
...
#SOLUTION : mutuallyExclusive(BoundedStack,[ print, pop, push ])
#SOLUTION : mutuallyExclusive(BoundedStack,[ full, pop, push ])
#SOLUTION : mutuallyExclusive(BoundedStack,[ empty, pop, push ])
#SOLUTION : mutuallyExclusive(BoundedStack,[ peek, pop, push ])
```

For other situations the rule might be needed though. For example, for a version of `BoundedStack` which only has two methods, `push` and `pop`, the rule would not be superfluous because without it the methods `push` and `pop` would not be regarded as being mutually exclusive with one another.

10.5 Conclusion

In this chapter we have presented how an aspect language can be embedded in the logic paradigm as a set of well-chosen logic facts. We did this for a simplified subset of the Cool aspect language proposed by Lopes to express synchronization aspects of Java programs. For this particular example it was fairly easy to implement the aspect weaver as a special purpose code generator/transformer.

An equally clean syntactic separation between aspect program and basic functionality can be achieved when using a general purpose declarative programming language such as TyRuBa instead of a special purpose aspect language. In the presented example, one set of facts specifies the basic functionality of the `BoundedStack` class. Another completely disjunctive set specifies the synchronization aspect.

That aspects are expressed by means of a full-fledged logic programming language has a major fundamental advantage over using a restricted special-purpose aspect language. It allows for what we called *aspect-oriented meta programming.* Aspect declarations are merely facts in the logic program and they can therefore be consulted or defined indirectly by means of logic rules. Bluntly put, this means that if you do not like the aspect language the way it is, you can simply define your own language or language extension by means of a few simple rules. As an example we presented and implemented an alternative to the Cool declarations that allows us to capture the reasoning behind the `mutuallyExclusive` and `selfExclusive` declarations in terms of which method inspects or modifies what state. Implementing this extension of the aspect language was very easy, only requiring three simple and intuitive logic rules.

Chapter 11

History and Future of TyRuBa

11.1 Introduction

In this chapter we will give some historic background about the development of the TyRuBa system, and some ideas about further research. This will help the reader in putting the system and its design into the proper context. The system certainly has many points upon which it can be criticized. However, keep in mind that it was mostly designed as a medium for experimentation. Its main purpose is to enable the exploration of the technique of type-oriented logic meta programming, providing evidence that this technique has considerable potential. We wanted to do all this with minimal effort and come to the point of experimentation and validation as soon as possible. Therefore, all was sacrificed in order to obtain maximal expressive power for the type manipulation language. Other aspects, including type checking and syntactic issues, were ignored as much as possible. As a result, the current TyRuBa system has two major shortcomings which make it unusable in a development environment.

The first shortcoming is the lack of a suitable surface syntax. As a consequence of this, the user has to program Java indirectly, describing the base language program by means of logic rules and facts. This is rather inconvenient and makes the TyRuBa programs unnecessarily verbose and difficult to read at times. Section 11.3 discusses this problem in some detail and sketches a prospective solution.

The second major shortcoming of TyRuBa is its inadequate support for type checking. TyRuBa itself does no type checking of its own. It only generates Java code. Type errors are detected no sooner than when the generated code is being compiled by the Java compiler. In section 11.4 we will present some ideas about how to integrate type checking into the TyRuBa system.

The remaining sections discuss various other topics for future research: more additions and improvements to the system/language, potential areas in which type-oriented logic meta programming might be useful, better implementations of the logic inference engine etc.

11.2 History of TyRuBa

We will start by presenting a short summary of the historic evolution of the ideas behind TyRuBa, and the different versions of the system. So far there are four versions of the TyRuBa system/language. In the following discussion, we will refer to these as TyRuBa0.9, TyRuBa1.0, TyRuBa1.9 and TyRuBa2.0. TyRuBa2.0 is the current version, used throughout this dissertation. Whenever we neglect to mention an explicit version number, it is assumed that we are referring to the current version: TyRuBa2.0.

Only two of these four have actually been implemented: TyRuBa1.0 and TyRuBa2.0. The other two systems were never completely implemented but can be considered to be premature versions of their respective successors.

11.2.1 TyRuBa0.9

The first sketchy presentation of the ideas behind TyRuBa was given in [DVM97]. This short position paper presents a fictional extension of a Pizza-like language with a special kind of declaration resembling a logic rule.

Since TyRuBa0.9 was only a sketchy idea and was never concretized and turned into a working system we will not elaborate on it any further. Instead we move on to TyRuBa1.0 which can be regarded as a concrete instantiation of the vague ideas of TyRuBa0.9.

11.2.2 TyRuBa1.0

The TyRuBa1.0 system was the result of an attempt at implementing the TyRuBa0.9 proposal. In order to make the implementation easier a few changes were made to the syntax. Most importantly, the lexical convention of prefixing logic variables with a "?" was adopted.

TyRuBa1.0 is similar to current TyRuBa2.0 in many ways. The major differences are that in TyRuBa1.0 the representational mapping and code generator are hardwired, and that it has a Pizza-like surface syntax which is intimately linked to the representational mapping.

The particular representational mapping adopted in TyRuBa1.0 corresponds most closely to the "_I" mapping from chapter 8. This mapping reifies classes as being composed of pieces which implement a particular interface.

TyRuBa1.0 provides a Pizza-like syntax that allows parametric and bounded parametric types to be expressed easily. An extra extension of the syntax allows expressing logic rules that infer implementations for interfaces onto classes. Let us have a look at some examples illustrating the surface syntax.

Parametric Types

Figure 11.1 lists a small excerpt from one of our early experiments with type oriented metaprogramming: the implementation of a small collection hierarchy in TyRuBa1.0. The figure shows three parametric interface declarations. The only major difference with Pizza parametric types is the lexical convention that type variables start with a "?".

The TyRuBa1.0 parser simply converts these declarations into logic facts, in more or less the same way that parametric types were emulated by means of TyRuBa2.0 in section 7.3. The declaration of the `Searchable` interface for example corresponds to the following logic facts.

```
interface_I(Searchable<?El>,{
   boolean includes(?El e);
}).

extends_I(Searchable<?El>,Collection<?El>).
```

```
/** Interface for *all* collections */
interface Collection<?El> {
  boolean isEmpty();
}

/** Interface for collections with a finite number of elements */
interface Finite<?El> extends Collection<?El> {
  int size();
}

/** Interface for collections you can search for the presence
  of an element */
interface Searchable<?El> extends Collection<?El> {
  boolean includes(?El e);
}
```

Figure 11.1: Some TyRuBa1.0 parametric interface examples.

Bounded Parametric Types

TyRuBa1.0 syntax also has provisions for expressing bounded parametric types. An outline of a TyRuBa1.0 bounded parametric `Set` class declaration is given below.

```
class Set<?El>
whenever ?El implements Equality<?El> ;
implements ...
{
  private LinkedStorage<?El> store = new LinkedStorage<?El>();

  ...
}
```

This corresponds to a class which is declared by a rule, rather than by a simple fact. The condition for the rule is provided in a `whenever` clause.

It would probably have been more logical to use the keyword `where` instead of `whenever`, in analogy with WhereJava [MBL97]. However, the type constraints expressible in TyRuBa1.0 are different from WhereJava's. TyRuBa1.0 type constraints are based on the Java subtype hierarchy whereas WhereJava instead adopts the notion of signature conformance.

Rules to implement interfaces

Inspired by the way qualified types introduce rules into Gofer's type language (see chapter 2), and by the analogy between Gofer's type classes and Java's interfaces, TyRuBa1.0 provides a syntax for declaring rules that infer implementations of interfaces onto classes. The following is an example of such a rule.

```
/** Default implementation for "isEmpty" for all finite collections */
rule ?X implements Collection<?El>
whenever ?X implements Finite<?El>;
```

```
{
  public boolean isEmpty() {
    return size()==0;
  }
}
```

A rule may also target a more specific class rather than use a variable. Thus it is for example possible to declare a “conditional” interface implementation which is only included into a particular parametric class depending on whether some condition on its type parameters is fulfilled. The following example rule implements `Readable` for a parametric class `ArrayStorage` if the elements stored in it are themselves `Readable`.

```
/** Conditionally implement Readable for class ArrayStorage */
rule ArrayStorage<?El> implements Readable<ArrayStorage<?El> >
whenever ?El implements Readable<?El>;
{
  static ArrayStorage<?El> read(BufferedReader in) {
    int size = Number<int>.read(in).value();
    ?El[] initvals = new ?El[size];
    for (int i=0;i<size;i++) {
      initvals[i] = ?El.read(in);
    }
    return new ArrayStorage<?El>(initvals);
  }
}
```

Inherent Restrictions Imposed by TyRuBa1.0

The use of this kind of surface syntax seems preferable over using the logic meta language’s Prolog like syntax directly. The TyRuBa1.0 surface syntax however imposes a number of implicit limitations which we will now discuss.

Because it only offers explicit notations for certain kinds of predicates, the syntax implicitly limits the kinds of predicates that can be expressed. In TyRuBa1.0, for example, the syntax only provides a way to express `implements` predicates. A special, Java-like implements-clause syntax is provided for this purpose. Other kinds of predicates are not expressible however. A consequence of this is that it is nearly impossible for the user to implement or invoke auxiliary predicates.

TyRuBa1.0 does not have the quoting mechanism of TyRuBa2.0. In a way, it does have a similar mechanism *internally*. There is however no syntax that allows the user to make explicit use of it! The quoting mechanism is only used internally, for example when parsing a rule which implements an interface. In this case the body of code that implements the interface will be parsed as quoted Java code. There is however no syntax with which the user may explicitly quote a piece of Java code and subsequently manipulate it.

Another implicit restriction linked with the surface syntax is due to the translation which transforms surface syntax into rules and facts. The translation algorithm is specific to the representational mapping. Therefore, providing a surface syntax translator also implies that the representational mapping be hard coded into the system. The hard-coded representational mapping of TyRuBa1.0 soon turned out to be too coarse grained. An indication of this is the fact that in our very first experiment with the system, an implementation of a small collection library, we got an enormous proliferation of small interfaces. Most interfaces contained only a single method each. There were even a number of "dummy" interfaces which did not contain methods at all but merely served as a kind of "logic tags" attached to classes.

The TyRuBa1.0 surface syntax, its parser implementation, and its code-generator and representational mapping are intimately related to one another. If the mapping is changed or refined, the surface syntax must be revised in order to make it possible to express predicates for accessing the newly reified information. The parser, which translates surface syntax into rules and facts according to the representational mapping, must also be revised. All of this is a lot of work and would seriously slow down and hinder the experimentation process. Realizing this, and also realizing that we probably were not going to find "the ultimate" representational mapping on the next try, we decided to drop the surface syntax altogether, at least for a while, and simply use the underlying logic language directly. This eventually gave rise to the TyRuBa1.9 proposal [DV98].

11.2.3 TyRuBa1.9 and TyRuBa2.0

The TyRuBa1.9 proposal as presented in [DV98] was only implemented partially, but it is highly similar to the current TyRuBa2.0 version. TyRuBa2.0 has already been discussed and used elaborately throughout this dissertation, therefore we will not discuss TyRuBa1.9 and 2.0 any further here.

We directly move on to presenting some ideas about future research, and improvements to the TyRuBa system and its implementation.

11.3 A Surface Syntax for TyRuBa

One of the major shortcomings of the current TyRuBa system is that the user is required to write everything in the logic meta language directly. This has some important drawbacks. The logic representation often exhibits a certain degree of redundancy and is more verbose than the equivalent written in a Java or Pizza-like surface syntax. As an illustration of the problem, consider the following outline of a `Hashtable` class declaration expressed in PJava[1].

```
class Hashtable<?k,?v>
where ?k implements Hashable
{
  public void put(?k k,?v v) { ... k.hashCode() ... }
  public ?v get(?k k) { ... }
}
```

If we express this in TyRuBa's logic language, using the "_M" mapping we get the following:

```
class_M(Hashtable<?k,?v>) :- implements(?k,Hashable).

method_M(Hashtable<?k,?v>,void,put,[?k,?v],{
  public void put(?k k,?v v) { ... k.hashCode() ... }
```

[1] PJava is the fictional parametric types syntax we used in chapter 3.

```
}) :- implements(?k,Hashable).

method_M(Hashtable<?k,?v>,?v,get,[?k,?v],{
  public ?v get(?k k) { ... }
}) :-  implements(?k,Hashable).
```

This is clearly more verbose and less readable than the same thing expressed in PJava. The TyRuBa code also exhibits redundancy in several places. The name of the class for example must be repeated with every method declaration. In PJava on the other hand, the fact that a method belongs to a class is conveniently expressed by nesting. Another example of redundancy is the repetition of the predicate `implements(?k,Hashable)` which expresses a type constraint on type variable `?k`. The constraint must be repeated with every class member because it should affect the entire class declaration as a whole. In PJava this is succinctly expressed by a single where clause attached to the class. A last example of redundancy is the fact that type signatures for methods are expressed twice. Once inside the quoted code block, and once more in the other arguments of the `method_M` predicate.

The above shows that directly expressing things in the logic language is not very practical. Remember however that TyRuBa is an experimental system. A more mature implementation would offer a suitable surface syntax, shielding the user from the problems illustrated by the above example. A parser would read a program written in the surface syntax and translate it into the appropriate rules and facts.

As we explained before we did not design a surface syntax for TyRuBa2.0, and did not implement a parser and translator for it because of two reasons. First, it would have to be changed every time we adapt or refine the representational mapping, something which we did several times in this dissertation! Second, the syntax itself can be an implicit limitation of how the underlying logic system can be used. The latter problem can probably be solved by providing a suitable escape mechanism with which one can always resort to expressing logic rules and facts directly whenever the regular surface syntax falls short.

To give a general idea of what a suitable surface syntax *could* look like, we present a sketchy description of the hypothetical system, FutuRuBa.

11.3.1 A Hypothetical System: FutuRuBa

FutuRuBa is a system designed for allowing type-oriented logic meta programming. It is built around a core implementation, basically corresponding to the TyRuBa2.0 system. The TyRuBa logic language is hidden from the user behind a convenient surface syntax which supports parametric and bounded parametric types. FutuRuBa syntax is parsed by the FutuRuBa parser and translated into TyRuBa rules and facts.

The rules and facts which are produced by the translation are defined in terms of predicates defined by FutuRuBa's representational mapping, and in terms of some predefined predicates from a library of predicates for deriving higher level information from the directly reified information.

We will now briefly discuss the above mentioned components of the FutuRuBa system one by one.

Surface Syntax, Parser and Translation

FutuRuBa has a surface syntax which is based on PJava syntax. It offers the PJava syntax for easy expression of parametric and bounded parametric types. FutuRuBa supports three kinds of where clauses, for expressing different kinds of type constraints.

The first one corresponds to subtyping constraints, following the declared `extends` and `implements` relationship between classes and interfaces. Such a constraint is expressed with an `extends` clause:

```
where ?a extends ?b
```

The second where clause syntax supports signature-conformance constraints as in WhereJava. The syntax for it is similar to WhereJava's:

```
where ?X { ...signatures... }
```

The third where clause syntax is general purpose. It allows any arbitrary TyRuBa query to be used as a constraint. The two former where-clause syntaxes can thus be considered to be mere syntactic sugar for particular uses of this one. The where clause

```
where ?El extends Equality<?El>
```

can for example be written using the general syntax as follows:

```
where :- subtype(?El,Equality<?El>).
```

Any of these where clauses may be attached to a class or interface as a whole, or alternatively, to an individual method, constructor or instance variable.

For situations where FutuRuBa surface syntax is an obstruction rather than a convenience, we can use the keyword `rule` to declare rules or facts directly. The syntax of such a rule is similar to the rules in TyRuBa1.0 (see section 11.2.2).

Representational Mapping

FutuRuBa's representational mapping is a refinement of the "_M" representational mapping which reifies information about individual members of classes and interfaces. FutuRuBa's representational mapping also reifies information about individual constructors, methods and instance variables. In contrast to the sketchy "_M" mapping which still has "holes" in it, in the sense that it does not reify all of the available information, FutuRuBa's mapping is carefully designed to reify all information in a suitable form. This includes:

- The `implements` and `extends` relationship between classes and interfaces.
- Individual members: methods, constructors, instance variables
- Declared type signatures of each member:
  - Argument and return types
  - Exception clauses
  - tags `public`, `private`, `protected`, `static`, ...

- A parse-tree representation for bodies of methods and constructors.
- ...

In order to support the reification of method and constructor bodies as parse trees, FutuRuBa's core system has a more sophisticated quoting mechanism than TyRuBa2.0. In contrast to the string-like quoted code blocks offered by TyRuBa2.0, FutuRuBa quoted code blocks are parsed according to Java syntax, and turned into a parse tree which is represented by nested TyRuBa compound terms. The parse-tree terms behave exactly like normal TyRuBa terms, except that printing them generates Java code.

A Predicate Library for Deduced Information

FutuRuBa provides an extensive library of predefined predicates that derive all kinds of useful information from basic representational mapping facts. The representational mapping only reifies the most basic information about classes and interfaces. What information is reified, is typically that which can be read directly from the base-level program. More high-level information which requires some deduction is typically derived by the rules in the library.

The predicates which are used for expressing type constraints are usually of the more high-level kind. For example, the `subtype` predicate used in the translation of an `extends` clause constraint is defined by a set of rules. These rules define `subtype` in terms of the facts that state which classes or interfaces list which other classes or interfaces directly in their extends or implements clause. The latter information, which can be read directly from base-level programs, is reified by the representational mapping.

In TyRuBa2.0 we already followed the same approach. The initialization files for TyRuBa typically provide an implementation for a code generator, together with a library of predicates which derive higher-level information, such as for example the `subtype` predicate. The TyRuBa version of the library is only a simple proof of concept implementation. It does not take into account `public`, `private` and `protected` tags and how these tags interact with inheritance for example. FutuRuBa's library on the other hand, is carefully crafted to follow the Java semantics precisely. It comprises a complete implementation of a Java type checker. Every little bit of information deduced by the type checker is available through predicates defined in the library. Some examples of what the FutuRuBa predicate library has to offer:

- Take `public`, `private` and `protected` tags into account.
- The Java subtype relationship.
- Testing classes and interfaces for the presence of basic features:
  - instance variables (with type)
  - constructor (with argument types)
  - method (with argument and return types)
- Signature conformance testing.
- Information about which method or class throws what exceptions.
- ...

11.4 Type Checking of FutuRuBa Programs

The second major shortcoming of TyRuBa2.0 is its lack of type checking. In this section we will present some ideas of how it might be feasible to integrate a form of type-checking into FutuRuBa. We will discuss two approaches. The first approach attempts to type check only "nice" programs which do not make sophisticated use of logic meta programming, and leaves more complicated programs unchecked. The second, more ambitious approach, tries to devise a general mechanism for type checking which could also be used to type check more complicated examples.

11.4.1 Determine a Type-Checkable Subset of FutuRuBa

The simplest and most obvious way to introduce *some* type checking into the FutuRuBa system, is establishing a sufficiently conservative subset of FutuRuBa that *can* be type checked relatively easily. Programs or parts of programs which fall into this subset can then be checked, whereas more complicated FutuRuBa code, that makes more elaborate use of the underlying logic meta system, is left unchecked.

It is not hard to argue that there exists a non-trivial subset of all FutuRuBa programs which can be statically type checked. After all, WhereJava and Pizza both can be regarded as restricted versions of FutuRuBa, and type checking algorithms for these languages already exist.

This solution may not be perfect, some programs will not be type checked, but at least programs that could be expressed and type checked before in Pizza and the like, will also be type checked in FutuRuBa. This means that FutuRuBa will be at least as expressive as Pizza or WhereJava, without losing type checking, and if the need arises, type-checking can be abandoned in order to gain extra expressive power.

11.4.2 Type Checking Rules

A more ambitious research goal is to determine a more general mechanism with which arbitrary FutuRuBa programs can be verified. The idea here is to try and do type checking at the level of the logic representation of the program itself, checking rules and facts directly. This is a hard problem indeed. It does not come down to merely defining a static type system for a Prolog like language, but also requires incorporating type checking of potentially partially unknown pieces of Java code. We must realize in advance that given the nature of the representation, a full fledged logic program, it is probably not possible to devise a type-checking algorithm which always terminates for every FutuRuBa program. The challenge here is to find a checking algorithm which terminates most of the time. The least we would require of the algorithm is that type checking of FutuRuBa meta programs which always terminate would also terminate.

Now let us reason a little bit about the most essential part of type checking, the type checking of method bodies. The difficulty lies in checking methods which are defined by rules. These are difficult because they may be partially unknown, depending on a logic variable which is constrained or computed by the condition of the rule. We will try to reduce checking of such a method to ordinary Java type checking. For the sake of the argument let us assume that we have access to a Java type checker, implemented by means of logic rules, that can be invoked by executing the following query:

```
:- typeCorrect(aClassName,aMethodName,anArgumentTypeList).
```

Let us have a look at a typical rule defining a method, and reason about how we can reduce type checking of such a rule to ordinary Java type checking.

```
method_M(Array<?El>,boolean,contains,[?El],{
  boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equals(at(i++));
    return found;
  }
}) :- implements(?El,Equality<?El>).
```

The idea is that a rule like this is considered type correct, if the condition of the rule imposes enough constraints to guarantee that the method defined by this rule is type correct in all possible instantiations. In other words, assuming the fact

```
implements(?El,Equality<?El>).
```

We should be able to deduce that

```
typeCorrect(Array<?El>,contains,[?El])
```

The deduction of the latter fact should make no assumptions about `?El`, other than those provided in the condition of the rule. The deduction should therefore be possible without binding anything to the variable `?El`. In essence, a variable which cannot be bound is no longer truly a variable. We might therefore just replace it by some (unique) identifier to guarantee that it will not be bound to anything. In other words, type checking the rule comes down to assuming the fact (by temporarily adding the assumption to the logic data base for example)

```
implements(El,Equality<El>).
```

and executing the following query:

```
:- typeCorrect(Array<El>,contains,[El]).
```

Note that executing this query comes down to an invocation of a plain Java type checker, on a fully instantiated class `Array<El>`. In other words, this means that type checking of the example rule is effectively reduced to ordinary Java type checking.

Of course, in order for this to work, the `typeCorrect` predicate should be defined in such a way that it knows what a fact such as `implements(El,Equality<El>)` implies. In this case, it should be able to deduce that the type `El` has at least an `equals` method. It will have to be investigated how to structure the implementation of the type checker so that it can deduce what it needs to know from the various assumptions that may occur in conditions of rules. This may occasionally entail that in order to type check arbitrary constraints, expressed by arbitrary user defined predicates, the user may have to write some logic code and plug this into hooks provided by the type checker. It is also interesting to investigate in how far the rules actually defining the user defined predicates may be used to automatically provide type checking support for them.

Of course, there still are a great deal of lose ends and unanswered questions. Nevertheless, we think this example and discussion gives some hope at least that a general type checking mechanism for arbitrary FutuRuBa programs is possible, provided that the user may occasionally be required to implement part of the type checker himself, in order to be able to support arbitrary user defined type constraints.

11.5 Better Integration with Java

Another important point the system can be greatly improved upon, is the integration with the Java language, and its virtual machine and class file format.

Currently TyRuBa is completely separate from the underlying Java language implementation for which it generates code. Ultimately we would want to merge the Java implementation and the TyRuBa meta system into a single tightly integrated whole. The meta system should provide access to all kinds of information typically available in the Java type checker. Currently this is achieved through a library of derived information predicates. This library actually reimplements part of the Java type checker by means of logic rules. Ideally everything — the Java type checker, class reader, compiler, etc. — would be integrated neatly into a single system which can be accessed directly from the logic meta language.

A strongly related point is providing access to existing class libraries, such as for example the standard Java class library. In a system where the Java compiler is tightly integrated with the logic meta system, information about existing classes from existing `.class` files would be reified as part of the type checker. This would allow logic meta programs to access and use classes from existing class libraries easily. In the current system information about existing classes is not reified. We usually worked around this by providing what little information we needed about existing standard Java classes, as part of the initialization files.

The problems discussed in this section are certainly not trivial. Nevertheless, we consider them mostly technical in nature. They could all be fixed by a more sophisticated implementation. The most obvious (but not necessarily most efficient) way to solve most of these problems would be to implement the entire Java compiler as a TyRuBa logic program. Doing this would probably be helpful in solving the type checking problems discussed in section 11.4.2 also.

11.6 Extensions of the Logic Meta Language

The point made in this dissertation is that the static type system of programming language should comprise a type language which is itself designed as a programming language. To show this we illustrated what difference this could make to the expressiveness of the language, by enabling type-oriented meta programming.

The TyRuBa system offers a true programming language with which types and their corresponding implementations can be manipulated at will. However, the TyRuBa logic meta language is still a very simple programming language. Many extensions are still possible and useful to increase its potential even further. A few examples are discussed in the following subsections.

11.6.1 Module System

An extension that would be very useful is a module system for the meta language. This would allow the logic rules to be packaged into modules. Currently we compensate for the lack of a suitable module system by using naming conventions and the primitive mechanism of include files. An important point in the design of a module system for the meta language is the interaction with the base language's module system: Java packages.

11.6.2 Object-Oriented Extensions

It is also interesting to consider using a more object oriented approach, and consider extending the logic meta language with object oriented features that allow incremental modification of meta programs. There are numerous proposals which extend the logic paradigm with object-oriented features [Dav93, McC92, Mos94] which can be used as sources of inspiration.

11.6.3 Reflection and Meta Circularity

So far we have steered clear of adding reflective features to the logic meta language. We did this to avoid the complications entailed by its self referential nature. Nevertheless, it is a natural next step to want to apply the technique of logic meta programming to the TyRuBa meta programs themselves.

To do this we need a reflective system, which offers logic meta representations of the logic meta programs themselves. This leads to a complicated reflective architecture comprising a (virtual) infinite tower of meta representations stacked one on top of the other, each layer representing the layer below it. The bottom layer would be a representation for the Java base language program.

The theory behind such a system and its implementation are very complicated matters. It would require considerable research effort to design and implement this system. For more detailed information about matters related to reflection and infinite towers we refer to [Smi82, Smi84, Mae87, WF88, KdRB91, Ste94b, DVS95].

11.7 Using and Deducing Very-High Level Information

In this dissertation we have focussed on using logic meta programming to generate code. The information used for this purpose was usually rather basic and relatively low level information about types: subtype information, basic type information about methods and instance variables etc.

Another potential application of logic meta programming is as a tool to deduce design information and verify design constraints. The idea of using a logic meta language and its powerful pattern matching and backtracking capabilities for this purpose is also being investigated at our Lab. [Wuy98].

A possible application of this is, for example, detection of design patters [GHJV95]. It can also be used to verify whether certain design decisions are respected as programs evolve. The kind of constraints that can be verified this way are numerous: naming conventions, reuse contracts [Luc97, SLMD96], design patterns, consistency between methods and classes etc.

What we use logic meta programming for in this dissertation is completely complementary to the above. The TyRuBa system is built to generate (Java) code from information, whereas the above suggest using meta programming the other way around, starting from code and deducing or verifying information about it. Ultimately a system should be built which allows both techniques to be used together. It is to be expected that interaction between the two complementary techniques will open up new opportunities for more sophisticated applications. The deduction of more high level information will allow more powerful, more high level and to the point code generation.

11.8 Adaptive Object-Oriented Software

Lieberherr [Lie96] presents a methodology using so called *propagation patterns* to write adaptive code. An adaptive program is written in terms of a partial class structure. It may be used with any concrete class graph that is "compatible" with the partial class structure. Thus the program is adaptive in the sense that it adapts to a concrete class graph. Propagation patterns are a kind of graph constraints which provide information about what assumptions the adaptive code makes about the actual class graph. Writing programs like this, that only make use of partial knowledge about the actual class graph structure, is what Lieberherr calls *class shy* programming. Lieberherr's adaptive programs are customized onto concrete class structures by means of a code generator which inserts appropriate propagation code and wrappers into the concrete classes upon which the adaptive program is *instantiated*.

There seems to be a lot of common ground shared between type-oriented logic meta programming and adaptive object-oriented software with propagation patterns. The techniques differ, but they both provide some mechanism that allows for a form of class shy programming. Perhaps the best illustration of what both have in common is the discussion about "positioning abstract code" we presented in section 8.4.2. We discussed how an arbitrary TyRuBa logic query may be used to characterize the applicability of a piece of abstract code. The problem with "normal" Java programs is that abstract code must be inserted into a specific class. This requires pinpointing an exact location for it in the concrete class tree structure. This in direct contradiction with the fact that abstract code should ignore concrete implementation details about concrete classes as much as possible. In other words, abstract code must be written separately from the actual class tree structure, in a class shy manner. In TyRuBa it is possible to specify abstract code separately from the concrete class tree, and use an arbitrary logic expression to characterize the applicability of the code. In Lieberherr's approach propagation patterns serve the same goal.

The exact relationship between the approach of using propagation patterns or using logic expressions has not been investigated yet. This would be an interesting topic for future research. It is our feeling however that adaptive object-oriented software with propagation patterns could be considered as a particular application of type-oriented logic meta programming. All that is essentially required to introduce support for propagation patterns into TyRuBa is a predefined library of predicates with which propagation patterns can be expressed. By providing such a library the technique of using propagation patterns can be embedded into the more general framework of logic meta programming offered by TyRuBa. We suspect this would bring additional advantages because of the availability of a Turing complete programming language which can always be used when the existing notion of propagation patterns is not exactly what is required in a given situation. In this case, it would for example be possible to implement slight variations of the propagation patterns predicates, in terms of the already existing ones.

11.9 Improving the Logic Engine

11.9.1 More Efficient Implementation

The current logic inference engine is a straightforward Prolog like inference engine. There are two major ways in which it can be improved upon. One is its efficiency, it is currently not very efficient. We did not want to put a lot of effort into building an efficient Prolog interpreter. Many techniques already exist to build efficient Prolog interpreters [Roy94]. Since TyRuBa is essentially Prolog, these techniques could also be used to make the TyRuBa engine more efficient.

11.9.2 Better Support for Recursion

The second and more interesting adaption of the logic engine concerns its mode of inference. Prolog's evaluation mechanism is rather simplistic. Starting out from a goal, Prolog searches backwards from the goal, comparing it with conclusion of rules, and using the rules "backwards", from conclusion to condition . This kind of backward reasoning is called *backward chaining* [Fla94]. Prolog has a very simple strategy concerning which rules to try and in what order to try them: try all rules, exactly in the order in which they were declared. TyRuBa is similar, the only difference basically is that TyRuBa tries the rules in the reverse order. The simplicity of the inference algorithm allows Prolog to be easily implemented, but it also has some drawbacks. A problem we have encountered several times in our experiments is that it does not handle certain kinds of recursive rules well. The prototypical example is the following simple logic program:

```
person(Micky).
person(Minny).
married(Micky,Minny).
married(?x,?y) :- person(?x),person(?y),married(?y,?x).
```

The rule in this program is intended to capture the symmetric nature of the `married` relationship. Saying that a person `?x` is married to a person `?y` is equivalent to saying that `?y` is married to `?x`. The above programs seems perfectly reasonable and intuitively we would expect it to work. Unfortunately a simplistic backward chaining engine does not handle this program as we want it to. Executing a query `married(?x,?y)` for example will cause an infinite loop. The `married` rule is invoked recursively over and over again, each time swapping the variables.

It is also possible to build an inference engine which uses the logic rules the other way around: reasoning from condition to conclusion. This is called *forward chaining.* A forward chaining algorithm maintains a "model", which contains facts which can be deduced using the declarations of the logic program. Initially the model starts out with all the facts of the logic program. Then it verifies whether there are any rules which can be instantiated in such a way that its condition is true (is included in the model) but its conclusion is not. If such a rule is found, then the conclusion is added to the model. This process continues until no more facts can be added to the model. With a forward chaining algorithm the above logic program would be handled quite nicely. The model would start out with the facts

```
person(Micky).
person(Minny).
married(Micky,Minny).
```

In this model, the married rule can be instantiated as follows

```
married(Minny,Micky) :- person(Minny),person(Micky),married(Micky,Minny).
```

The condition of this rule is true. Since the conclusion is not yet in the model it is added. After this one addition to the model no more new facts can be deduced this way and the inference engine stops. Performing a query in such a system simply comes down to matching the query to the facts in the model constructed by the inference engine. Hence in a system with forward chaining, the query `:-married(?x,?y)` would produce the expected answer:

```
married(Micky,Minny).
married(Minny,Micky).
```

From this example, it would seem that forward chaining is more interesting than backward chaining. Unfortunately a simplistic forward chaining algorithm has other important problems. It does not work in a "goal oriented" way and fires rules rather arbitrarily. As a consequence it typically deduces a lot of useless information which is not needed for answering a given query. This is especially problematic when the model which is being constructed is infinitely large. In this case the inference algorithm does not stop. Infinitely large models are not an exception, but occur frequently. Most real logic programs have infinitely large models. As an example consider the following very simple logic program.

```
class_M(Object).
class_M(BarFoo<?c>) :- class_M(?c).
```

In TyRuBa, this declares an empty parametric class `BarFoo`. The condition of the rule says that the parameter of the class may be any declared class. The model that a forward chaining engine would construct would be infinitely large and contain the following facts:

```
class_M(Object).
class_M(BarFoo<Object>).
class_M(BarFoo<BarFoo<Object>>).
class_M(BarFoo<BarFoo<BarFoo<Object>>>).
class_M(BarFoo<BarFoo<BarFoo<BarFoo<Object>>>>).
...
```

A backward chaining engine such as TyRuBa would have no problem with this logic program when we only query it for a specific class. Asking the TyRuBa engine to evaluate the following query for example

```
:- class_M(BarFoo<Object>).
```

will not cause any trouble. A backward chaining engine works backwards from the query and does not construct the infinite model.

Both of the provided examples involved a recursive rule. The one kind is handled better by a forward chaining engine and the other is handled better by a backward chaining engine.

What is needed is some sophisticated combination of backward and forward chaining. Basically the deduction algorithm should be a forward chaining algorithm, constructing proof trees from the bottom up. However, it should still be goal oriented and not simply "blindly" fire rules, but make a founded decision on which rules to fire through some form of backward reasoning. This combined algorithm will have the benefits of a forward reasoning algorithm but would only construct its model "lazily". Thus only the part of the model which is relevant to answering queries will actually be constructed.

We know of an existing rule based system which offers this kind of sophisticated inference engine. This system is ECLIPSE [Hal91] and it is mostly used in the context of expert systems programming. ECLIPSE is a highly sophisticated and efficient goal driven forward chaining inference engine. It is a very interesting topic of future research to study how the ECLIPSE engine or a similar engine, may be substituted for the simple naive and inefficient TyRuBa2.0 engine, and what benefits this would bring with respect to expressiveness.

11.10 Away From Java

In this dissertation we have used TyRuBa as a meta language for the existing class-based object-oriented language Java. An interesting topic of future research is to try and apply the same technique to other base languages as well. The logic meta programming technique is in theory generally applicable to any form of structured textual data.

It would be interesting however to try and design a base language specifically tuned for, and integrated with, the logic meta system. This could increase the potential of type-oriented meta programming even further. Currently there are some restrictions which are more or less inherited from the Java type system, which can not easily be circumvented. Whatever happens in between, eventually ordinary Java code is generated, compiled and type checked. Therefore the type checking process is basically Java type checking. As a consequence, the notion of a "type" in TyRuBa is basically the same as Java's notion of a type: a class or an interface name. As we discussed earlier, ultimately the separation between base-language implementation and meta system should fade away and both should become tightly integrated. This opens up extra opportunities, such as a more general and open ended notion of types defined by arbitrary logic rules.

11.10.1 Arbitrary Rules as Types

Essentially, a type is characterized by a set of values. If a variable is declared as being of a certain type, this means it may only hold values which belong to this set. In a class based language, the finest possible distinction made on the level of the static type system is based on what class a value belongs to. Therefore, a static type in a class based language basically corresponds to a set of classes. A variable of a given type may thus be assigned any value which is an instance of one of the classes in this set.

Considering that we have at our disposal a logic language, what would be the most general and natural way to characterize a set of classes? A good characterization would be an `isa` predicate defined by means of FutuRuBa rules. This `isa` predicate has two arguments. The first one is the name of a type and the second one the name of a class. The predicate should be defined in such a way that it is true if and only if the class is considered an element of the type.

Existing types in Java fit nicely into this notion of a type. Suppose that `X` is the name of a class for example. The type `X` could then be defined by means of the following TyRuBa rule:

```
isa(X,?class) :- subtype(?class,X).
```

Interfaces can also be considered as predicates on classes. The fact that a class is an element of a interface type comes down to satisfying a predicate which verifies that the class provides a number of required method signatures, and declares the interface in an implements clause.

If arbitrary `isa` rules could be used as "type declarations" sophisticated user defined types become possible. The user may state *exactly*, using a fully Turing complete logic language, what properties a class must have in order to belong to a given type. This may have very useful applications and help in solving some problems currently exhibited by the Java type system. We provide a classical example. The example comes from [Sha96] which describes in a rather amusing way why in Java cows are not animals. Cows and animals apparently have incompatible eating habits which cannot be reconciled in Java. The problem is essentially that Java's type system cannot capture exactly what an animal is.

We start presenting the example by giving two concrete animal classes: `Cow` and `Tiger`. Subsequently we try to determine the commonalities between the two and try to define an interface which can serve as a supertype for both. We will see why it is not possible in Java to define a suitable `Animal` supertype for cows and tigers. Subsequently we will show how a type characterized by a logic rule can solve the problem.

```
class Cow {
  public void eat(Grass food) { ... munch munch ... }
}

class Tiger {
  public void eat(Meat food)  { ... crunch crunch ...}
}
```

In this simple example, apparently cows are entities which eat `Grass` and tigers are entities which eat `Meat`. Grass and meat are both a type of food:

```
abstract class Food {
  ...
}

class Grass extends Food {
  ...
}

class Meat extends Food {
  ...
}
```

Now the question is: "What is an animal?". Apparently what animals have in common is that they eat a kind of food. So it would be logical to declare the following interface to describe the type `Animal`.

```
interface Animal {
  void eat(Food food);
}
```

Unfortunately, in Java, the classes `Cow` and `Tiger` cannot be declared to implement this interface because they eat a more specific type of food. Java has what is called *no variance typing*, meaning that method signatures in a subtype must be exactly the same as method signatures in a supertype.

In order to allow `Animal` to be a supertype of `Cow` and `Tiger` we need covariant typing. In covariant typing, method signatures in subtypes may replace argument types by subtypes. Covariant typing however has the problem that it is not type safe. To see this, suppose that an `Animal` *is* a supertype of `Cow` and `Tiger`. This would mean that the following code would be acceptable to the static type checker.

```
  ...
  Animal creature = new Cow();
  Food   dinner   = new Meat();
  creature.eat(dinner); // Runtime type error:
  ...
```

Regrettably, this code will cause a runtime type error when a cow gets to eat meat and chokes on it. The problem is that the interface `Animal` is not a correct characterization of what an animal is. According to this interface, an animal is something which eats *all* kinds of food. This is not usually the case however. A certain type of animal usually only eats a certain type of food.

If we can specify what an animal is by means of declaring an arbitrary TyRuBa `isa` rule, we can be more precise, and state exactly what it means to be an animal. We can define an `Animal` as something which eats a certain type of food:

```
isa(Animal,?class) :-
   AnimalFoodType(?class,?Food),
   feature(?class,method<void,eat,[?Food]>),
   subtype(?Food,Food).
```

Every type of animal has a specific type of food associated to it by means of the `AnimalFoodType` predicate. The following declarations specify the food type that goes with cows and tigers respectively.

```
AnimalFoodType(Cow,Grass).
AnimalFoodType(Tiger,Meat).
```

The above declarations of `Animal` and `AnimalFoodType` define accurately what an animal is: something which eats a specific kind of food. What kind of food depends on the animal in question. Both the classes `Cow` and `Tiger` are classes of type `Animal` according to the above `isa` rule declaration.

Now let us reconsider how to feed an animal. As before, we can define a variable of type `Animal` and assign it an instance of class `Cow`, because a cow is an animal:

```
  Animal creature = new Cow();
```

Similarly we can declare a variable of type `Food` and assign it an instance of the class `Meat`:

```
  Food   dinner   = new Meat();
```

The following however should be rejected by the static type checker:

```
  creature.eat(dinner); // Not allowed
```

The reason why the type checker[2] should reject it is that the type `Animal` implies insufficient assumptions to allow it to be fed with something of type `Food`. The assumptions the type checker may make about a `?class` of type `Animal` are found in the body of the `isa` rule defining the `Animal` type:

```
   AnimalFoodType(?class,?Food),
   feature(?class,method<void,eat,[?Food]>),
   subtype(?Food,Food).
```

In plain English we can restate these assumptions as follows: "There is a type `?Food`, which is a subtype of `Food`. The class `?class` has a method `eat` which accepts the type `?Food` as argument." There is no way to conclude from these assumptions that `?class` has an `eat` method which accepts an argument of type `Food`. We can only assume that the `eat` method accepts some subtype of `Food`, but this does not imply that it accepts any kind of `Food`.

This means we cannot feed animals without somehow determining the specific type of food that this particular type of animal eats. As an example we show how to implement a section of a zoo where animals are being fed. There will be several such sections in a zoo, each section providing a specific type of food. Only animals which eat the specific kind of food provided in a particular section will be allowed to eat there.

```
class FeedingSection<?Food>
where ?Food extends Food
{
   void feed(EaterOf<?Food> animal) {
      ?Food food = new ?Food();
      animal.eat(food);
   }
}
```

The type `EaterOf<?Food>` is once more a type defined by an `isa` rule:

```
isa(EaterOf<?Food>,?class) :-
   isa(Animal,?class),
   AnimalFoodType(?class,?Food).
```

Thus the type `EaterOf<?Food>` corresponds to a set of classes such that every class in the set is some kind of animal that eats food of type `?Food`.

[2]This would be the kind of type checker discussed in section 11.4.2.

11.10.2 Virtual Types, Families of Types

In chapter 3 we presented an overview of existing parametric types proposals for Java. We also discussed some alternatives to parametric types: virtual types and Bruce's alternative to virtual types. Virtual types and parametric types seem to be mostly complementary. It was shown that parametric types can be emulated in TyRuBa quite naturally. So far we haven't said much about virtual types and the like however. It turned out that examples which are typically handled well by virtual types, but are difficult with parametric types cannot always be easily expressed in TyRuBa.

Virtual types are much harder to emulate in TyRuBa. The cause of this however is basically that TyRuBa still generates ordinary Java code and thus basically only has Java's simplistic notion of a type as a class or interface name. To explain this a little further, we use the example of cows tigers and animals once more. This is a typical example which is expressed easily with virtual types, but hard with parametric types. Virtual types are attributes of classes which are themselves types. These attributes may be overridden in subclasses. The type of food is typically a virtual type attribute of the class `Animal`. A subclass of `Animal`, such as `Cow` or `Tiger` may override the type of food to a more specific type. Thus this example can be easily expressed with virtual types. As was illustrated by the discussion in the previous section, the generalized notion of a type as a set of classes characterized by an arbitrary `isa` rule declaration would also allow expressing this example quite naturally. The latter solution is however more general than a virtual-types-like solution. Emulation of virtual types is but a particular usage of the power of the logic paradigm to capture relationships between types. In this example, `AnimalFoodType` fact declarations were used to establish the relationship between a type of animal and the type of food that it eats.

11.11 Summary

In this chapter we started by presenting a short historic overview of the development of TyRuBa up to the current TyRuBa2.0 version. It was argued that many of the shortcomings of the system are justifiable because of the experimental goals behind its design and implementation.

Subsequently we discussed how a hypothetical future version of TyRuBa, FutuRuBa could try to solve these problems. Not all of the problems will be solved easily. But at least workable partial solutions can be found, given more time and resources to do further research. The first addition to the system would be a suitable surface syntax which would make FutuRuBa programs more succinct and readable than current TyRuBa programs which are often verbose and exhibit redundancy.

Another important point is the issue of type checking which was neglected in TyRuBa2.0. In part, this lack of type checking helped in increasing the system's expressiveness. In a real development environment however, type checking is a must. A solution which would be workable is type checking "nice" more conservative programs and allow more sophisticated uses of the logic meta-system to go unchecked. A more ambitious goal is defining a general type checking scheme which can also deal with the more complicated cases. We have hopes that this is possible, even though it will definitely require serious research effort.

In the final section of this chapter (section 11.10) we let our imagination run free and mused about designing the base and meta-language specifically tuned for each other. In our opinion this could be the basis for "the type system of the future". The type system of the future should incorporate a fully Turing equivalent programming language into its type language. Preferably this language would be a declarative logic language. However, a naive Prolog like implementation seems too simplistic and inefficient for a practical system. A more sophisticated and efficient inference engine based on a combination of goal directed reasoning and backward chaining would be required. The type system of the future would be a fully open ended system that allows types to be characterized by arbitrary predicates implemented by means of arbitrary rules. Existing notions of types such as subtyping, interfaces, signature conformance etc. can all be incorporated elegantly into such a system by means of a number of predefined rules. The user may add rules of his own. In doing so, he would be able to actually use any set of classes that can be characterized by any Turing-computable predicate as a type.

Chapter 12

Conclusion

12.1 Thesis

In this dissertation we have illustrated that undecidable and ambiguous type systems have useful applications and should therefore be considered viable options for future statically typed object-oriented languages. This statement can probably be made about static type systems in general. This dissertation however focuses on object-oriented languages.

The usefulness of a type system which is undecidable and which is less concerned with avoiding ambiguity, will stem from its greatly enhanced expressive power. The type language of such a system can be designed as a true programming language. This allows static types and type information to be manipulated and consulted by "real" type programs which are run as part of the type checking or inferencing process. As a result, static types can play an active role in programs and become useful in ways beyond their traditional descriptive role.

12.2 Evidence for the Thesis

Chapters 2 through 9 represent the body of the dissertation, presenting arguments to support the thesis. Basically there are three steps in our "proof". We briefly summarize and discuss these steps in the following subsections.

12.2.1 Existing Type Systems and Their Restrictions

We started by illustrating the loss of expressiveness incurred by existing type systems because of their type language's deficiency as a programming language. We discussed some existing state of the art type systems in chapters 2 and 3. Chapter 2 presented and discussed Gofer and its type system because we think that Gofer is a language which comes near to our ideal of having a true programming language for a type language. Chapter 3 gave an overview of existing parametric types proposals for Java. These were discussed mainly because of their relevance to our work as being steps in the direction of type-oriented meta programming in the context of Java.

Gofer as well as the Java parametric types extensions were assessed by means of a few simple benchmark programming assignments aimed at probing the expressiveness of the type language as a programming language. Attempts at implementing these benchmarks in existing parametric types proposals were seriously hampered by restrictions inherent to their type languages. As a result, existing parametric types extensions for Java performed poorly on the benchmarks and almost none of them could be expressed adequately. The functional language Gofer, the type language of which is much more sophisticated, performed a lot better. However, it did not solve all of the benchmark problems adequately either. We consider the language Gofer to come very close to our ideal of having a programming language embedded in the type system. Gofer's qualified types introduce a kind of logic rules into the type system. In a way this represents a restricted logic language. Gofer's type system is not even decidable although this does not seem to pose problems in practice. Nevertheless, Gofer's type language is not designed as a programming language and still has a lot of implicit and explicit restrictions built into it. This is the underlying reason why it falls short on solving some of the benchmark problems.

12.2.2 Building an Experimental System

The next step in the "proof" is the construction of an experimental system which makes it possible to illustrate the usefulness of type oriented (logic) meta programming. The most logical approach would have been to design and implement a programming language, with an undecidable and potentially ambiguous type system, which has a full-fledged logic language embedded in its type language. Subsequently examples of its usefulness could be presented, and thus the truth of the thesis established. Unfortunately, due to limited time and resources we were unable to follow this approach exactly and had to cut a few corners. We did not design and implement a true type system. The system we built, TyRuBa, was described in chapter 6. It does not perform any type-checking of its own, but generates Java code and leaves type checking to the Java compiler which is used to compile the generated code. What the system does offer however, is a full-fledged logic programming language, with which static type information can be reified and manipulated at compile time. Chapter 7 showed how the TyRuBa system can be used to reify static type information and can thus be used to achieve type-oriented logic meta programming.

As was argued, the extra usefulness of having a true (logic) programming language as part of the static type system is complementary to the traditional descriptive role of types. The *added* potential of such a system does not come from type checking, but rather from the power of the type language to actively manipulate static types and type information as data at compile time. Because the traditional type-checking role of a type system and the extra potential of active type manipulation by compile-time meta programs are completely complementary, neglecting the one does not preclude illustrating the potential usefulness of the other. Therefore, the TyRuBa system is sufficient to prove our point.

12.2.3 The Expressiveness of Type-Oriented Logic Meta Programming

The final step in the "proof" is showing that effectively, the availability of a full-fledged compile-time logic meta language to manipulate static type information *is* very useful and *does* offer tremendous possibilities.

Chapter 8 showed that the expressiveness of the TyRuBa system is considerably greater than that of traditional type languages. The TyRuBa system was compared to Gofer and existing parametric types proposals for Java. The benchmark programming assignments which were previously used to asses the expressiveness of Gofer's type language and of Java parametric-types extensions are used as a basis for this comparison. The fact that TyRuBa is able to express all of the benchmark problems adequately is a strong indication that the availability of a true programming language, to manipulate static types with, does represent a considerable gain in expressiveness, and that this gain can be put to good use in solving real programming problems.

Because the benchmark problems are only small programming tasks, chapter 9 presented a somewhat larger scale example illustrating the usefulness of type-oriented logic meta programming in the context of the implementation of a calculator-expression evaluator framework. This example makes sophisticated use of the technique of type-oriented meta programming to generate part of the framework's class hierarchy. The instantiator of the framework provides classes which implement the values handled by the calculator. From this, the expression class and interface hierarchy is generated by the framework.

12.3 Aspect-Oriented Logic Meta Programming

Chapter 10 presents a sophisticated example of using the TyRuBa system, beyond type-oriented logic meta programming. It was shown in this chapter how TyRuBa can be used to support aspect-oriented programming and aspect-oriented meta programming.

A type language and an aspect language have a lot in common. They are both special-purpose declarative formalisms for expressing program annotations. In a way, aspect declarations can often be regarded as a kind of high-level domain-specific type annotations. Therefore our arguments also apply to aspect-oriented programming. Aspect languages, just like type languages, are typically restricted declarative languages which do not offer the full power of a programming language. We feel that an aspect language should be embedded in a general purpose (logic) meta-programming language. This will facilitate aspect-oriented (logic) meta programming.

As an illustration of the potential of this approach, we implemented a subset of the aspect language Cool as proposed by Lopes [LK97]. We did not copy all of her work because this would have taken too much time and effort. Instead we restricted ourselves to a simplified subset. What we wanted to draw attention to is the advantage of using a full-fledged logic language. The Cool aspect declarations are represented as logic facts and can therefore be accessed and declared by logic rules. The fundamental advantage this offers is that it enables *aspect-oriented logic meta programming*, thus increasing the expressiveness of the "aspect language" tremendously. We showed how aspect-oriented meta programming can for example be used to extend or adapt the aspect language "on the fly" in order to better suit a particular situation.

12.4 Summary of the Contributions Made by this Dissertation

We feel the most important contribution of this dissertation is in pointing out that limitations imposed on static type systems by requiring decidability and ambiguity of typing should be reconsidered. We feel that currently existing programming languages adopt these restrictions as prerequisites without much thought. The restrictions were very sound in the early days of type systems when the role of static types was mostly descriptive. However, as type systems naturally evolve towards more sophisticated "programming-language-like" type languages, static types are starting to play an additional, complementary and more active role. Because decidability and ambiguity of typing have been considered prerequisites for static type systems for so long, we feel that they are hardly being questioned any more. In our opinion, in light of the importance of active usage of static type information, it should be considered to design type languages as true programming languages. As a consequence the restrictions implied by decidability and unambiguity of typing should be reconsidered. The choice of whether or not the expressiveness of the type language is more important than decidability and unambiguity of typing depends on the particular purpose for which one designs a particular programming language. This choice should be made with a critical mind however, and it should *not* be taken as a prerequisite that *every* type system should be decidable and unambiguous. We think that this dissertation has provided sufficient arguments and examples to show that designing a type-language as a programming language does represent a tremendous potential gain in expressiveness, and that this expressiveness does have important advantages and applications.

Another contribution made by this dissertation is the definition of a conceptual framework of logic meta programming. We explained our notion of logic meta-programming in terms of a representational mapping which views a base-language program as a set of logic propositions. The definition of this conceptual framework has allowed us to build TyRuBa. TyRuBa is a simple but remarkably flexible experimental system with which we were able to illustrate the currently unharvested potential of writing logic meta programs which manipulate static type information at compile time.

Finally, we think that the ideas presented in this dissertation can be applied in a much broader context than merely type systems and static types. In principle any restricted declarative formalism to express program annotations is subject to the same kind of reasoning. In essence, static types are simply a particular kind of program annotations. We have illustrated how the same line reasoning applies to the aspect language Cool. We feel that we have thus made valuable contributions to the aspect-oriented programming community in two ways. First by illustrating that aspect-oriented (logic) meta programming is an interesting idea and that therefore it should be considered to embed aspect declarations in a full-fledged declarative programming language rather than expressing them in a restricted special purpose language. Second, by showing that a system like TyRuBa, offering a flexible logic meta-programming framework, is a powerful tool for implementing aspect weavers, and a first step in the direction of the implementation of a general framework for aspect-oriented programming. This was illustrated by the remarkable simplicity with which we were able to implement the simplified variant of the Cool aspect weaver.

Appendix A

Benchmark Examples Code

The following code is also available from `ftp://progftp.vub.ac.be/languages/TyRuBa/`.

A.1 Initialization Files

File: examples/initfile.rub

```
#include "prolog.rub"
#include "java.rub"
#include "codegeneration.rub"
#include "codegenerationI.rub"
#include "deduced.rub"
```

File: examples/prolog.rub

```
/* A few useful Prolog predicates */

append([],?X,?X).
append([?X | ?XS],?YS,[?X | ?ZS]) :- append(?XS,?YS,?ZS).

element(?E,[?E | ?R]).
element(?E,[?A | ?R]) :- element(?E,?R).

length([],0).
length([?x|?r],?i) :- length(?r,?iminus1),+(1,?iminus1,?i).

equal(?x,?x).
```

File: examples/java.rub

```
/***** Auxiliary predicates for constructing Java syntax */

/*Create an implements or extends clause (?symbol indicates which) */
JavaClause(?symbol,[],{}).
JavaClause(?symbol,[?f|?r], {?symbol ?itflist})
```

```
 :- JavaCommaList([?f|?r],?itflist).

/*Construct a list of comma separated elements*/
JavaCommaList([],{}).
JavaCommaList([?e|?r],?c) :-
   JavaCommaCons(?e,?cr,?c),
   JavaCommaList(?r,?cr).

/*Construct a list of Semi-colon terminated elements */
JavaSemiList([],{}).
JavaSemiList([?f | ?r],{?f; ?sr}) :- JavaSemiList(?r,?sr).

/*Generate a list of formals from types and variable names*/
//JavaFormals(?Types,?Names,{...}).
JavaFormals([],[],{}).
JavaFormals([?T|?R],[?n|?r],?formals) :-
   JavaCommaCons({?T ?n},?Rr,?formals),
   JavaFormals(?R,?r,?Rr).

/*Generate a list of formals from type names */
JavaFormals(?T,?f) :-
  JavaGenNames(?T,?n),
  JavaFormals(?T,?n,?f).

/*Generate a list of formals with type {?Type a<?Name,?counter>}*/
JavaFormals(0, ?Type, ?Name, {}).
JavaFormals(?Dim, ?Type, ?Name, ?formals)
 :- >(?Dim,0), +(?DDim,1,?Dim),
     JavaFormals(?DDim, ?Type, ?Name, ?RestFormals),
     JavaCommaCons({?Type a<?Name,?Dim>},?RestFormals,?formals).

/*Is a formal an element of a list of JavaFormals? */
JavaFormalsElement({?T ?n},?formals) :-
  JavaCommaCons({?T ?n},?rest,?formals).
JavaFormalsElement({?T ?n},?formals) :-
  JavaCommaCons({?TTT ?nnn},?rest,?formals),
  JavaFormalsElement({?T ?n},?rest).

JavaGenNames([],[]).
JavaGenNames([?x1],[f<1>]).
JavaGenNames([?x1,?x2|?xs],[f<?c1>,f<?c2>|?rest]) :-
  JavaGenNames([?x2|?xs],[f<?c2>|?rest]),
  +(?c2,1,?c1).

JavaCommaCons(?x,{},{?x}).
JavaCommaCons(?x,{?y},{?x,?y}).
JavaCommaCons(?x,{?y,?z},{?x,@{?y,?z}}).
```

```
/*Generate code for a method declaration without the body */
JavaMethod(method<?R,?m,?A>,?a,{?R ?m(?formals)}) :-
  JavaFormals(?A,?formals).

/*Generate a "Conjunction": (A1) && (A2) && (A3) ... */
JavaConjunction([?a],{(?a)}).
JavaConjunction([?a1, ?a2 | ?r],{(?a1) && ?rest}) :-
  JavaConjunction([?a2 | ?r],?rest).
```

File: examples/codegeneration.rub

```
/************************************************************************
The TyRuBa code generator calls the query
generate(aClassOrInterfaceName,?code)
This allows the code generator to be implemented in
TyRuBa itself.

This file provides a code generator for a very coarse representational
mapping that splits up Java source as shown in the following
Schematic example:

=== Java ===
class CLASS
extends SUPER
implements INTERFACE1,INTERFACE2
{ CLASSBODY }
=== TyRuBa ==>
class_(CLASS,{ CLASSBODY }).
extends_(CLASS,SUPER).
implements_(CLASS,INTERFACE1).
implements_(CLASS,INTERFACE2).
=============

=== Java ===
interface ITF
extends INTERFACE1,INTERFACE2
{ ITFBODY }
=== TyRuBa ==>
interface_(ITF,{ ITFBODY }).
extends_(ITF,INTERFACE1).
extends_(ITF,INTERFACE2).
=============

Note: We adopt as a convention that predicates that are "part of" the
representational mapping and directly affect generated java code end with
an "_"
```

```
This file provides the coarsest representational mapping. It represents
the class and interface body as an atomic unit. More fine grained
mappings can be obtained making the class_ predicate to be
computed from a more fine grained specification.

*********************************************************************/

generate(?class,{
class ?class
?extendsclause
?implementsclause
{ ?body }
}) :- class_(?class,?body),
      generate_extendsclause(?class,?extendsclause),
      generate_implementsclause(?class,?implementsclause).

generate(?itf,{
interface ?itf
?extendsclause
{ ?body
}
}) :- interface_(?itf,?body),
      generate_extendsclause(?itf,?extendsclause).

generate_extendsclause(?x,?extendsclause) :-
  FINDALL(NODUP(?extended,extends_(?x,?extended)),
          ?extended,?extendslist),
  JavaClause(extends,?extendslist,?extendsclause).

generate_implementsclause(?x,?implementsclause) :-
  FINDALL(NODUP(?itf,implements_(?x,?itf)),
          ?itf,?implementslist),
  JavaClause(implements,?implementslist,?implementsclause).
```

File: examples/codegenerationI.rub

```
/***********************************************************************
A refinement of the "_" code generator. This code generator chops
class bodies up into smaller parts implementing interfaces.
Predicates related to this code generator follow the naming convention of
ending with "_I".
***********************************************************************/

class_(?class,{?basics ?interfaces}) :-
  class_I(?class,?basics),
  FINDALL(NODUP(?itf,implements_I(?class,?itf,?how)),
          ?how,?interfaces).
implements_(?cl,?itf) :- implements_I(?cl,?itf,?bod).

/* Predicates implements_ extends_ and interface_ retain the same
   meaning as before. We provide renamed versions of them
   ending in "_I". Therefore the former "_" versions should no longer
   be used directly */
extends_(?a,?b) :- extends_I(?a,?b).
interface_(?itf,?bod) :- interface_I(?itf,?bod).
```

File: examples/deduced.rub

```
/**** Predicates that deduce information from "_" facts */

/* Is there a class named ?cl */
class(?cl) :- class_(?cl,?basics).

/* Is there an interface named ?itf */
interface(?itf) :- interface_(?itf,?body).

/* Does a class implement an interface (directly or indirectly) */
implements(?cl,?itf) :- implements_(?cl,?itf).
implements(?cl,?itf) :- BOUND(?itf),extends(?itf0,?itf),
                        implements_(?cl,?itf0).
implements(?cl,?itf) :- BOUND(?cl),NOT(BOUND(?itf)),
                        implements_(?cl,?itf0),extends(?itf0,?itf).
implements(?cl,?itf) :- extends_(?cl,?super),implements(?super,?itf).

/* Does ?cl extend ?super (?cl class or interface) */
extends(?cl,?super) :- extends_(?cl,?super).
extends(?cl,?super) :- extends_(?cl,?super0),extends(?super0,?super).

/* Is ?cl a subclass of ?super */
subclass(?cl,?super):-extends(?cl,?super),class(?cl).

/* Is ?t1 a subtype of ?t2 according to Java type rules */
subtype(?t1,?t2) :- extends(?t1,?t2);implements(?t1,?t2).
subtype(?t,?t) :- class(?t);interface(?t).
```

A.2 Benchmark Solutions

A.2.1 Benchmark 1 and 2

File: examples/SearchableArrayTest.jrub

```
#verbatim {package aRuBa.tyRuBa.examples;}

#include "Equality.rub"
```

```
#include "Searchable.rub"
#include "Enumerable.rub"
#include "Array.rub"

#include "AbstractSearchable.rub"
#include "SearchableArray.rub" //Overrides the abstact implementation
                               //comment out to
                               //test the abstract implementation.

implements_I(Array<?El>,Enumerable<?El>,{
  public Enumeration<?El> elements() {
    return new Enumeration<?El>() {
       int pos = 0;
       public boolean hasMoreElements() {
         return pos<length();
       }
       public ?El nextElement() {
         return elementAt(pos++);
       }
    };
  }
}).

#generate Searchable<String>
#generate Enumerable<String>
#generate Enumeration<String>
#generate Array<String>

#verbatim {

public class SearchableArrayTest {

  static final int size = 3;

  public static void main(String[] args) {
    Array<String> box = new Array<String>(size);
    for (int i=0;i<size;i++) {
      box.setElementAt("("+i+")",i);
    }
    for (int i=0;i<size;i++) {
      System.out.print(box.elementAt(i));
    }
    System.out.println();

    String e = "(1)";
    System.out.println("contains \""+e+"\"="+box.contains(e));
    e = "(2)";
    System.out.println("contains \""+e+"\"="+box.contains(e));
    e = "asads)";
    System.out.println("contains \""+e+"\"="+box.contains(e));
    e = "(0)";
    System.out.println("contains \""+e+"\"="+box.contains(e));
    e = "(rrasads)";
    System.out.println("contains \""+e+"\"="+box.contains(e));

  }
}

}
```

File: examples/Equality.rub

```
interface_I(Equality<?this>,{
  boolean equals(?this e);
}).

implements_I(String,Equality<String>,{}).
```

File: examples/Searchable.rub

```
interface_I(Searchable<?El>,{
  boolean contains(?El e);
}).
```

File: examples/Enumerable.rub

```
interface_I(Enumerable<?El>,{
  Enumeration<?El> elements();
}).

interface_I(Enumeration<?El>,{
  boolean hasMoreElements();
  ?El nextElement();
}).
```

File: examples/Array.rub

```
class_I(Array<?El>,{
  private ?El[] contents;

  /** Construction */
  Array<?El>(int sz) { contents = new ?El[sz]; }

  /** Basic Array functionality */
  ?El elementAt(int i) { return contents[i]; }
  void setElementAt(?El e,int i) { contents[i]=e; }
  int length() { return contents.length; }
}).
```

File: examples/AbstractSearchable.rub

```
/* Abstract implementation of Searchable on top of Enumerable */
implements_I(?X,Searchable<?El>,{
 public boolean contains(?El e) {
    boolean found = false;
    Enumeration<?El> elems = this.elements();
    while (!found && (elems.hasMoreElements()))
      found = e.equals(elems.nextElement());
    return found;
 }
}):- implements(?X,Enumerable<?El>),
     implements(?El,Equality<?El>).
```

File: examples/SearchableArray.rub

```
/** Implement, on the array class, the Searchable interface whenever
    the elements in the array support an equality test (implement the
    Equality interface) */

implements_I(Array<?El>,Searchable<?El>,{
  public boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equals(elementAt(i++));
    return found;
  }
}) :- implements(?El,Equality<?El>).
```

A.2.2 Benchmark 3

File: examples/SetTest.jrub

```
#verbatim {package aRuBa.tyRuBa.examples;}

#include "Equality.rub"
/** You can control whether TyRuBa knows that "String" implements
    Ordered by commenting out the line below */
#include "Ordered.rub"
#include "SetWithInterface.rub"

#generate SetInterface<String>
#generate List<String>
```

```
#generate Tree<String>
#generate Set<String>

#verbatim {

public class SetTest {

  public static void main(String[] args) {
    Set<String> set = new Set<String>();
    System.out.println(set.contains("World"));
    set.insert("World");
    System.out.println(set.contains("World"));
    System.out.println(set.contains("Hello"));
    set.insert("Hello");
    System.out.println(set.contains("Hello"));
    set.insert("Hello");
  }
}

}
```

File: examples/Ordered.rub

```
interface_I(Ordered<?this>,{
  int compareTo(?this);
}).

implements_I(String,Ordered<String>,{}).
```

File: examples/SetWithInterface.rub

```
/***********************************************************\
 A Set the internal representation of which depends on the
 type of element stored in it.
\***********************************************************/

/*----------------------------------------------------------
  The functionality of all Sets
\*----------------------------------------------------------*/
interface_I(SetInterface<?El>,{
   void insert(?El e);
   boolean contains(?El e);
}).

/*----------------------------------------------------------
  Sets with ?El which merely implement Equality
\*----------------------------------------------------------*/
```

```
/** Representation Class */
class_I(Set<?El>,{
   private List<?El> representation = null;
}) :- implements(?El,Equality<?El>).

/** Representation Data Structure */
class_I(List<?El>,{
  ?El first;
  List<?El> rest;

  List<?El>(?El f,List<?El> r) {first=f;rest=r;}
}).

/** Implementation of Set functionality */
implements_I(Set<?El>,SetInterface<?El>,{

   public void insert(?El e) {
     if (!contains(e)) {
       representation = new List<?El>(e,representation);
     }
   }

   public boolean contains(?El e) {
     return listContains(representation,e);
   }

   private static boolean listContains(List<?El> l,?El e) {
     return (l!=null) && (
       l.first.equals(e) || listContains(l.rest,e)
     );
   }

}) :- implements(?El,Equality<?El>).

/*----------------------------------------------------------
  Sets with Ordered ?El
\*----------------------------------------------------------*/

class_I(Set<?El>,{
   private Tree<?El> representation = null;
}) :- implements(?El,Ordered<?El>).

class_I(Tree<?El>,{
  ?El elem;
  Tree<?El> left;
  Tree<?El> right;
  Tree<?El>(?El e,Tree<?El> l,Tree<?El> r) {
     elem=e; left=l; right=r;
  }
}).

/** Implementation of Set functionality */
implements_I(Set<?El>,SetInterface<?El>,{

   public void insert(?El e) {
     representation = insertTree(representation,e);
   }

   private static Tree<?El> insertTree(Tree<?El> t,?El e) {
     if (t==null)
        return new Tree<?El>(e,null,null);
     else {
        int comparison = t.elem.compareTo(e);
        if (comparison<0)
          t.left = insertTree(t.left,e);
        else if (comparison>0)
          t.right = insertTree(t.right,e);
        return t;
     }
   }

   public boolean contains(?El e) {
     return treeContains(representation,e);
   }

   private static boolean treeContains(Tree<?El> t,?El e) {
     if (t==null)
       return false;
     else {
        int comparison = t.elem.compareTo(e);
        if (comparison<0) /*Less*/
          return treeContains(t.left,e);
        else if (comparison>0) /*Greater*/
          return treeContains(t.right,e);
        else /*Equal*/
          return true;
     }
   }

}) :- implements(?El,Ordered<?El>).
```

A.2.3 Benchmark 4

File: examples/TupleTest.jrub

```
#verbatim {package aRuBa.tyRuBa.examples;}

#include "tuple.rub"

#generate Tuple<[int,String,char]>

#generate Project<0,int>
#generate Project<1,String>
#generate Project<2,char>

/** When using the recursive version the following are needed */
#generate Tuple<[String,char]>
#generate Tuple<[char]>
#generate Tuple<[]>
#generate Project<0,String>
#generate Project<1,char>
#generate Project<0,char>

#verbatim {

public class TupleTest {

  static Tuple<[int,String,char]> test =
    new Tuple<[int,String,char]>(1,
      new Tuple<[String,char]>("Hallo",
        new Tuple<[char]>('f',
          new Tuple<[]>())));

  public static void main(String[] args) {
    System.out.println(test.project<0>());
    System.out.println(test.project<1>());
    System.out.println(test.project<2>());
  }
}

}
```

File: examples/tuple.rub

```
/*************************************************************\
 Generic tuples:
   The arity of a tuple may be any integer from {0,1,2,3,...}
   The arity of a tuple is statically known.
   Every element of a tuple has a certain type.
   Not all of the elements must have the same type.
\***************************************************************/

/*-------------------------------------------------------------
  Recursively defined tuples
  -------------------------------------------------------------*/

/** 0 arity Tuple */
class_I(Tuple<[]>,{
}).

/** Inductive definition of Tuple of arity > 0 */
class_I(Tuple<[?T|?R]>,{
  private ?T first;
  private Tuple<?R> rest;

  Tuple<[?T|?R]>(?T f,Tuple<?R> r) {
    first = f;
    rest  = r;
  }
}) :- class(Tuple<?R>).

interface_I(Project<?index,?El>,{
  ?El project<?index>();
}).

implements_I(Tuple<[?El|?R]>,Project<0,?El>,{
  public ?El project<0>() { return first; };
}).

implements_I(Tuple<[?F|?R]>,Project<?i,?El>,{
  public ?El project<?i>() { return rest.project<?iMinus1>();}
}) :- implements(Tuple<?R>,Project<?iMinus1,?El>),
      +(?iMinus1,1,?i).
```

File: examples/FlatTupleTest.jrub

```
#verbatim {package aRuBa.tyRuBa.examples;}

#include "flatTuple.rub"

#generate Tuple<[int,String,char]>

#generate Project<0,int>
#generate Project<1,String>
#generate Project<2,char>
```

```
#verbatim {

public class FlatTupleTest {

  static Tuple<[int,String,char]> test =
    new Tuple<[int,String,char]>(1,"Hallo",'f');

  public static void main(String[] args) {
    System.out.println(test.project<0>());
    System.out.println(test.project<1>());
    System.out.println(test.project<2>());
  }
}

}
```

File: examples/flatTuple.rub

```
/*************************************************************\
 Generic tuples:
   The arity of a tuple may be any integer from {0,1,2,3,...}
   The arity of a tuple is statically known.
   Every element of a tuple has a certain type.
   Not all of the elements must have the same type. Types
\*************************************************************/

/*-------------------------------------------------------------
  Flat Tuples
  -------------------------------------------------------------*/

class_I(Tuple<?TypeList>,{
  Tuple<?TypeList>(?constructorFormals) {
    ?constructorBody
  }
}) :- tupleElementNames(?TypeList,?formalsNamesList),
      JavaFormals(?TypeList,?formalsNamesList,?constructorFormals),
      tupleConstructorBody(?formalsNamesList,?constructorBody).


tupleElementNames(?T,?N) :- tupleElementNames(0,?T,?N).
tupleElementNames(?idx,[],[]).
tupleElementNames(?idx,[?T|?TR],[element<?T,?idx>|?R]) :-
  +(?idx,1,?idxPlus1),
  tupleElementNames(?idxPlus1,?TR,?R).

tupleConstructorBody([],{}).
tupleConstructorBody([?i | ?r],{this.?i=?i; ?cr}) :-
   tupleConstructorBody(?r,?cr).

interface_I(Project<?index,?El>,{
  ?El project<?index>();
}).

implements_I(Tuple<?TypeList>,Project<?idx,?El>,{
  private ?El element<?El,?idx>;

  public ?El project<?idx>() { return element<?El,?idx>; };
}) :- tupleElementNames(?TypeList,?elements),
      element(element<?El,?idx>,?elements).
```

Appendix B

Calculator Expressions Code

B.1 Initialization Files

File: calculator/initfile.rub

```
#include "prolog.rub"
#include "java.rub"
#include "codegenerationM.rub"
#include "deduced.rub"
```

File: prolog.rub

See appendix A.

File: java.rub

See appendix A.

File: calculator/codegenerationM.rub

```
/***************************************************************************
Code generator that chops a class body up into pieces declaring

1) Constructors:

   constructor_M(?class,?argTypes,?declaration).

2) Methods:

   method_M(?class,?returnType,?name,?argTypes,?declaration).

3) Instance variables:

   var_M(?class,?type,?name,?declaration).

Also relationships between classes and interfaces declared in implements
```

```
and extends clauses are "reified".

1) When a ?name apears in the extends clause of ?type

extends_M(?type,?name).

2) When a ?name apears in the implements clause of ?type

implements_M(?type,?name).

** Naming conventions *************************************************

CG_... : Predicates defined in this file meant to only be used internally
         (auxiliaries)


..._M  : Predicates which affect code generation directly. These
         are declared by the user to "make code appear" in the generated
         classes and interfaces.

Other:   "Globally visible names"

************************************************************************
************************************************************************
************************************************************************/

CG_feature(?class,var<?type,?name>,{?type ?name ?init;}) :-
  var_M(?class,?type,?name,?init).
CG_feature(?class,method<?name,?argTypes>,?impl) :-
  method_M(?class,?retType,?name,?argTypes,?impl).
CG_feature(?class,constructor<?argTypes>,?impl) :-
  constructor_M(?class,?argTypes,?impl).

CG_extendsclause(?x,?extendsclause) :-
  FINDALL(NODUP(?extended,extends_M(?x,?extended)),
          ?extended,?extendslist),
  JavaClause(extends,?extendslist,?extendsclause).
CG_implementsclause(?x,?implementsclause) :-
  FINDALL(NODUP(?itf,implements_M(?x,?itf)),
          ?itf,?implementslist),
  JavaClause(implements,?implementslist,?implementsclause).

CG_abstract(?class,{abstract}) :- abstract_M(?class).
CG_abstract(?class,{}) :- NOT(abstract_M(?class)).

generate(?class,{
?abstract class ?class
```

```
  ?extendsClause
  ?implClause
  { ?features }
}) :-
  class_M(?class),
  CG_abstract(?class,?abstract),
  CG_extendsclause(?class,?extendsClause),
  CG_implementsclause(?class,?implClause),
  FINDALL(NODUP([?class|?feature],CG_feature(?class,?feature,?implem)),
          ?implem,
          ?features).

generate(?itf,{
interface ?itf
  ?extendsClause
  { ?features }
}) :-
  interface_M(?itf),
  CG_extendsclause(?itf,?extendsClause),
  FINDALL(NODUP([?itf|?feature],CG_feature(?itf,?feature,?implem)),
          ?implem,
          ?features).
```

File: calculator/deduced.rub

```
/*******************************************************************\
 * Derived information :
 *  This is information computed from the "virtual base program"
 *  DO NOT ASSERT any of these! Only CONSULT them.
\*******************************************************************/

class(?class) :- class_M(?class).
interface(?itf) :- interface_M(?itf).

/** subtype(?subtype,?supertype) :-
      ?subtype is a subtype of ?supertype according to Java type rules
*/
subtype(?t,?t) :- class(?t);interface(?t).
subtype(?sub,?super) :- BOUND(?sub),
   subtype1(?sub,?mid),subtype(?mid,?super).
subtype(?sub,?super) :- NOT(BOUND(?sub)),
   subtype1(?mid,?super),subtype(?sub,?mid).

/** Transitive closure of extends_M */
extends(?t1,?t2) :- extends_M(?t1,?t2).
extends(?t1,?t3) :- BOUND(?t1),extends_M(?t1,?t2),extends(?t2,?t3).
extends(?t1,?t3) :- BOUND(?t3),extends_M(?t2,?t3),extends(?t1,?t2).
```

```
/** Class mentions an interface in its implements clause */
implements1(?class,?itf) :- implements_M(?class,?itf).
implements1(?class,?itf) :- implements_M(?class,?itf2),extends(?itf2,?itf).

/** subtype1(?subtype,?supertype)  :-
       ?supertype occurs in an extends or implements clause
       of ?subtype
*/
subtype1(?subtype,?supertype) :-
  implements_M(?subtype,?supertype);extends_M(?subtype,?supertype).


/** Is a "feature" in a type? */

/* Directly ? */
feature1(?type,method<?returnType,?name,?argTypes>) :-
   method_M(?type,?returnType,?name,?argTypes,?decl).
feature1(?type,var<?varType,?name>) :-
   var_M(?type,?varType,?name,?decl).
feature1(?type,constructor<?argTypes>) :-
   constructor_M(?type,?argTypes,?decl).

/* Which features are inheritable ? */
inheritable(method<?R,?n,?A>).
inheritable(var<?T,?n>).

/* Direct or inherited feature ? */
feature(?type,?f) :- inheritable(?f),
  NODUP([?type|?f],extends(?type,?super),feature1(?super,?f)).
feature(?type,?f) :- feature1(?type,?f).
```

B.2 Framework Code

File: calculator/calculator.rub

```
/** Expressions of type ?T implement Exp<?T> */
interface_M(Exp<?T>) :- value(?T).
method_M(Exp<?T>,?T,eval,[],{
  ?T eval();
}).

/** Values can be turned into self evaluating expressions */
implements_M(?T,Exp<?T>) :- value(?T).
method_M(?T,?T,eval,[],{
  public ?T eval() { return this; }
}) :- value(?T).
```

```
/** Declare the factory and add methods for creating constants */
class_M(Factory).
method_M(Factory,?T,q,[?Init],{
  static ?T q(?Init init) { return new ?T(init); }
}) :- value(?T),
      feature(?T,constructor<[?Init]>).

/** Declare an Op<?name,?ArgTypes> class for every operation on values */
class_M(Op<?name,[?A1 | ?Ar]>) :-
  value(?A1),
  feature(?A1,method<?R,op<?name>,?Ar>).

/** Auxiliary predicate: calculate expression types from value types */
CAL_ExpTypes([],[]).
CAL_ExpTypes([?T | ?R],[Exp<?T> | ?ER]) :- CAL_ExpTypes(?R,?ER).

/** Insert constructors into the Op<...> classes */
constructor_M(Op<?name,[?A1 | ?Ar]>,?Exp,{
  Op<?name,[?A1 | ?Ar]>(?formals) {
    ?initCode
  }
}) :- class_M(Op<?name,[?A1 | ?Ar]>),
      CAL_ExpTypes([?A1 | ?Ar],?Exp),
      JavaFormals(?Exp,?formals),
      JavaFormals(?Exp,?names,?formals),
      CAL_initCode(?names,?initCode).

/** Auxiliary predicate to calculate body of constructor method */
CAL_initCode([],{}).
CAL_initCode([?f|?r],{this.?f=?f; ?ir}) :- CAL_initCode(?r,?ir).

/** Declare instance variables for Op<...> classes */
var_M(Op<?name,[?A1 | ?Ar]>,?type,?var,{}) :-
  class_M(Op<?name,[?A1 | ?Ar]>),
  CAL_ExpTypes([?A1 | ?Ar],?Exp),
  JavaFormals(?Exp,?formals),
  JavaFormalsElement({?type ?var},?formals).

/** Implement the Exp<?T> interface on Op<...> classes */
implements_M(Op<?name,[?A1 | ?Ar]>,Exp<?T>) :-
      class_M(Op<?name,[?A1 | ?Ar]>),
      feature(?A1,method<?T,op<?name>,?Ar>).
method_M(Op<?name,[?A1 | ?Ar]>,?T,eval,?Context,{
  public ?T eval() {
    return ?a1.eval().op<?name>(?evalar);
  }
}) :- class_M(Op<?name,[?A1 | ?Ar]>),
      feature(?A1,method<?T,op<?name>,?Ar>),
      CAL_ExpTypes([?A1 | ?Ar],?Exp),
      JavaFormals(?Exp,?instVars),
      JavaFormals(?Exp,[?a1 | ?ar],?instVars),
      CAL_evalcals(?ar,?evalar).

/** Auxiliary: Add {.eval()} to a list of arguments */
CAL_evalcals([],{}).
CAL_evalcals([?a1],{?a1.eval()}).
CAL_evalcals([?a1 , ?a2 | ?ar],{?a1.eval(),?rest}) :-
   CAL_evalcals([?a2 | ?ar],?rest).

/** Declare Factory methods for creating Op<...> objects */
method_M(Factory,Op<?name,?A>,?name,?CA,{
  static public Op<?name,?A> ?name(?formals) {
     return new Op<?name,?A>(?actuals);
  }
}) :- class_M(Op<?name,?A>),
      feature(Op<?name,?A>,constructor<?CA>),
      JavaFormals(?CA,?formals),
      JavaFormals(?CA,?actualsL,?formals),
      JavaCommaList(?actualsL,?actuals).

/************ Subtyping ****************************************************/

/** A class TypeConv<?R,?T> for every value ?R and ?T such that
    ?R is directly convertable to ?T */
class_M(TypeConv<?R,?T>) :-
   value(?R),value(?T),
   feature(?R,method<?T,as<?T>,[]>).

/** The converted expression is stored in an instance variable */
var_M(TypeConv<?R,?T>,Exp<?R>,from,{}) :-  class_M(TypeConv<?R,?T>).
constructor_M(TypeConv<?R,?T>,[Exp<?R>],{
  public TypeConv<?R,?T>(Exp<?R> init) { from = init; };
}) :- class_M(TypeConv<?R,?T>).

/** TypeConv<?R,?T> is an expression of type ?T */
implements_M(TypeConv<?R,?T>,Exp<?T>) :- class_M(TypeConv<?R,?T>).
method_M(TypeConv<?R,?T>,?T,eval,[],{
  public ?T eval() {
    return from.eval().as<?T>();
  }
}) :- class_M(TypeConv<?R,?T>).

/** Common supertype for every expression convertable to ?T */
```

```
interface_M(Convertable<?T>) :- value(?T).
method_M(Convertable<?T>,Exp<?T>,cast<?T>,[],{
  Exp<?T> cast<?T>();
}) :- interface_M(Convertable<?T>).

/** Trivial type conversion ?T -> ?T */
extends_M(Exp<?T>,Convertable<?T>) :- interface_M(Exp<?T>).
method_M(?E,Exp<?T>,cast<?T>,[],{
  public Exp<?T> cast<?T>() { return this; }
}) :- implements_M(?E,Exp<?T>).

/** Transitive closure of type conversions */
extends_M(Convertable<?R>,Convertable<?T>) :- class_M(TypeConv<?R,?T>).
method_M(?E,Exp<?T>,cast<?T>,[],{
  public Exp<?T> cast<?T>() {
     return new TypeConv<?R,?T>(this.cast<?R>());
  }
}) :- implements1(?E,Convertable<?R>),
      class_M(TypeConv<?R,?T>).

/** Subtype aware factory class */
class_M(SubtypeFactory).
method_M(SubtypeFactory,?R,?m,?A,{
  static ?R ?m(?formals) { return Factory.?m(?actuals); }
}) :- feature(Factory,method<?R,?m,?A0>),
      CAL_convertableTypes(?A0,?A),
      JavaFormals(?A,?formals),
      JavaFormals(?A,?a,?formals),
      CAL_typeConversions(?a,?A0,?actualsL),
      JavaCommaList(?actualsL,?actuals).

CAL_convertableTypes([],[]).
CAL_convertableTypes([?A0|?A0r],[?A|?Ar]) :-
  CAL_convertableType(?A0,?A),
  CAL_convertableTypes(?A0r,?Ar).

CAL_convertableType(Exp<?T>,Convertable<?T>).
CAL_convertableType(?A,?A) :- NOT(equal(Exp<?T>,?A)).

CAL_typeConversions([],[],[]).
CAL_typeConversions([?a|?r],[?A|?R],[?Ca|?Cr]) :-
  CAL_typeConversion(?a,?A,?Ca),
  CAL_typeConversions(?r,?R,?Cr).

CAL_typeConversion(?a,Exp<?T>,{?a.cast<?T>()}).
CAL_typeConversion(?a,?A,?a) :- NOT(equal(Exp<?T>,?A)).
```

```
//-----------------------------------------------------------------------
```

B.3 Instiantiation Code

File: calculator/Calculator.jrub

```
#verbatim{package aRuBa.tyRuBa.calculator;}
#include "calculator.rub"

/******* Values used in the calculator ***********************/

value(Integer).
class_M(Integer).
var_M(Integer,int,value,{}).

value(Float).
class_M(Float).
var_M(Float,float,value,{}).

value(Double).
class_M(Double).
var_M(Double,double,value,{}).

/** Constructors for values */
constructor_M(?class,[?V],{
  public ?class(?V init) { value = init; }
}) :- value(?class), feature(?class,var<?V,value>).

/** Implement a method ?binOp on all value(?class) containing
    a numeric value */
method_M(?class,?class,op<?binOp>,[?class],{
  public ?class op<?binOp>(?class b) {
    return new ?class(this.value ?javaOp b.value);
  }
}) :- value(?class),
      feature(?class,var<?Num,value>),
      numeric(?Num),
      nativebinop(?Num,?binOp,?javaOp).

numeric(int).
numeric(float).
numeric(double).

nativebinop(?Num,add,{+}).
nativebinop(?Num,sub,{-}).
nativebinop(?Num,mul,{*}).
```

```
/** Type Conversions between values */
method_M(Integer,Float,as<Float>,[],{
  public Float as<Float>() { return new Float(value); }
}).
method_M(Float,Double,as<Double>,[],{
  public Double as<Double>() { return new Double(value); }
}).

#generate SubtypeFactory

#generate TypeConv<Integer,Float>
#generate TypeConv<Float,Double>

#generate Exp<Integer>
#generate Exp<Float>
#generate Exp<Double>

#generate Convertable<Integer>
#generate Convertable<Float>
#generate Convertable<Double>

#generate Double
#generate Float
#generate Integer

#generate Factory

#generate Op<add,[Integer,Integer]>
#generate Op<add,[Float,Float]>
#generate Op<add,[Double,Double]>

#generate Op<mul,[Integer,Integer]>
#generate Op<mul,[Float,Float]>
#generate Op<mul,[Double,Double]>

#generate Op<sub,[Float,Float]>
#generate Op<sub,[Integer,Integer]>
#generate Op<sub,[Double,Double]>

#verbatim {
public class Calculator extends SubtypeFactory {

  static public void main(String[] args) {
//    Exp<Integer> e = add(add(q(5),q(3)),
//          add(q(4),q(5)));
//    int i = e.eval().value;
//    System.out.println(i);
```

```
    Exp<Double> e = add(add(q(5.0),q(3)),
       add(q((float)4.0),q(5)));
    double i = e.eval().value;
    System.out.println(i);
//    Exp<Boolean> e2 = not(not(not(q(true))));
//    boolean b = e2.eval().value;
//    System.out.println(b);
  }

}
}
```

Appendix C

Aspect-Oriented Example Code

C.1 Initialization Files

File: cool/initfile.rub

```
#include "prolog.rub"
#include "java.rub"
#include "layered.rub"
#include "coolCodegenerator.rub"
#include "deduced.rub"
```

File: prolog.rub

See appendix A

File: java.rub

See appendix A

File: cool/layered.rub

```
/**********************************************************************
 Layers of code to code transformations on top of eachother.

 General Idea:

 ============================================================
  bottom layer= Base code. No aspects.
  +
  /--------------------------------
  | Aspect declarations
  /--------------------------------
 ============ V generate code ===============================
  code with aspects "weaved" into it

  /--------------------------------
  | More aspect declarations
  /--------------------------------
 ============ V Generate code ================================
  ....
 ============ V generate code ================================
  Final code with all aspects
 ============================================================

*/

/* The final layer is always called FI. This layer is used to
   actually generate real Java output code.
*/

CG_abstract(?class,{abstract}) :- abstract_(FI,?class).
CG_abstract(?class,{}) :- NOT(abstract_(FI,?class)).

CG_feature(?class,var<?name>,?impl) :-
  var_(FI,?class,?type,?name,?impl).
CG_feature(?class,method<?name,?argTypes>,{?head {?body}}) :-
  method_(FI,?class,?retType,?name,?argTypes,?head,?body).
CG_feature(?class,constructor<?argTypes>,{?head {?body}}) :-
  constructor_(FI,?class,?argTypes,?head,?body).

generate(?class,{
?abstract class ?class
  ?extendsClause
  ?implClause
  { ?features }
}) :-
  class_(FI,?class),
  CG_abstract(?class,?abstract),
  CG_extendsclause(?class,?extendsClause),
  CG_implementsclause(?class,?implClause),
  FINDALL(NODUP([?class|?feature],CG_feature(?class,?feature,?implem)),
          ?implem,
          ?features).

generate(?itf,{
interface ?itf
  ?extendsClause
  { ?features }
}) :-
  interface_(FI,?itf),
  CG_extendsclause(?itf,?extendsClause),
  FINDALL(NODUP([?itf|?feature],CG_feature(?itf,?feature,?implem)),
          ?implem,
        ?features).

CG_extendsclause(?x,?extendsclause) :-
  FINDALL(NODUP(?extended,extends_(FI,?x,?extended)),
          ?extended,?extendslist),
  JavaClause(extends,?extendslist,?extendsclause).
CG_implementsclause(?x,?implementsclause) :-
  FINDALL(NODUP(?itf,implements_(FI,?x,?itf)),
          ?itf,?implementslist),
  JavaClause(implements,?implementslist,?implementsclause).

/** If you want to copy a layer to another layer just
assert the fact: copyLayer(?source,?dest).
*/

class_(?dest,?cl) :- copyLayer(?source,?dest), class_(?source,?cl).
interface_(?dest,?cl) :- copyLayer(?source,?dest), interface_(?source,?cl).
implements_(?dest,?cl,?itf) :- copyLayer(?source,?dest),
    implements_(?source,?cl,?itf).
extends_(?dest,?cl,?itf) :- extends_(?source,?dest),
    implements_(?source,?cl,?itf).
var_(?dest,?class,?type,?name,?impl) :- copyLayer(?source,?dest),
   var_(?source,?class,?type,?name,?impl).
method_(?dest,?class,?retType,?name,?argTypes,?head,?body) :-
   copyLayer(?source,?dest),
   method_(?source,?class,?retType,?name,?argTypes,?head,?body).
constructor_(?dest,?class,?argTypes,?head,?body) :-
   copyLayer(?source,?dest),
   constructor_(?source,?class,?argTypes,?head,?body).
```

File: cool/coolCodegenerator.rub

```
/**********************************************************************
 The cool "code generator" is the layer below the JCore layer. It "weaves"
 COOL aspects into the JCore program.
 **********************************************************************/

/** First we copy all code from the layer above: */
copyLayer(JCore,COOL).

/** For the time being the COOL layer is the final layer */
copyLayer(COOL,FI).

/*********************************************************************
 requires(?class,?method,?condition).
   means that the method ?method of the class ?class must wait
   for ?condition to become true before it may start executing.
   If there is more than one such condition these are combined by means
   of an &&
 *****************************************************************/

/** If a method is selfExclusive it is "guarded" by the following
    required expression */
requires(?class,?name,{COOLBUSY<?name> == 0}) :-
 selfExclusive(?class,?name).

/** If a method is mutually exclusive with an ?other method */
requires(?class,?name,{COOLBUSY<?other> == 0}) :-
 mutuallyExclusive(?class,?names),
 element(?name,?names),
 element(?other,?names),
 NOT(equal(?name,?other)).

COOL_allRequired(?class,?name,?exp) :-
  FINDALL(NODUP(?cond,requires(?class,?name,?cond)),
          ?cond,?conditions),
  JavaConjunction(?conditions,?exp). //Note: fails when ?conditions=[]

/** Wrapper code for COOL methods */
method_(COOL,?class,?Return,?name,?Args,?head,{
    synchronized (this) {
      while (!(?condition)) {
        try { wait(); }
        catch (InterruptedException COOLe) {}
      }
      ?atStart
    }
    try {?body}
    finally {
      synchronized(this) {
        ?atEnd
        notifyAll();
      }
    }
}) :- method_(JCore,?class,?Return,?name,?Args,?head,?body),
      COOL_allRequired(?class,?name,?condition),
      COOL_atStartStatements(?class,?name,?atStart),
      COOL_atEndStatements(?class,?name,?atEnd).

COOL_atStartStatements(?class,?name,?statements) :-
   FINDALL(onEntry(?class,?name,?stat),
           ?stat,?statements).

COOL_atEndStatements(?class,?name,?statements) :-
```

```
   FINDALL(onExit(?class,?name,?stat),
           ?stat,?statements).

/** Declare a COOLBUSY counter variable for every
    method which is either self or mutually exclusive */
var_(COOL,?class,int,COOLBUSY<?name>,{
  private int COOLBUSY<?name> = 0;
}) :- NODUP([?class,?name],
        selfExclusive(?cl,?name);
        mutuallyExclusive(?cl,?mutList),element(?name,?mutList)).

/** Every method for which there is a COOLBUSY counter
    must get some atEnd and atStart statements to maintain
    these counters */
onEntry(?class,?name,{
        ++COOLBUSY<?name>;
}) :- feature1(COOL,?class,var<int,COOLBUSY<?name>>).
onExit(?class,?name,{
        --COOLBUSY<?name>;
}) :- feature1(COOL,?class,var<int,COOLBUSY<?name>>).

/**********************************************************
 An alternative "aspect" language which is defined in
 terms of mutuallyExclusive and selfExclusive

 modifies(?class,?method,?thing)
   :- the ?method modifies the state of ?thing

 inspects(?class,?method,?thing)
   :- the ?method consults the state of ?thing
 **********************************************************/

selfExclusive(?class,?method) :- modifies(?class,?method,?thing).
mutuallyExclusive(?class,[?inspecter|?modifiers]) :-
   inspects(?class,?inspecter,?thing),
   FINDALL( NODUP(?method,modifies(?class,?method,?thing)),
            ?method,
            ?modifiers ).
mutuallyExclusive(?class,?modifiers) :-
   NODUP([?class,?thing],modifies(?class,?xxx,?thing)),
   FINDALL( NODUP(?method,modifies(?class,?method,?thing)),
            ?method,
            ?modifiers ).
```

File: cool/deduced.rub

```
/*****************************************************************\
 * Derived information :
 *  This is information computed from the "virtual base program"
 *  DO NOT ASSERT any of these! Only CONSULT them.
\*****************************************************************/

class(?layer,?class) :- class_(?layer,?class).
interface(?layer,?itf) :- interface_(?layer,?itf).

/** subtype(?subtype,?supertype) :-
      ?subtype is a subtype of ?supertype according to Java type rules
*/
subtype(?t,?t) :- class(?t);interface(?t).
subtype(?sub,?super) :- BOUND(?sub),
   subtype1(?sub,?mid),subtype(?mid,?super).
subtype(?sub,?super) :- NOT(BOUND(?sub)),
   subtype1(?mid,?super),subtype(?sub,?mid).

/** Transitive closure of extends_ */
extends(?layer,?t1,?t2) :- extends_(?layer,?t1,?t2).
extends(?layer,?t1,?t3) :- BOUND(?t1),extends_(?layer,?t1,?t2),
                           extends(?layer,?t2,?t3).
extends(?t1,?t3) :- BOUND(?t3),extends_(?layer,?t2,?t3),
                    extends(?layer,?t1,?t2).

/** Class mentions an interface in its implements clause */
implements1(?layer,?class,?itf) :- implements_(?layer,?class,?itf).
implements1(?layer,?class,?itf) :- implements_(?layer,?class,?itf2),
                                   extends(?layer,?itf2,?itf).

/** subtype1(?subtype,?supertype)  :-
       ?supertype occurs in an extends or implements clause
       of ?subtype
*/
subtype1(?layer,?subtype,?supertype) :-
  implements_(?layer,?subtype,?supertype);
  extends_(?layer,?subtype,?supertype).


/** Is a "feature" in a type? */

/* Directly ? */
feature1(?layer,?type,method<?returnType,?name,?argTypes>) :-
   method_(?layer,?type,?returnType,?name,?argTypes,?head,?body).
feature1(?layer,?type,var<?varType,?name>) :-
```

```
   var_(?layer,?type,?varType,?name,?decl).
feature1(?layer,?type,constructor<?argTypes>) :-
   constructor_(?layer,?type,?argTypes,?head,?body).

/* Which features are inheritable ? */
inheritable(method<?R,?n,?A>).
inheritable(var<?T,?n>).

/* Direct or inherited feature ? */
feature(?layer,?type,?f) :- inheritable(?f),
  NODUP([?type|?f],extends(?layer,?type,?super),feature1(?layer,?super,?f)).
feature(?layer,?type,?f) :- feature1(?layer,?type,?f).
```

C.2 Bounded Stack Example

File: cool/BoundedStack.jrub

```
#verbatim{package aRuBa.tyRuBa.cool;}

#include "BoundedStack.rub"

#generate BoundedStack
```

File: cool/BoundedStack.rub

```
/*============================================================
  Basic functionality (no aspect stuff)
  ============================================================*/

/** The class BoundedStack */
class_(JCore,BoundedStack).
extends_(JCore,BoundedStack,Object).

/** Instance Variables */
var_(JCore,BoundedStack,int,MAX,{static final int MAX = 10;}).
var_(JCore,BoundedStack,{Object[]},contents,
  {Object[] contents = new Object[MAX];}).
var_(JCore,BoundedStack,int,pos,{int pos = 0;}).

constructor_(JCore,BoundedStack,[],{public BoundedStack()},{}).

method_(JCore,BoundedStack,boolean,full,[],{
   public boolean full()},{
      return pos==MAX;}).

method_(JCore,BoundedStack,boolean,empty,[],{
   public boolean empty()},{
      return pos==0;}).

method_(JCore,BoundedStack,void,push,[Object],{
   public void push(Object e)},{
     contents[pos++]=e; }).

method_(JCore,BoundedStack,Object,pop,[],{
   public Object pop()},{
     return contents[--pos]; }).

method_(JCore,BoundedStack,Object,peek,[],{
   public Object peek()},{
     return contents[pos]; }).

method_(JCore,BoundedStack,void,print,[],{
   public void print()},{
      System.out.print("[");
      for (int i=0;i<pos;i++) {
         System.out.print(contents[i]+" ");
      }
      System.out.print("]");
   }).

/*============================================================
  Synchronization Aspect Declarations
  ============================================================*/

//selfExclusive(BoundedStack,push).
//selfExclusive(BoundedStack,pop).
//selfExclusive(BoundedStack,print).

//mutuallyExclusive(BoundedStack,[push,pop,peek]).
//mutuallyExclusive(BoundedStack,[push,pop,empty]).
//mutuallyExclusive(BoundedStack,[push,pop,full]).
//mutuallyExclusive(BoundedStack,[push,pop,print]).

requires(BoundedStack,push,{!full()}).
requires(BoundedStack,pop,{!empty()}).

/* Another way of expressing the outcommented lines above
  (The advantage here is that the declarations do not "cross cut"
   methods. Adding a new method is easy and does not require
   reconsidering all of the "mutuallyExclusive" declarations
*/
modifies(BoundedStack,push,this).
modifies(BoundedStack,pop,this).
inspects(BoundedStack,peek,this).
inspects(BoundedStack,empty,this).
inspects(BoundedStack,full,this).
modifies(BoundedStack,print,SystemOut).
inspects(BoundedStack,print,this).
```

File: cool/Test.java

```
package aRuBa.tyRuBa.cool;

class Consumer extends Thread {
  public void run() {
    Object hold;
    for (int i=1;i<100;i++) {
      synchronized(Test.st) {
        hold = Test.st.pop();
        System.out.print("C: ");
        System.out.println(hold);
      }
    }
  }
}

class Producer extends Thread {
  public void run() {
    for (int i=1;i<100;i++) {
      synchronized(Test.st) {
        Test.st.push(""+i);
        System.out.println("P: "+i);
      }
    }
  }
}

public class Test {

  static BoundedStack st = new BoundedStack();

  static Thread consumer = new Consumer();
  static Thread producer = new Producer();

  public static void main(String[] args) {
    producer.start();
    consumer.start();
  }

}
```

Bibliography

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchel. Adding type parametrization to java. In *OOPSLA '97 Conference Proceedings*, volume 32(10) of *ACM SIGPLAN Notices*, pages 49–65, 1997.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., second edition, 1996.

[BCC$^+$95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BDMN73] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studentliteratur, Schweden, 1 edition, 1973.

[BI82] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 133–141. ACM, ACM, January 1982.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Object-Oriented Programming 11th European Conference, ECOOP '98, Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear.

[BPF97] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In Mehmet Aksit and Satoshi Matsuoka, editors, *Object-Oriented Programming 11th European Conference, ECOOP '97, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer-Verlag, 1997.

[Bru97] Kim B. Bruce. Increasing java's expressiveness with thistype and match-bounded polymorphism. Technical report, Williams College, 1997.

[CCH$^+$89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Dav93] A. Davison. A Survey of Logic Programming-based Object Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 42–106. MIT Press, Cambridge, MA, 1993.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 156–168, 1995.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional prograias. In Richard DeMillo, editor, *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, NM, January 1982. ACM Press.

[DN67] O.-J. Dahl and K. Nygaard. Simula Begin. Technical report, Norsk Regnesentral (Norwegian Computing Center), Oslo/N, 1967.

[DV98] Kris De Volder. Type-oriented logic meta programming for java. Technical Report vub-prog-tr-98-03, Programming Technology Lab, Vrije Universiteit Brussel, 1998.

[DVM97] Kris De Volder and Wolfgang De Meuter. Type-oriented programming. In Jan Bosch and Stuart Mitchell, editors, *ECOOP 97 Workshop Reader*, Lecture Notes in Computer Science, pages 123–125. Springer Verlag, 1997.

[DVS95] Kris De Volder and Patrick Steyaert. Construction of the Reflective Tower Based on Open Implementations. Technical Report vub-prog-tr-95-01, Programming Technology Lab, Vrije Universiteit Brussel, 1995.

[Fla94] P. A. Flach. *Simply Logical: Intelligent Reasoning by Example*, chapter 2. John Wiley, 1994.

[FM96] K. Fisher and J.C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1:189–220, 1996. Preliminary version appeared in *Proc. Theoretical Aspects of Computer Software,* Springer LNCS 789, 1994, 844–885.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1997.

[GK76] Adele Goldberg and Alan Kay. Smalltalk-72 instruction manual. Technical report, Learning Research Group, Xerox Palo Alto Research Center, 1976.

[GR81] A. Goldberg and D. Robson. The Smalltalk-80 system. *Byte Magazine*, 6(8):36–48, August 1981.

[Gra97] Ian S. Graham. *HTML Sourcebook: A Complete Guide to HTML 3.2 and HTML Extensions.* Wiley, New York, NY, USA, third edition, February 1997.

[Hal91] Paul Haley. Data-driven backward chaining. In *Proceedings of the Second Annual CLIPS Conference*, Houston TX, September 1991. NASA Johnson Space Center.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, 1969.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop.* Amsterdam, The Netherlands, June 1997.

[Jon91a] Mark P. Jones. *GOFER 2.20, An Introduction to Gofer.* CS, Yale, 1991.

[Jon91b] Mark P. Jones. *GOFER 2.21 Release Notes.* CS, Yale, februari 1991.

[Jon93a] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., June 1993. ACM Press.

[Jon93b] Mark P. Jones. *GOFER 2.28 Release Notes.* Departement of Computer Science, Yale University, februari 1993.

[Jon94a] Mark P. Jones. ML typing, explicit polymorphism and qualified types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 56–75. Springer-Verlag, April 1994.

[Jon94b] Mark P. Jones. A theory of qualified types. *Science of Computer Programming*, 22(3):231–256, June 1994.

[Jon95] M. P. Jones. *Gofer.* CS, Yale, August 1995.

[JW85] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report — ISO Pascal Standard.* Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., third edition, 1985. Revised by Andrew B. Mickel and James F. Miner.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 1988.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented*

Programming, 11th European Conference, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[LC98] V. Litvinov and C. Chambers. Constraint-based polymorphism in cecil. Technical Report TR-98-01-01, University of Washington, Department of Computer Science and Engineering, January 1998.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343, January 1995.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software — The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-007 P9710047, Xerox Palo Alto Research Center, http://www.parc.xerox/aop, 1997.

[Llo88] J. W Lloyd. *Foundation of Logic Programming.* Springer-Verlag, 2 edition, 1988.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Vrije Universiteit Brussel, 1997.

[Mae87] Patti Maes. *Computational Reflection.* Phd thesis, Vrije Universiteit Brussel, Artificial Intelligence Lab., Brussels, Belgium, January 1987.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for java. In *Proceedings of 24th ACM symposioum on Principles of Programming Languages*, pages 132–145, 1997.

[McC92] Francis G. McCabe. *Logic & Objects.* International Series in Computer Science. Prentice-Hall, 1992.

[Mit96a] John C. Mitchell. *Foundations for Programming Languages*, chapter 1, pages 235–304. MIT Press, Cambridge, 1 edition, 1996.

[Mit96b] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, 1 edition, 1996.

[Mit96c] John C. Mitchell. *Foundations for Programming Languages*, chapter 4, pages 235–304. MIT Press, Cambridge, 1 edition, 1996.

[MLTK97] K. Mens, C. Lopez, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming. In Jan Bosch and Stuart Mitchell, editors, *ECOOP 97 Workshop Reader*, Lecture Notes in Computer Science, pages 483–496. Springer Verlag, 1997.

[MMMP90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong Typing of Object-Oriented Languages Revisited. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 140–150, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[MMP89] Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the 4th Annual Conference on Object-Oriented Programming : Systems, Languages and Applications (OOPSLA '89)*, pages 397–406, New Orleans, Louisiana, USA, October 1989. ACM Press.

[MMPN93] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, 1993.

[MN87] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, 1987.

[Mos90] Peter D. Mosses. Denotational semantics. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, N.Y., 1990.

[Mos94] Chris Moss. *Prolog++: The Power of Object-Oriented and Logic Programming.* Addison-Wesley, New York, N.Y., 1994.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[PH$^+$97] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.4. Technical report, Yale University, April 1997.

[PS90] Jens Palsberg and Michael I. Schwartzbach. Type Substitution for Object-Oriented Programming. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 151–160, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[Roy94] Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[Sch94] David A. Schmidt. *The Structure of Typed Programming Languages.* MIT Press, 1994.

[Sha96] D. Shang. Are cows animals? *Object Currents 1.* http://www.sigs.com/objectcurrents, January 1996.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *OOPSLA 1996 Conference Proceedings*, ACM Sigplan Notices. ACM Press, 1996.

[Smi82] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, January 1982. Also available as MIT/LCS/TR-272.

[Smi84] Brian C. Smith. Reflection and semantics in LISP. Report ISL-3, ACM/ Xerox PARC, Intell. Systems Lab., Palo Alto, CA, June 1984.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.

[Ste94a] G. L. Steele Jr. Building interpreters by composing monads. In ACM, editor, *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[Ste94b] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, 1994.

[Tho97] Kresten Krab Thorup. Genericity in java with virtual types. In Mehmet Aksit and Satoshi Matsuoka, editors, *Object-Oriented Programming 11th European Conference, ECOOP '97, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer-Verlag, 1997.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *19th Annual Symposium on Principles of Programming Languages*, January 1992.

[WB89] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988. Also to appear in Lisp and Symbolic Computation.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA'98*, 1998. To appear.