ALMA MATER STUDIORUM - Università di Bologna
DEIS - Dipartimento di Elettronica, Informatica e Sistemistica, sede di Cesena
Dottorato di Ricerca in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

Ciclo XXII

Settore Scientifico Disciplinare: ING-INF/05

Costrutti ed Applicazioni di Programmazione Generica in Linguaggi Object-Oriented

Autore

Ing. Maurizio Cimadamore

Coordinatore

Prof. Ing. Paola Mello

Relatore

Prof. Ing. Antonio Natali

Correlatori

Prof. Ing. Andrea Omicini
Dott. Ing. Mirko Viroli

Esame Finale - anno 2010

To my wife

Constructs and Applications of Generic Programming in Object-Oriented Languages

---

Contents

Chapter 1

Introduction

"In spite of its name, today's software is usually not soft enough: adapting it to new use turns out in most ease, to be a harder endeavour than should be."

- Bertrand Meyer -

1 Overview

As complexity in modern software systems grows, it is essential to find ways of satisfying such software requirements as *extendibility* (the ease with which a software system may be adapted to take into account modifications in its requirements), *reusability* (how well a system might be reused, either as a whole or in parts, for the construction of new systems) and *compatibility* (the ease of interconnecting a system with existing ones) [Kru92, Mey86, SDNB02]. Tackling these issues is not just matter of pure programming language design as it must include concerns such as specification and design techniques. It would be wrong, however, to underestimate the technical aspects, by not taking into account the role played by proper programming language features: in the end, any acceptable solution must be expressible in terms of programs written in some programming language. It has been said [Mey86] that "programming languages fundamentally shape the software designers' way of thinking", meaning that the constructs available in a programming language dramatically characterise not only the conciseness and the elegance of the solution of a given problem, but also the ease with which that solution can be adapted in order to satisfy changing requirements, or to solve new and unforeseen problems.

Object-Oriented programming is often associated with the concept of reusability [GHJV95, Mey89]; Object-Oriented languages provides constructs that allow the developer to focus on the classes of objects the system manipulates rather than on the functions the system performs. This is crucial, as the set of functions performed by a system often varies across different, while, on the other hand, the category of objects on which the system acts is likely to be more stable. Consequently, it is often wiser — in the long term — to decompose a system in terms of the categories of object it manipulates, provided that such categories feature a sufficient degree of *abstraction*.

Abstraction in Object-Oriented languages is typically achieved through *polymorphism*, that is the ability to define program entities that may take more than one form. Object-Oriented languages feature two powerful and orthogonal kinds of polymorphism: *subtype* and *parametric* polymorphism. The former makes it possible to define elements as extensions or restrictions of previously defined ones while the latter (commonly referred to as *genericity*) is a technique for defining elements that abstract from one or more parameters representing types. Of the two techniques, subtyping is probably the most commonly known, to the point that in the context of Object-Oriented languages it is often used as a synonym for polymorphism.

Subtype polymorphism is the ability of one type `A` to appear as and be used like another type `B`. The power of subtype polymorphism lies in the ability to provide more specific parameters to operations that have been defined in terms of more abstract data types - e.g. an operation manipulating an element whose type is `A` can be safely passed a parameter whose concrete type is `B`, as the set of functionalities provided by `B` is a strict *superset* of the set of functionalities provided by `A`.

Parametric polymorphism is the ability to define *generic* data-types and operations that abstract from one or more so-called type parameters — such parameters must be supplied (or *instantiated*) to the generic data-type or function before it can be used. For example, a function can be defined to work with lists of any kind, regardless of the concrete element type of the list — which thus becomes a type parameter of the generic function.

Generic programming [MS88] is a technique for developing maximally

reusable data-structures and algorithms. As first described by David Musser, Alexander Stepanov, Deepak Kapur and collaborators generic programming can be thought of as a discipline of "gradual lifting of concrete algorithm" that starts with a practical, useful algorithm and repeatedly abstract over details. Thus, a key idea of generic programming consists in finding the minimal set of abstract properties of the types manipulated by a given generic algorithm. The term *concept* is often used to denote an abstract, language-independent formalisation of such constraints. In the context of Object-Oriented languages, concepts are naturally expressed [SL05] imposing subtyping constraints on type-variables; this requires a powerful mixture of subtype and parametric polymorphism, called *bounded polymorphism*[1].

Curiously, while generic programming denotes a methodology whose underlying model is mostly language-independent, it is often equated with the language features providing support for such paradigm [DJ05]. As a result, definitions of "generic programming" are more or less crafted to mean what the specific programming language features under consideration support — i.e. in the context of C++, the boundaries between generic programming and *template* programming are often blurred, to the point that C++ has become the reference platform for discussions involving Object-Oriented generic programming.

Java has been recently updated (J2SE 5.0) to include features such as generics and wildcards that provide support for generic programming [BOSW98, THE+04]. The addition of genericity to the Java programming language has been particularly problematic as it posed serious compatibility issues which have been addressed by choosing a conservative implementation that ensures *migration compatibility* [Gaf04] — the exploitation of generic libraries by non-generic clients. In addition to generics, Java supports use-site variance [IV06] through wildcards, which allows for a smoother integration between subtype and parametric polymorphism; when used effectively, gener-

[1] Generic programming can still be successfully exploited in languages that do not provide support for bounded polymorphism (e.g. C++). The lack of ability of expressing constraints on type-variables can be workarounded, as described in [Str]; however this often lead to poor programming practices and makes the code less reusable [RS06].

ics and wildcards dramatically improve both the reusability and the ease of use of a Java library, as demonstrated by the generified Java Collection Framework [Mica].

Java generics appear to be exploited mainly as a basis for extending the Java language with new and powerful features such as closures [Lov07], immutable references [ZPA+07], Haskell type-classes [WLT07], ownership types [PNCB06], API versioning [HLS09], local variable type inference [Plu07], integration with domain specific languages (DSLs) [KR08], mixins [ABC03].

Despite frameworks such as [KETF07, DKTE04, vDD04] have been developed in order to facilitate generification of non-generic libraries and programs, Java generics still don't seem to permeate mainstream Java programming. This can be viewed as the result of many contributing factors. First, backward compatibility constraints when generics were first considered led to a translation technique called *type-erasure* [BOSW98], which resulted in several limitations, most noticeably the lack of reification of generic types. Type-erasure is a lossy translation scheme which turns a Java generic source into a behaviourally equivalent Java program without generics/wildcards (hence the term type-erasure, as generic types are *erased* during compilation). This results in a lack of support for type-dependent operations (such as type conversions, instance tests, etc.) involving generic types [Nin07, AR08, CAF04] which has been the subject of several studies [SA98, AFM97, SC06, MBL97, Vir05, CV08b]. Moreover, the late introduction of wildcards[2] to the Java language contributed to the overall impression that the Java type-system with generics/wildcards is both too complex and subtle for the average programmer [SC08, KP06, WT09, VR05].

In this scenario, where generic programming is likely to become a new challenge for a critical mass of developers, it is crucial to refine the support for generic programming in mainstream Object-Oriented languages — both at the design and at the implementation level — as well as to suggest novel ways to exploit the additional degree of expressiveness made available by genericity.

[2]Wildcards were not considered during the first draft of the specification for adding generics to the Java platform; they have been added relatively late in the process in order to enhance expressiveness and reusability of Java libraries.

This study is meant to provide a contribution towards bringing Java genericity to a more mature stage with respect to mainstream programming practice, by increasing the effectiveness of its implementation, and by revealing its full expressive power in real world scenario.

2 Contributions

With respect to the current research setting, the main contribution of the thesis is twofold. First, we propose a revised implementation for Java generics that greatly increases the expressiveness of the Java platform by adding reification support for generic types. Secondly, we show how Java genericity can be leveraged in a real world case-study in the context of the multi-paradigm language integration.

Reification of generic types Several approaches [SA98, AFM97, SC06, MBL97, Vir05, CV08b] have been proposed in order to overcome the lack of reification of generic types in the Java programming language. Existing approaches tackle the problem of reification of generic types by defining new translation techniques which would allow for a run-time representation of generics and wildcards. Unfortunately most approaches suffer from several problems: heterogeneous translations, such as the one defined in [SC06, AFM97], are known to be problematic when considering reification of generic methods and wildcards [CV08b]. On the other hand, more sophisticated techniques requiring changes in the Java runtime, as in [MBL97], supports reified generics through a true language extension (*where clauses*) so that backward compatibility is compromised.

In this thesis we develop a sophisticated type-passing technique for addressing the problem of reification of generic types in the Java programming language; this approach — first pioneered by the so called EGO translator [Vir05] — is here turned into a full-blown solution which reifies generic types inside the Java Virtual Machine (JVM) itself, thus overcoming both performance penalties and compatibility issues of the original EGO translator.

Java-Prolog integration Integrating Object-Oriented and declarative programming has been the subject of several researches and corresponding technologies. Such proposals come in two flavours, either attempting at joining the two paradigms as in [Esp06, ON94], or simply providing an interface library for accessing Prolog declarative features from a mainstream Object-Oriented languages such as Java as in [tuP02, swi, Min, k-p, JLo02]. Both solutions have however drawbacks: in the case of hybrid languages featuring both Object-Oriented and logic traits, such resulting language is typically too complex, thus making mainstream application development an harder task; in the case of library-based integration approaches there is no true language integration, and some "*boilerplate code*" has to be implemented to fix the paradigm mismatch.

In this thesis we develop a framework called PatJ [CV07, CV08a] which promotes seamless exploitation of Prolog programming in Java. A sophisticated usage of generics/wildcards allows to define a precise mapping between Object-Oriented and declarative features. PatJ defines a hierarchy of classes where the bidirectional semantics of Prolog terms is modelled directly at the level of the Java generic type-system.

3 Structure of the Thesis

We now provide an overview of the thesis structure and summaries for each chapter. The original contributions of this thesis — reification of Java generics and multi-paradigm intergration — are discussed in Chapters 4 and 5, respectively.

Chapter 1 - Introduction We provide an overview of the thesis, set the research context, describe the motivations and the actual contributions.

Chapter 2 - Generic Programming in Object-Oriented languages We provide some necessary background information that will be used throughout the rest of this thesis: in particular, we discuss how abstraction is achieved in mainstream Object-Oriented programming languages; finally, we discuss the key assets of generic programming, by

showing how this programming paradigm is leveraged in mainstream programming languages such as C++, Java and Scala.

Chapter 3 - Design and Implementation of Java Generics In this chapter we provide an overview of the features that enable generic programming in the context of the Java programming language, namely generics and wildcards. In particular, we focus on the main design issues posed by generics and wildcards such as subtyping, method type-inference and capture conversion. We then conclude this chapter by discussing the current implementation scheme, called type-erasure, and we provide a brief survey of the approaches that have been proposed so far in order to overcome its main limitations.

Chapter 4 - Reification of Generic Types in the JVM In this chapter we present a novel approach that reifies generic types inside the JVM. More specifically, we describe the results of a research project funded by Sun Microsystems which led to a prototype of a JVM called GCVM, featuring builtin support for generic types. First, we propose an extension to the current classfile format so that full generic type signatures are preserved via custom bytecode attributes; we then show how to extend the implementation of a JVM so that exact type-information for generic types is first reconstructed and then exploited during execution.

Chapter 5 - A Prolog-oriented extension of Java programming In this chapter we present PatJ, a framework that enables seamless cross-language integration between Java and Prolog. We start this chapter by illustrating the main limitations of existing Java vs. Prolog integration approaches and we then show how such problems are addressed in PatJ by introducing a sophisticated mapping between Object-Oriented and logic programming features that heavily relies on generic types, methods and annotations.

Chapter 6 - Conclusions We conclude summarising the thesis, highlighting the contributions and the limitations, and providing a detailed list of works related to the topics addressed in this thesis.

Chapter 2

Generic Programming in Object-Oriented Languages

In this chapter, we introduce some fundamental concepts and definitions that will be used throughout the thesis. First, we show how *polymorphism*, that is the ability to define program entities that may take more than one form, can be exploited in order to enhance the expressiveness in Object-Oriented programming languages. In this section we focus on two main kinds of polymorphism commonly known as subtyping and genericity; the generic programming paradigm, which allows to define flexible and maximally reusable data-structures, is typically enabled by a powerful variety of polymorphism called *bounded polymorphism* [CW85, CCH+89] — the result of the combined exploitation of subtyping and genericity. Finally, we show the language features and idioms enabling generic programming with respect to three mainstream Object-Oriented languages such as C++, Java and Scala.

1 Polymorphism in Object-Oriented Languages

Polymorphism (from greek *poly* = many + *morph* = form) is a common trait of all expressive and powerful type-systems in which a single piece of code can be reused with multiple types. Several varieties of polymorphism can be found in modern programming languages (see [CW85]) — in the context of Object-Oriented programming the most common forms are:

Subtype polymorphism This kind of polymorphism gives a single object many types. We say that a type `S` is a subtype of `T` (written `S <: T`,

if an object of type `S` can be used in any context in which a value of type `T` is expected. In *nominal*, class-based Object-Oriented languages subtype polymorphism has a stronger meaning, and it is often referred to as subclassing (equivalent definitions are *inheritance* or *inclusive polymorphism*). In such languages classes are used not only as a template for the creation of new objects, but also as a tool that enables code reuse, allowing new classes to be derived from existing ones by adding implementations of new methods or overriding (i.e. replacing) implementations of old methods. Moreover, in such languages, the hierarchy induced by subclassing coincides with the hierarchy induced by subtyping, as each new class defines a new type.

Parametric Polymorphism Parametric polymorphism allows a simple piece of code to be typed "generically", using *type-variables* in place of actual types. Each such type-variable can thus be *instantiated* with several concrete types. In Object-Oriented languages subtyping and parametric polymorphism are typically bundled together into a powerful and expressive construct, namely *bounded polymorphism*, which allows restriction on type-variables by specifying upper and/or lower bounds.

Ad-hoc Polymorphism This kind of polymorphism allows a symbol (typically a function) to expose different behaviours when *viewed* at different types which may — or may not not — exhibit a common structure, hence the term *ad-hoc*. The most common form of ad-hoc polymorphism in Object-Oriented languages, namely *overloading*, allows a single function symbol to be associated with several implementations. The compiler (or the runtime system, if *overload resolution* is dynamic) chooses an appropriate implementation for each application of such function, based on the type of the arguments. An important generalisation, known as *multi-method dispatch*, also takes into account the type of the object upon which the method is dispatched [Cha92]. Another form of ad-hoc polymorphism is *type-coercition*, that is, the ability of converting a value of a given type into a value of a different type. Type-coercitions are either *implicitly* performed by the compiler, or *explicitly* defined within

Syntax:

$\mathtt{T}$	:=	$\mathtt{Nat} \mid \mathtt{C}$	types
$\mathtt{L}$	:=	$\mathtt{class\ C\ \{\ \overline{f}{:}\overline{T}\ \}}$	class definitions
$\mathtt{e}$	:=	$\mathtt{new\ C}(\overline{\mathtt{e}})$	object instantiation
	|	$\mathtt{e.f}$	field access
	|	$\mathtt{n},\ \mathtt{n} \in \mathbb{N}$	numbers

Expression Typing:

$$\frac{\mathtt{n} \in \mathbb{N}}{\mathtt{n{:}Nat}} \quad \text{T-NAT}$$

$$\frac{\mathtt{class\ C\ \{\ \overline{f}{:}\overline{T}\ \}} \qquad \overline{\mathtt{e}}{:}\overline{\mathtt{T}}}{\mathtt{new\ C}(\overline{\mathtt{e}})\mathtt{{:}C}} \quad \text{T-NEW} \qquad \frac{\mathtt{class\ C\ \{\ \overline{f}{:}\overline{T}\ \}} \qquad \mathtt{e{:}C}}{\mathtt{e.f}_i\mathtt{{:}T}_i} \quad \text{T-FLD}$$

Reduction Rules:

$$\frac{\mathtt{class\ C\ \{\ \overline{f}{:}\overline{T}\ \}}}{\mathtt{new\ C}(\overline{\mathtt{e}})\mathtt{.f}_i \rightarrow \mathtt{e}_i} \quad \text{R-FLD}$$

Figure 2.1: Syntax, typing and inference rules of TINY

a program — e.g. by means of a type-conversion operator.

It is worth noting that the unqualified term "polymorphism" is, in itself, rather ambiguous, as it can be used to mean different concrete kinds of polymorphism, depending on the particular language community in which it is used. For object-oriented programmers it almost always means inclusion polymorphism, while for functional programmers it usually means parametric polymorphism.

In the remainder of this section we focus primarily on subtype and parametric polymorphism, as their combined exploitation — bounded polymorphism — is the most common technique by which generic programming is enabled in Object-Oriented languages [SL05].

1.1 A Formal Calculus: TINY

In this section we introduce TINY, a minimal Object-Oriented core language that will help us to develop some of the concepts presented throughout this

chapter in a simple, yet elegant way. Our aim is to capture the essence of subtype and parametric polymorphism in Object-Oriented programming while abstracting away from other less relevant and language-dependent details, which would just make the formalisation more verbose without adding any relevant description. Here we are not interested in a full soundness result — this is typically accomplished by proving that well-typed programs never lead to run-time errors, as in [WF94]; a more formal tractation of the concepts presented throughout the following sections — including full soundness proof — can be found in [Car84, CW85, ACC93, CCH+89, IPW99].

The syntax and the typing rules of TINY are given in Figure 2.1. In the following, the metavariables `C`, `D` range over class names; `S`, `T` range over types; `L` over class declarations; `f` and `g` over field names; and `e` and `d` range over expressions. Symbol $\overline{\texttt{f}}$ is written as shorthand for a possibly empty sequence $\texttt{f}_1, \texttt{f}_2, \ldots \texttt{f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{e}}$, etc.) and pairs of sequences are also abbreviated in the obvious way, writing $\overline{\texttt{e}}{:}\overline{\texttt{T}}$ as shorthand for $\texttt{e}_1 : \texttt{T}_1, \texttt{e}_2 : \texttt{T}_2, \ldots \texttt{e}_n : \texttt{T}_n$ — where the notation `e:T` is used to indicate that an expression `e` has type `T`. The empty sequence is denoted by $\bullet$, and concatenation of sequences by a comma.

The syntax of TINY is reminiscent of some mainstream programming languages such as Java and C#. Class are declared using the `class` keyword. A class definition can optionally include a list of *fields* $\overline{\texttt{f}} : \overline{\texttt{T}}$; TINY supports two kinds of types: class types — each class declaration implicitly defines a new type — and the builtin `Nat` type used to encode natural numbers. The following TINY program contains two class declarations modelling two-dimensional shapes: `Rectangle` has two fields — `width` and `height`, respectively — of type `Nat`; `Circle` has one field, namely `radius`, whose type is, again, `Nat`.

```
class Rectangle {
   width:Nat;
   height:Nat;
}

class Circle {
   radius:Nat;
}
```

TINY supports two kinds of expressions: instance creation and field access. An instance creation expression of the kind `new C(`$\overline{\texttt{e}}$`)` must provide the initial values $\overline{\texttt{e}} : \overline{\texttt{T}}$ for all the declared fields of `C`; moreover, the typing rule T-NEW states that the type of the value $\texttt{e}_\texttt{i}$ must match the declared type of the field $\texttt{f}_\texttt{i}$ in class `C`. The following program illustrates some examples of instance creation expressions.

```
new Rectangle(10, 20)
new Circle(6)
new Rectangle(42) //error - not enough values
new Circle(new Rectangle(10, 20)) // error - type mismatch
```

A field access of the kind $\texttt{e.f}_\texttt{i}$ takes an expression `e` of type `C` (where `C` is a classtype) and retrieves the value of the field $\texttt{f}_\texttt{i}$; consequently, it is required (see the T-FLD rule) that the name of the field $\texttt{f}_\texttt{i}$ must match the name of one of the fields declared by `C`. The following program illustrates some examples of field access expressions:

```
new Rectangle(10, 20).width
new Circle(6).radius
new Rectangle(10, 20).radius //error - no such field
1.height // error - selector must be a class
```

TINY defines just one basic computation rule for field access, namely R-FLD; this rule assumes the object operated upon is first simplified to a value — either a numeric value, or an expression of the kind `new C(`$\overline{\texttt{e}}$`)`[1]. A well-typed field access expression of the kind $\texttt{e.f}_\texttt{i}$, where `e` is an expression of the kind `new C(`$\overline{\texttt{e}}$`)`, simply evaluates to $\texttt{e}_\texttt{i}$, as shown in the following examples:

```
new Rectangle(10, 20).width → 20
new Circle(6).radius → 6
```

1.2 Case Study: a Monomorphic Container Class

We have seen how, given a class declaration of the kind `class C { `$\overline{\texttt{f}}$`:`$\overline{\texttt{T}}$` }`, the set of values $\overline{\texttt{e}}$ in an instance creation expression of the kind `new C(`$\overline{\texttt{e}}$`)` must match exactly the types of the declared fields $\overline{\texttt{f}}$ so that $\overline{\texttt{e}} : \overline{\texttt{T}}$. This can be seen

[1] This is somewhat similar to the beta-reduction rule of lambda-calculus, where it is assumed that the function is first simplified to a lambda abstraction.

(a) Monomorphic

```
class Rectangle { ... }
class Circle { ... }

class PairRect {
   fst:Rectangle
   snd:Rectangle
}

class PairCircle {
   fst:Circle
   snd:Circle
}
```

(b) Subtype Polymorphism

```
class Shape ◁ Object { }
class Rectangle ◁ Shape { ... }
class Circle ◁ Shape { ... }

class Pair {
   fst:Shape
   snd:Shape
}
```

(c) Parametric Polymorphism

```
class Rectangle { ... }
class Circle { ... }

class Pair<U,V> {
   fst:U
   snd:V
}
```

(d) Bounded Polymorphism

```
class Shape ◁ Object { }
class Rectangle ◁ Shape { ... }
class Circle ◁ Shape { ... }

class Pair<U ◁ Shape,V ◁ Shape> {
   fst:U
   snd:V
}
```

Figure 2.2: Different kinds of polymorphism at a glance

as annoyingly rigid: suppose that we want to define a pair-like data-structure for storing `Rectangle` *and* `Circle` objects; unfortunately, there's no way to define such a data-structure, no matter what type we choose for `fst` and `snd`. If we choose such type to be `Rectangle`, any attempt of creating a `Pair` from two objects of type `Circle` would fail, as an expression of type `Circle` cannot be used to initialise a field of type `Rectangle` — this would violate the typing rule T-New in Figure 2.1. A similar argument applies if `fst` and `snd` are given the type `Circle` instead.

The only way to solve this problem is to define many clones of the `Pair` class, `PairRect` for storing `Rectangle` objects, `PairCircle` for storing `Circle` objects, and so on — as shown in Figure 2.1a. These declarations are indeed very similar — the only difference being the declared type of the

fields `fst` and `snd`. This approach is, however, less than satisfactory: first, it leads to a significant duplication of code — as a new `Pair` clone is required each time we define a new shape class; secondly, it only partially addresses the original problem, as it is still not possible to create a `Pair` object storing e.g. a `Rectangle` and a `Circle`:

```
PairRect(Rectangle(10,20), Rectangle(20,10))
PairCircle(Circle(10), Circle(20))
PairRect(Circle(10), Rectangle(20,10)) //type-error
PairCircle(Circle(10), Rectangle(20,10)) //type-error
```

Classes like `PairRect` and `PairCircle` are said to be *monomorphic* — that is, their code is specific to the type of the elements stored in the pair; in the following section we discuss several extensions to the basic typing rules given in Figure 2.1 that would allow us to define a more abstract, *polymorphic* implementation for `Pair`.

1.3 Subtype Polymorphism

Subtype polymorphism (see Figure 2.3) greatly enriches the expressiveness of a programming language, by defining a reflexive, transitive relation '`<:`' on types; we say that `S` is a subtype of `T`, written `S <: T`, meaning that any term of type `S` can safely be used in a context where a term of type `T` is expected (see rule T-Sub) — this is also called *Liskov substitution principle* [Lis87]. Subtyping can be intuitively be understood in terms of specialisation: given two types `S` and `T` where `S <: T`, we say that `S` is a specialises (or refines) `T` — that is the set of features provided by `S` is a strict superset of the features provided by `T`.

There is an important distinction between *nominal* subtyping, in which only types declared in a certain way may be subtypes of each other, and *structural* subtyping, in which the structure of two types determines whether or not one is a subtype of the other. Tiny features nominal subtyping, where each class declaration corresponds to a new type; at the level of user-defined classes, subtyping is expressed through the $\triangleleft$ relation; we say that `C` is a *subclass* of `N` if `C` $\triangleleft$ `N` — where `N` can be either a user-defined class or the special class `Object` (which is assumed to be the *root* of the subclassing hierarchy).

Syntax:

T	:=	Nat | N	types
N	:=	Object | C	class types
L	:=	class C ◁ N { $\overline{\texttt{f}}$:$\overline{\texttt{T}}$ }	class definitions
e	::=	new N($\overline{\texttt{e}}$)	object instantiation
	|	e.f	field access
	|	n, n ∈ $\mathbb{N}$	numbers

Expression Typing:

$$\frac{\texttt{n} \in \mathbb{N}}{\texttt{n:Nat}}\ \text{T-Nat} \qquad \frac{\texttt{S <: T} \quad \texttt{e:S}}{\texttt{e:T}}\ \text{T-Sub}$$

$$\frac{\mathit{fields}(\texttt{N}) = \overline{\texttt{f}} : \overline{\texttt{T}} \quad \overline{\texttt{e}} : \overline{\texttt{T}}}{\texttt{new N(}\overline{\texttt{e}}\texttt{):T}}\ \text{T-New} \qquad \frac{\mathit{fields}(\texttt{N}) = \overline{\texttt{f}} : \overline{\texttt{T}} \quad \texttt{e:N}}{\texttt{e.f}_\texttt{i}\texttt{:T}_\texttt{i}}\ \text{T-Fld}$$

Subtyping Rules:

$$\texttt{T <: T}\ \text{S-Ref} \qquad \frac{\texttt{S <: U} \quad \texttt{U <: T}}{\texttt{S <: T}}\ \text{S-Tra} \qquad \frac{\texttt{class C} \triangleleft \texttt{N \{ } \overline{\texttt{f}}\texttt{:}\overline{\texttt{T}} \texttt{ \}}}{\texttt{C <: N}}\ \text{S-Cls}$$

Field Lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet\ \text{F-Obj} \qquad \frac{\texttt{class C} \triangleleft \texttt{N \{ } \overline{\texttt{f}}\texttt{:}\overline{\texttt{T}} \texttt{ \}} \quad \mathit{fields}(\texttt{N}) = \overline{\texttt{g}} : \overline{\texttt{S}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{f}} : \overline{\texttt{T}}\ ;\ \overline{\texttt{g}} : \overline{\texttt{S}}}\ \text{F-Cls}$$

Reduction Rules:

$$\frac{\mathit{fields}(\texttt{N}) = \overline{\texttt{f}} : \overline{\texttt{T}}}{\texttt{new N(}\overline{\texttt{e}}\texttt{).f}_\texttt{i} \rightarrow \texttt{e}_\texttt{i}}\ \text{R-Fld}$$

Figure 2.3: Syntax, typing and inference rules of Tiny<:

Note that, in general, subtyping and subclassing define two distinct relations on classes. This is not true in Tiny, where subclassing implies subtyping — that is, whenever C ◁ N we also have that C <: N (see rule S-Cls).

In this new enhanced variant of Tiny, an object creation expression of the kind new N($\overline{\texttt{e}}$) is well-typed if the values $\overline{\texttt{e}}$ have types $\overline{\texttt{S}}$ with $\overline{\texttt{S}}$ <: $\overline{\texttt{T}}$, where $\overline{\texttt{T}}$ are the declared types of the fields of N (see rule T-New in Figure 2.3).

Subclassing can be successfully exploited in order to overcome the limitations described in Section 1.2, by observing that the `Rectangle` (resp. `Circle`) class can be viewed as a specialisation of a more abstract class `Shape` — that is, `Rectangle` $\triangleleft$ `Shape` and `Circle` $\triangleleft$ `Shape`. We can now define a general purpose pair-like data-structure that works uniformly on every user-defined shape class: this is accomplished by defining a class `Pair` containing two fields, `fst` and `snd`, whose type is `Shape`, as shown in Figure 2.1b. An instance of `Pair` can be created from any two given values $\texttt{e}_1$:$\texttt{S}_1$, $\texttt{e}_2$:$\texttt{S}_2$, provided that $\texttt{S}_1$ <: `Shape` and $\texttt{S}_2$ <: `Shape` — that is, whenever $\texttt{e}_1$, $\texttt{e}_2$ are instances of a user-defined shape class (thanks to the S-S-Cls rule). Conversely, it is not possible to create a `Pair` object from e.g. two numeric values, as `Nat` is not a subtype of `Shape`:

```
Pair(Rectangle(10, 5), Rectangle(20, 30))
Pair(Rectangle(10, 5), Circle(20))
Pair(1, 2) //type-error
```

The interaction between subtyping and other language features can be very subtle. Consider a field access expression of the kind $\texttt{e.f}_\texttt{i}$, where `e:C`. Thanks to subtyping/subclassing, $\texttt{f}_\texttt{i}$ can now be a field declared in any superclass `N` of `C`; this is accomplished by introducing the lookup operator *fields*, which yields all the accessible fields in a given class `C` — note that the set of fields of the special class `Object` is empty (see rules F-Obj and F-Cls in Figure2.3). However, this extra flexibility comes at a cost: as subtyping allows to selectively forget about type information — this happens each time a more specific type `S` is turned into a more general type `T` via the T-Sub rule — there are situations in which an apparently harmless expression cannot be type-checked, as shown below:

```
Pair(Rectangle(10, 5), Circle(20)).fst.width //type-error
Pair(Rectangle(10, 5), Circle(20)).snd.radius //type-error
```

The type of a field access expression of the kind `new Pair(`$\overline{\texttt{e}}$`).fst` is `Shape`; consequently, any subsequent field access expression would fail to type-check. In fact, the field lookup on the base class `Shape` yields the empty set $\bullet$ — as stated in the F-Cls rule, as *(i)* `Shape` declares no fields, *(ii)* `Shape` $\triangleleft$ `Object`

Syntax:

T	:=	Nat | C<T> | X	types
L	:=	class C<$\overline{X}$> { $\overline{f}$:$\overline{T}$ }	class definitions
e	:=	new C<$\overline{T}$>($\overline{e}$)	object instantiation
	|	e.f	field access
	|	n, n $\in \mathbb{N}$	numbers

Expression Typing:

$$\frac{n \in \mathbb{N}}{\Delta \vdash n : Nat}\ \text{T-Nat} \qquad \frac{\texttt{class C<}\overline{X}\texttt{> \{ }\overline{f}\texttt{:}\overline{T}\texttt{ \}} \quad \Delta \vdash \overline{e} : [\overline{S}/\overline{X}]\overline{T}}{\Delta \vdash \texttt{new C<}\overline{S}\texttt{>(}\overline{e}\texttt{):C<}\overline{S}\texttt{>}}\ \text{T-New}$$

$$\frac{\texttt{class C<}\overline{X}\texttt{> \{ }\overline{f}\texttt{:}\overline{T}\texttt{ \}} \quad \Delta \vdash e : C\texttt{<}\overline{S}\texttt{>}}{\Delta \vdash e.f_i : [\overline{S}/\overline{X}]T_i}\ \text{T-Fld}$$

Reduction Rules:

$$\frac{\texttt{class C<}\overline{X}\texttt{> \{ }\overline{f}\texttt{:}\overline{T}\texttt{ \}}}{\texttt{new C<}\overline{S}\texttt{>(}\overline{e}\texttt{).}f_i \rightarrow e_i}\ \text{R-Fld}$$

Figure 2.4: Syntax, typing and inference rules of TinyV

and *(iii)* $fields(\texttt{Object}) = \bullet$. In other words, there's no way to statically recover the types of the values $\overline{e}$ once T-Sub is first applied — in the above case, this is required in order to check that a value of type `Rectangle` (resp. `Circle`) can be used to initialise a field, namely `fst` (resp `snd`), whose declared type is `Shape`. In mainstream Object-Oriented programming language, this problem is typically addressed by adding some form of explicit type conversion, which would allow to turn a more general type `T` into a more specific type `S`, provided that `S <: T`.

1.4 Parametric Polymorphism

It is sometimes possible that two or more classes have identical structure except for the type annotations being used in their declarations. For instance, the monomorphic classes `PairRect` and `PairCircle` shown in Figure 2.1a are structurally similar, as they both declare two fields, namely `fst` and `snd`,

but with different types: in `PairRect` both fields have type `Rectangle`, while in `PairCircle` they both have type `Circle`.

Parametric polymorhism takes a different approach to the problem of code reuse, by allowing a piece of code to abstract from one or more types; that is, a class declaration might optionally introduce one or more *type-variables* that can be used throughout the class declaration in place of concrete types. Figure 2.4 shows an extension of TINY featuring parametric polymorphism. The new typing rules gives us the ability to view `PairRect` and `PairCircle` as concrete instantiations of a more abstract class declaration in which two type-variables, namely `U` and `V`, are used to model the (abstract) types of the fields `fst` and `snd`, respectively — as shown in Figure 2.1c.

Note that the typing relation is now a ternary relation between a typing environment Δ, used to keep track of the type-variables declared in a given scope, an expression `e` and a type `T` — namely, the expression `e` has type `T` under the typing environment Δ, written $\Delta \vdash$ `e:T`. An instance creation is now an expression of the kind `new C<`$\overline{\texttt{S}}$`>(`$\overline{\texttt{e}}$`)` where the types $\overline{\texttt{S}}$ provide an instantiation for all the abstract types $\overline{\texttt{X}}$ defined by `C`, where `C` is a class declaration of the kind `class C<`$\overline{\texttt{X}}$`> { `$\overline{\texttt{f}}$`:`$\overline{\texttt{T}}$` }`. Note that the typing rule which describes instance creation (T-NEW) is more convoluted, as the field types $\overline{\texttt{T}}$ *might* contain one or more type-variables in $\overline{\texttt{X}}$. Consequently, the types of the values $\overline{\texttt{e}}$ must match the types in $\overline{\texttt{T}}$, where all the occurrences of the type-variables in $\overline{\texttt{X}}$ are replaced with the actual types in $\overline{\texttt{S}}$ — written $[\overline{\texttt{S}}/\overline{\texttt{X}}]\overline{\texttt{T}}$:

```
Pair<Number,Number>(1,2)
Pair<Rectangle,Circle>(Rectangle(10, 5), Circle(20))
Pair<Rectangle,Circle>(1, Circle(20)) //type-error
```

The new `Pair` definition works uniformly with every type `T`, regardless of whether `T` is a classtype or the builtin `Nat` type: consequently, it is not possible to define a pair class that only works on custom-defined shape classes, as *any* pair of types `S`, `T` is a valid instantiation for `Pair`'s type-variables `U` and `V`.

The typing rule which describes field access (T-FLD) is also more convoluted: given a class declaration of the kind `class C<`$\overline{\texttt{X}}$`> { `$\overline{\texttt{f}}$`:`$\overline{\texttt{T}}$` }`, the type of a field access expression of the kind $\texttt{e.f}_\texttt{i}$, where `e` has type `C<S>`, is obtained

by replacing every occurrences of the type-variables $\overline{\texttt{X}}$ in $\texttt{T}_\texttt{i}$ with the actual types in $\overline{\texttt{S}}$ — analogously to the case of instance creation expressions. Below are reported some examples of field access expressions:

```
new Pair<Rectangle,Nat>(Rectangle(10,5),2).fst.width
new Pair<Nat,Circle>(2,Circle(20)).fst.radius //type-error
```

The reader might appreciate that no explicit type conversion is needed here; in fact, given an instance creation expression of the kind `new C<`$\overline{\texttt{S}}$`>(`$\overline{\texttt{e}}$`)`, the rule T-New preserves the types of the values $\overline{\texttt{e}}$, so that the resulting expression has now type `C<S>`. Since no type information is lost here, a subsequent field access expression of the kind $\texttt{e.f}_\texttt{i}$, where `e` is the result of the above instance creation expression, is now well-typed.

1.5 Bounded Polymorphism

Bounded polymorphism is one of the most powerful varieties of polymorphism which combines the expressive power of subtype and parametric polymorphism. A key feature of bounded polymorphism is the ability to associate constraints — commonly referred to as *bounds* — with type-variables. A bound is used to rigorously define the set of types $\overline{\texttt{S}}$ which can be considered as valid replacements for a given type-variable `X`. For example, a type-variable `X`, whose (upper) bound is `U`, can be instantiated with any type `S`, provided that `S <: [S/X]U`; the type substition is necessary as the bound type `U` might refer to the variable `X` itself — this feature is known as *f-bounded* polymorphism [CCH+89].

An extension of Tiny featuring bounded polymorphism is shown in Figure 2.5. The typing environment Δ is used to keep track of the bounds associated with type-variables defined in the current scope; if `X` is a type-variable defined in the current scope — written $\texttt{X} \in \mathit{dom}(\Delta)$ — its bound is denoted by $\Delta(\texttt{X})$.

Again, subclassing can be successfully leveraged, in order to define a hierarchy of shape classes similar to the one discussed in Section 1.3, where `Rectangle` $\triangleleft$ `Shape` and `Circle` $\triangleleft$ `Shape`, respectively. We can thus define a new variant of the `Pair` class that works uniformly over all pair of types $\texttt{S}_1$, $\texttt{S}_2$, where $\texttt{S}_1$ `<: Shape` and $\texttt{S}_2$ `<: Shape`; this is accomplished by modelling

Syntax:

T	:=	Nat | N | X	types
N	:=	Object | C<$\overline{T}$>	class types
L	:=	class C<$\overline{X} \lhd \overline{U}$> $\lhd$ N { $\overline{f}$:$\overline{T}$ }	class definitions
e	:=	new N($\overline{e}$)	object instantiation
	|	e.f	field access
	|	n, n $\in \mathbb{N}$	numbers

Expression Typing:

$$\frac{n \in \mathbb{N}}{\Delta \vdash n : Nat}\ \text{T-Nat} \qquad \frac{\Delta \vdash N\ ok \qquad fields(N) = \overline{f} : \overline{T} \qquad \Delta \vdash \overline{e} : \overline{T}}{\Delta \vdash \texttt{new}\ N(\overline{e}) : N}\ \text{T-New}$$

$$\frac{S <: T \qquad \Delta \vdash e : S}{\Delta \vdash e : T}\ \text{T-Sub} \qquad \frac{\Delta \vdash N\ ok \qquad fields(N) = \overline{f} : \overline{T} \qquad \Delta \vdash e : N}{e.f_i : T_i}\ \text{T-Fld}$$

Subtyping Rules:

$$\Delta \vdash T <: T\ \text{S-Ref} \qquad \frac{\Delta \vdash S <: U \qquad \Delta \vdash U <: T}{\Delta \vdash S <: T}\ \text{S-Tra}$$

$$\frac{\texttt{class C<}\overline{X} \lhd \overline{U}\texttt{>} \lhd N\ \texttt{\{}\ \overline{f}\texttt{:}\overline{T}\ \texttt{\}}}{\Delta \vdash \texttt{C<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]N}\ \text{S-Cls} \qquad \Delta \vdash X <: \Delta(X)\ \text{S-Var}$$

Well-formed types:

$$\Delta \vdash \texttt{Object ok}\ \text{W-Obj} \qquad \frac{X \in dom(\Delta)}{\Delta \vdash X\ ok}\ \text{W-Var}$$

$$\frac{\texttt{class C<}\overline{X} \lhd \overline{U}\texttt{>} \lhd N\ \texttt{\{}\ \overline{f}\texttt{:}\overline{T}\ \texttt{\}} \qquad \Delta \vdash S\ ok \qquad \Delta \vdash \overline{S} <: [\overline{S}/\overline{X}]\overline{U}}{\Delta \vdash \texttt{C<}\overline{S}\texttt{>}\ ok}\ \text{W-Cls}$$

Field Lookup:

$$fields(\texttt{Object}) = \bullet\ \text{F-Obj} \qquad \frac{\texttt{class C<}\overline{X} \lhd \overline{U}\texttt{>} \lhd N\ \texttt{\{}\ \overline{f}\texttt{:}\overline{T}\ \texttt{\}} \qquad fields([\overline{S}/\overline{X}]N) = \overline{g} : \overline{V}}{fields(\texttt{C<}\overline{S}\texttt{>}) = \overline{f}\texttt{:}\ [\overline{S}/\overline{X}]\overline{T}\ ;\ \overline{g} : \overline{V}}\ \text{F-Cls}$$

Reduction rules:

$$\frac{fields(N) = \overline{f} : \overline{T}}{\texttt{new}\ N(\overline{e}).f_i \rightarrow e_i}\ \text{R-Fld}$$

Figure 2.5: Syntax, typing and inference rules of Tiny$\forall$<:

the (abstract) types of the fields `fst` and `snd` using two type-variables `U` and `V`, whose declared bound is `Shape` — as shown in Figure 2.1d

An instance creation expression of the kind `new C<`$\overline{\texttt{S}}$`>(`$\overline{\texttt{e}}$`)`, where $\overline{\texttt{e}} : \overline{\texttt{T}}$, must provide a *valid* instantiation for all the abstract types $\overline{\texttt{X}}$ defined by `C`, where `C` is a class declaration of the kind `class C<`$\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}$`> ◁ N { `$\overline{\texttt{f}}:\overline{\texttt{V}}$` }`. More specifically, the type `C<S>` must be *well-formed* — written $\Delta \vdash$ `C<`$\overline{\texttt{S}}$`> ok` — that is, the types $\overline{\texttt{S}}$ must be compatible with the declared bounds $\overline{\texttt{U}}$ of the type-variables $\overline{\texttt{X}}$ declared in `C` (see rule W-Cls in Figure 2.5). Examples of instance creation expressions in this new augmented system are:

```
Pair<Rectangle,Circle>(Rectangle(10, 5), Circle(20))
Pair<Rectangle(10, 5),Circle>(1, Circle(20)) //type-error
Pair<Nat,Nat>(1,2) //type-error
```

Since both type-variables `U` and `V` of class `Pair` have bound `Shape`, a pair of types `S`, `T` is a valid instantiation for `U`, `V` if `S <: Shape` and `T <: Shape`. Consequently, `Rectangle` and `Circle` are both valid choices, as `Rectangle ◁ Shape` and `Circle ◁ Shape`. On the other hand, `Nat` is not a valid replacement for neither `U` nor `V`, as `Nat` $\not<:$ `Shape`.

Field access rules are similar to the ones discussed in the previous section. The only difference is that now, thanks to subtyping/subclassing, $\texttt{f}_\texttt{i}$ can be a field declared in any superclass `N` of `C`; again, this is accomplished by introducing a lookup operator *fields*, which yields all the accessible fields in a given class `C`. More specifically, given a class declaration of the kind `class C<`$\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}$`> ◁ N { `$\overline{\texttt{f}}:\overline{\texttt{T}}$` }`, a field access expression of the kind $\texttt{e.f}_\texttt{i}$, where `e` has type `C<`$\overline{\texttt{S}}$`>`, yields a type `V`, where `V` is obtained by replacing all occurrences of `X` in the declared type of $\texttt{f}_\texttt{i}$ (see rule T-Fld):

```
Pair<Rectangle,Nat>(Rectangle(10,5),1).fst.width //type-error
Pair<Circle,Circle>(Circle(10),Circle(20)).snd.radius //ok
```

Analogously to the case of parametric polymorphism, no explicit type conversion is needed.

2 Generic Programming

Generic programming [MS88] is an effective, language-independent methodology for developing reusable software libraries that focusses on the process

of *lifting* a concrete algorithm to a more abstract representation so as to maximise reusability without introducing any performance loss. This process leads to the definition of a so-called *generic algorithm*: an abstract, highly parameterised specification of an algorithm which makes only minimal assumptions about the data abstractions the algorithm manipulates — thus leading to maximally reusable and interoperable code.

Generic programming has been pioneered by Musser and Stepanov in the late 1980's who successfully applied it to the construction of sequence and graph algorithms in Scheme, Ada and C. In the early 1990's they shifted focus to C++ and took advantage of templates to construct the Standard Template Library [SL94] (STL). The STL became part of the C++ Standard, which brought generic programming into the mainstream. Since then, generic programming has been successfully applied in the creation of generic libraries for numerous problem domains[BCD+99, JWL03, LSL99, BGL02].

Even though C++ remains the most commonly used language for implementing generic libraries, there is an increasing number of mainstream Object-Oriented languages supporting generic programming features, such as Java, C#, Scala. The goal of this section is twofold: first we try to characterise the minimal set of idioms that enable generic programming in modern Object-Oriented languages; secondly we provide a comparative study of generic programming support in three mainstream Object-Oriented languages such as C++, Java and Scala.

2.1 Concepts

Generic algorithms are specified in terms of abstract *properties* of types. Such properties are typically expressed by formulating an abstract set of requirements on types called *concepts*. Examples of concepts are e.g. an integer data type with an addition operation satisfying the usual axioms; or a list of data objects with a first element, an iterator for traversing the list, and a test for identifying the end of the list.

Types that meet the requirements of a concept are said to *model* the concept. Concepts support the notion of refinement; thus, a concept C_1 might incorporate the requirements of another concept C_2 — in which case C_2 is

said to *refine* C_1. There are three main kinds of constraints defined by a concept [DRS06]:

Syntactic Constraints A syntactic constraint describes the minimal set of features that must be provided by any type modelling a given concept C. There are two kinds of syntactic requirements: *use patterns* and *associated types*. The former is used to denote a set of operations that must be provided by a modelling type — e.g. the availability of a `next()` operation on iterators. The latter is used to express the set of types that must be defined by a modelling type — e.g. the existence of the associated types `Arc` and `Node` in a type modelling a graph. In the context of Object-Oriented languages, syntactic constraints are typically expressed as a set of methods, fields and member types that must be available in a given type `T` modelling a concept C.

Semantic Constraints A semantic constraint is used to describe certain run-time properties that must be uniformly exposed by all the instances of a given modelling type `T`. Such constraints are typically expressed as a set of pre-conditions and post-conditions that instances of the modelling type must preserve. An example of semantic requirement is e.g. that the size of an empty list is always 0.

Performance Constraints A performance constraint is used to express non functional requirements on operations provided by a given modelling type `T` — usually by specifying maximum limits on how long the execution of a given operation will take, or how much of various resources its computation will use. An example of performance constraint is e.g. the requirement that element access on hash maps must execute in constant time.

Concepts play an important role in specifying generic algorithms. Since a concept may be modeled by any concrete type meeting its requirements, algorithms specified in terms of concepts must be able to be used with multiple types — thus, generic algorithms are naturally polymorphic. In the context of Object-Oriented languages featuring bounded polymorphism, concepts

are naturally expressed as a set of bounds that are used to constrain the instantiation of type parameters in a generic function definition (see [SL05]).

The purpose of such restrictions is to guarantee that a generic function can only be instantiated with some concrete types $\overline{\mathtt{T}}$ that provide all the functionalities required by the function implementation. In other words, such constraints can be seen as a set of requirements that must be met, so as not to produce a compile-time error (in the case of a syntactic requirement) or a run-time error (in the case of a semantic requirement) in the function body.

2.2 Concept Support in Mainstream Object-Oriented Languages

The main features of generic programming, i.e., generic algorithms, concepts, refinement, modelling, and constraints, are realised in different ways in different programming languages. In this section we show how generic programming is supported in three mainstream Object-Oriented languages such as C++, Java and Scala. Our case study consists in defining a concept hierarchy modelling two-dimensional moveable shapes. Each shape has a position that is described in terms two-dimensional coordinates x and y; this basic concept is then refined by another concept representing shapes that can be moved in a two-dimensional space — such shapes must additionally provide an operation for updating their position. We then show a generic algorithm for translating a moveable two-dimensional shape of a given offset. We also define a type representing two dimensional circles, that is meant to be a modelling type for the concepts described above.

2.2.1 Generic Programming in C++

Generic programming in C++ is typically enabled through an extensive use of C++'s *templates* feature. Templates allow the programmer to define types and function that abstracts over one or more template variables — hence, templates are a form of parametric polymorphism. The example in Figure 2.6, defines a template function, namely `translate()`, which embodies our generic algorithm; this function defines a template variable called `MoveableShape`, which is used inside the function body to abstract over the concrete type

```
// concept Comparable:
// bool better(const T&, const T&)
template <class MoveableShape>
const MoveableShape&
   translate(MoveableShape& ms, const int xDelta, const int yDelta) {
      ms.moveTo(xDelta + ms.x, yDelta + ms.y);
      return ms;
}

class Circle {
   public:
   int x, y, radius;

   Circle(int x, int y, int radius) {
      this->x = x;
      this->y = y;
      this->radius = radius;
   }
   void moveTo(int x, int y) {
      this->x = x; this->y = y;
   }
};

int main(int, char*[]) {
   Circle c(5, 5, 3);
   c = translate(c, 10, 20);
   printf("%d, %d, %d", c.x, c.y, c.radius);
}
```

Figure 2.6: A taste of generic programming in C++

of a moveable two-dimensional shape. Inside the function body, the shape position is retrieved and then updated — both coordinates are incremented by corresponding offsets that are passed as argument to the template function.

Note that C++ does not provide explicit supports for concepts[2]: in C++ concept constraints are typically expressed in the form of documentation[JWL03, SL05] — it is customary to identify concepts by naming template variables appropriately.

Worse, concepts cannot be translated in terms of C++ templates, as templates do not feature bounded quantification. Hence, it is not possible to associate constraints with template variables — in the case of `MoveableShape`, possible requirements are *(i)* the existence of a pair of coordinates `x` and `y`, that can be *(ii)* updated using the `moveTo()` operation. If the modelling type

[2]Several attempts have been made in order to add concepts to the C++ language [RS06]; however, as of today it remains unclear as to whether concepts will ever be part of the C++ standard.

```
interface Shape2D<X extends Shape2D<X>> {
   int getX();
   int getY();
}

interface MoveableShape2D<X extends MoveableShape2D<X>> extends Shape2D<X> {
   X moveTo(int x, int y);
}

class Circle implements MoveableShape2D<Circle> {
   int centerX, centerY, radius;
   Circle(int x, int y, int radius) {
      centerX = x; centerY = y;
      this.radius = radius;
   }
   public Circle moveTo(int x, int y) {
      centerX = x; centerY = y;
      return this;
   }
   public int getX() { return centerX; }
   public int getY() { return centerY; }
}

class Animator {
   <S extends MoveableShape2D<S>> S translate(S s, int xDelta, int yDelta) {
      s.moveTo(xDelta + s.getX(), yDelta + s.getY());
      return s;
   }
}

class Test {
   public static void main(String[] args) {
      Circle c = new Animator().translate(new Circle(5, 5, 3), 10, 20);
      System.out.println(c.getX());
      System.out.println(c.getY());
      System.out.println(c.radius);
   }
}
```

Figure 2.7: A taste of generic programming in Java

fails to meet such requirements, the template function `translate()` will fail to type-check — unfortunately this can only be discovered when the template function is instantiated. Consequently, C++ does not support key principles of generic programming such as concept definition, refinement and modelling.

2.2.2 Generic Programming in Java

In the Java programming language, generic algorithms are usually realised leveraging generics [JGSB05]; Java generics allow the programmer to define parameterised classes and methods, whose body abstracts over one or more

type-variables (see Section 1.1). Each type-variable can (optionally) be given an upper bound — hence, generics are a form of bounded polymorphism. Despite there is no direct language support for concepts, the reader might appreciate how the combined exploitation of generics and inheritance lead to a concise and elegant specification of the concept constraints. The example in Figure 2.7 defines a hierarchy of generic interfaces, namely `Shape2D` and `MoveableShape2D` modelling two-dimensional shapes and moveable two-dimensional shapes, respectively. Concept refinement is accomplished through ordinary Java inheritance (as `MoveableShape2D` subclasses from `Shape2D`). Consequently, concept modelling is obtained through Java interface implementation — that is, any type modelling `MoveableShape2D` will be required to implement the methods defined in both `Shape2D` and `MoveableShape2D`.

Both interfaces abstract over a type-variable `S` whose upper bound is recursively defined: for instance, the type-variable `S` defined in `Shape2D` must be instantiated with a subtype of `Shape2D<S>`. This recursive definition allows to express the constraint that a modelling type `C` is required to implement `Shape2D<C>` (a similar conclusion holds for `MoveableShape2D`). Note that, as we are using Java interfaces for representing concepts, *all* the syntactic requirements must be expressed in terms of *methods* that a modelling type must implement — as Java interfaces cannot declare non-constant fields.

Thus, our modelling type `Circle` is required to implement both `Shape2D<Circle>` and `MoveableShape2D<Circle>` — thanks to subtyping the latter subsumes the former. Note that the type `Circle` is a valid instantiation for the type-variables defined in `Shape2D` and `MoveableShape2D` — as `Circle <: MoveableShape2D<Circle>` (follows from the class declaration) and `Circle <: Shape2D<Circle>` (follows from subtyping, as `MoveableShape2D<S> <: Shape2D<S>` for any `S`).

Our generic algorithm is implemented in terms of the generic method `translate()`; this method defines a type-variable, namely `S`, whose declared bound is `MoveableShape<S>`. This method type-variable is used to abstract over the concrete type of the moveable two-dimensional shape that is supplied to the generic method `translate()`. Again, a recursive bound definition is used in order to express the constraint that the concrete type instantiating

```
trait Shape2D[X <: Shape2D[X]] {
   var x:Int;
   var y:Int;
}

trait MoveableShape2D[X <: MoveableShape2D[X]] extends Shape2D[X] {
   def moveTo(x:Int, y:Int):X;
}

case class Circle(override var x:Int,
                  override var y:Int,
                  radius:Int) extends MoveableShape2D[Circle] {
   override def moveTo(x:Int, y:Int):Circle = {
      this.x = x;
      this.y = y;
      this;
   }
}

object animator {
   def translate[S <: MoveableShape2D[S]](s:S, xDelta:Int, yDelta:Int) : S = {
      s.moveTo(xDelta + s.x, yDelta + s.y);
   }
}

object test extends Application {
   var c = animator.translate(Circle(5, 5, 3), 10, 20);
   println(c.x);
   println(c.y);
   println(c.radius);
}
```

Figure 2.8: A taste of generic programming in Scala

the method type-variable `S` must implement the `MoveableShape2D` interface.

Hence, concepts can easily be expressed in Java using interfaces; refinement is accomplished through standard interface inheritance, while concept constraints can be expressed as type-variable bounds. Not only the resulting code is more expressive than its equivalent in C++; the Java compiler will also enforce that concept requirements are met by modelling types (e.g. `Circle` must implement methods defined by its superinterfaces) and also that the generic method `translate()` is only applied to a suitable modelling type `T`, where `T <: MoveableShape<T>`).

2.2.3 Generic Programming in Scala

Scala is a powerful Object-Oriented language supporting many features borrowed from functional programming, such as first-class function types,

actor-based concurrency, algebraic types, etc. Generic programming is accomplished in Scala through an effective mixture of trait-based composition and genericity. On the one hand traits [SDNB02] enables all the type-checking features we have discussed in the previous section — a subclass of a trait must provide a definition for all the abstract members in the trait. On the other hand trait-based composition allows for great flexibility, espcially if compared with Java interface inheritance, as a trait can define variables, method bodies, etc.

The example in Figure 2.8 defines two traits, `Shape2D` and `MoveableShape2D`. Concept refinement is expressed in terms of trait inheritance (as `MoveableShape2D` specialises `Shape2D`). Consequently, concept modelling is obtained through trait implementation — that is, any type modelling `MoveableShape2D` will be required to implement the method `moveTo()` declared in `MoveableShape2D`. Note that, since traits can include variable definition, there is no getter method in the `Shape2D` trait; this trait defines two variables, namely `x` and `y` that will be implicitly inherited by all classes implementing the trait.

Concept constraints, as in Java, are expressed as type-variable bounds, as Scala genericity supports bounded polymorphism. This leads to patterns that are indeed identical — except from some minor syntax differences — to those described in the previous section. Hence, concepts can easily be represented using Scala traits; refinement is accomplished through traits inheritance, while concept constraints can be expressed (as in Java) by means of type-variable bounds. As in Java, the static type-checking carried out by the Scala compiler enforces that concept requirements are met by modelling types (e.g. `Circle` must define the `moveTo()` method) and also that the generic method can only be applied to a modelling type (the actual argument passed to `translate()` is a type `T`, where `T <: MoveableShape<T>`). Finally, it has been shown [N'g06, OG08] how Scala provides a more natural mapping for expressing different kinds of syntactic constraints such as access to associated types (not discussed here), thanks to the *type-definition* and *type-aliasing* features. As we speak, Scala is probably the Object-Oriented language featuring the most complete support for generic programming.

Chapter 3

Design and Implementation of Java Generics

The long awaited extension of Java with *generics* has been shipped since J2SE 5.0 after several years of research and development, and currently represents the most substantial Java extension so far. Java generics allow the programmer to define parameterised classes and methods, whose body abstracts over one or more types variables. Each type-variable can (optionally) be given an upper bound — hence, generics are a form of bounded polymorphism. Examples of generic types are `List<String>`, `Map<String, List<Integer>>`. In addition to generics, JDK 5.0 is equipped with a brand new mechanism called *wildcards* — this is the result of applying the construct known as *use-site variance* to the Java programming language [IV06, THE$^+$04]. Wildcard types are types of the kind `List<? extends T>`, `List<? super T>`, `List<?>` — where `T` can be any valid reference type. Hence, wildcards can be considered as a notation to abstract over a number of different instantiations of the same generic class, e.g. any `List<T>` where `T` is subtype of `Number` can be passed to where a `List<? extends Number>` is expected.

On the one hand, the degree of expressiveness provided by the combined exploitation of generics and wildcards finds many suitable applications, e.g., in the Java Collections Framework (JCF) and the the Java Reflection API; in general, wildcards provide a means by which subtyping (inclusive polymorphism) can better integrate with generics (parametric polymorphism). On the other hand, the late introduction of wildcards to the Java language

contributed to the overall impression that the Java type-system with generics/wildcards is both too complex and subtle for the average programmer [SC08, KP06, WT09, VR05] — wildcards essentially feature a multi-variant subtyping structure that (partially) hides a type-system based on existential types, as described in [TEPH05, CDE08, CD09, WT09].

Generics are implemented using a lossy translation scheme named *type-erasure* that literally erases generic types and wildcards during the compilation process; hence, they never enter the runtime domain of the Java Virtual Machine (JVM) — namely, there is no *reification* of them during execution; in fact, generic types and wildcards are mainly introduced as compile-time abstractions to enforce type-safety. As described in detail in [Nin07, AR08, CAF04], this makes generics hardly integrate with important Java frameworks such as Serialization and Reflection; moreover, generics differ from standard Java types as far as type-dependent operations are concerned (cast conversions, type tests through `instanceof` operator, array operations). But most importantly, the lack of reification causes the so-called *heap pollution* problem: certain cast operations are statically accepted (with a warning) and succeed at runtime, but later can cause any field access or method invocation to fail with an unexpected runtime error.

Several solutions have been studied to address this problem — a rather complete list of references is [SA98, AFM97, SC06, MBL97, Vir05, CV08b]. Existing approaches tackle the problem of reification of generic types by defining new translation techniques which would allow for a runtime representation of generics and wildcards. Unfortunately most approaches suffer from several problems: heterogeneous translations such as the one defined in [SC06, AFM97] are known to be problematic when considering reification of generic methods and wildcards [CV08b]. On the other hand, more sophisticated techniques requiring changes in the Java runtime, as in [MBL97], support reified generics through a true language extension (*where clauses*) so that backward compatibility is compromised.

In this chapter, we illustrate how generics and wildcards can be leveraged in Java programs; more specifically we discuss the main features such as generic classes, generic methods and wildcards. We then provide an in depth

analysis of the technical details involved in the design of Java generics, such as method type-inference, capture-conversion and support for raw types. Finally we focus on how generic are effectively deployed in the Java platform; we discuss the type-erasure technique and its main limitations — most noticeably the lack of reification of generic types. We then conclude, by providing a brief survey of the solutions that have been proposed so far in order to add runtime support for generic types and wildcards.

1 Overview of Java Generics

Generics were not considered in the first releases of the Java language, as a sufficient degree of genericity could be achieved by mixing other language features — most noticeably, inclusive polymorphism provided by Java inheritance. In fact, since `Object` is the common supertype for all Java classes, it is possible to define flexible and reusable data-structures that work uniformly on any custom-defined class — it only suffices to use `Object` in place of the concrete element type of the container class. This programming idiom, called the *homogeneous generic idiom* [BOSW98, OW97], was widely used in the pre-generics implementation of the Java Collections Framework (classes `Vector`, `Hashtable`, etc.). The example in Figure 3.1a defines a linked-list class exploiting the generic idiom. As it can be seen, the list can be used to store any kind of Java object; subtyping and inheritance essentially guarantee that e.g. a `String` object can be passed to a method where an object of type `Object` is expected.

The main downside of this approach (as discussed in Section 1.3) is that it causes a loss of type information whenever an element is added to the list; for example, in order to retrieve a string element from the list, an explicit type-conversion is required, as the static type of the `head` field is `Object` — consequently it is not possible to directly access e.g. a member of the type `String` on the element returned by the list. As more complex elements are being added to the list (e.g. list of list of strings), the code used for retrieving and using the list elements becomes increasingly cumbersome and error-prone:

```
List ls = new List(new List("One", null), null);
String one = (String)((List)ls.head).head;
```

```
class List {
  Object head;
  List tail;
  List(Object head,
       List tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

(a) Homogeneous idiom

```
class List<X> {
  X head;
  List<X> tail;
  List(X head,
       List<X> tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

(b) Generified

Figure 3.1: Two implementations of the `List` class

Another problem with this approach is the lack of expressiveness: while this approach can be successfully exploited for coding heterogeneous collection classes, it is not possible to express constraints on the element type of a given list — so that e.g. a compile-time error is issued when an element of the wrong type is added to the list. This lack of expressiveness typically leads to runtime errors (typically `ClassCastException`) when the actual type of the element retrieved from the collection does not match the expected type, as shown below:

```
List ls = new List("One", new List(2, null));
String one = (String)ls.head;
String two = (String)ls.tail.head; //CCE
```

1.1 Generics Classes

Java generics offer a natural solution to the problems posed by the generic idiom, as they allow a class (resp. method) declaration to abstract from one ore more types — this is accomplished by using type-variables in place of concrete Java types. The code in Figure 3.1b shows a possible way to generify our list class; `List` is parameterised on the type-variable `X`; this type-variable is used as a placeholder for the list element type throughout the whole class declaration. In order to create an instance of a generic class, one must provide an *instantiation* for each type-variable occurring in the class declaration. In this case, as `List` declares just one type-variable, only one concrete type must be supplied.

The following code is used to create a list of strings:

```
List<String> ls = new List<String>("One", null);
```

The reader might appreciate that, thanks to Java generics, it is now possible to express the constraint that a given list holds elements of type `T`. This constraint can be successfully exploited during compilation, in order e.g. to check that the elements being added to the list match the expected type `T`.

```
List<String> ls = new List<String>("One", null);
ls.head = new Integer(1); //error
String s = ls.head; //no cast
```

Consequently, no explicit type-conversion is required when an element is to be retrieved from the list, as the type-system now guarantees that a container object of type `List<String>` holds elements of type `String`.

Generic classes, as any other Java class, support inheritance — that is, a generic class can extend (resp. implement) another generic class (resp. interface). This comes handy when it is needed to e.g. define a specialised, non-parameterised version of a given collection class:

```
class NumList extends List<Number>  ...
...
NumList = new NumList(new Integer(1), null);
NumList = new NumList(new Float(1.0f), null);
NumList = new NumList(new String("One"), null); //error
```

Here, `NumList` is a non-generic class subclassing from `List<Number>`. This means that `NumList` inherits all members from `List<Number>` — a field `head` of type `Number` and a field `tail` of type `List<Number>`, respectively. Consequently, it is possible to create a `NumList` from either an `Integer` or a `Float`, as `Integer <: Number` and `Float <: Number`. On the other hand, an object of type `String` cannot be passed to the `NumList` constructor, as `String` $\not<:$ `Number`.

Type-variables can optionally declare one or more upper bounds; a bound can be used to restrict the set of types which can be considered as valid substitutions for a given type-variable — for example, a type-variable `X` whose (upper) bound is `String`, can be instantiated to any type `S`, provided that `S <: String`:

```
class CList<X extends Comparable<X>> extends List<X>  ...
...
CList<String> = new CList("One", null);
CList<Integer> = new CList(new Integer(1), null);
CList<Object> = new CList(new Object(), null); //error
```

In the above code, a specialised implementation of the `List` container is shown, where the type-variable `X` is given an upper bound, namely `Comaparable<X>`. Note that `X` occurs in the declaration of its own bound — this feature is called *f-bounded polymorphism* [CCH+89]. Hence, `CList<String>` and `CList<Integer>` are *well-formed types*, as `String` <: `Comparable<String>` and `Integer` <: `Comparable<Integer>`; on the other hand `CList<Object>` is not well-formed, as `Object` $\not<:$ `Comparable<Object>`.

The ability of expressing recursive bounds on type-variables is a key feature for enabling generic programming in the Java programming language (see section 2.2.2).

1.2 Generic Methods

A generic method is a method abstracting from some types by declaring one or more *method type-variables*; analogously to the case of generic classes, these variables can be used either in the method signature or in the method body — e.g. to declare local variables and to perform type-dependent operations. When a generic method is invoked, the programmer generally has to provide an instantiation of its type parameters, analogously to the case of type-variables instantiation when allocating generic classes. In Java this can be done either *explicitly*, by specifying which concrete types should be replaced for the method type-variables, or *implicitly*, by having the compiler to infer such types from e.g. the type of the actual arguments supplied in a generic method invocation.

In the example in Figure 3.2, the standard list constructors `nil()` and `cons()` are added to the definition of our class `List<X>`. Such constructors are added as static generic methods parameterised in a type-variable `Y`, which is used to represent the element type of the newly created list. In the first call to `nil()`, `Y` is inferred to have type `Integer`, as the method call occur in an assignment context where the type `List<Integer>` is expected. Calling

```
class List<X> {
   ...
   static <Y> List<Y> nil() {
       return new List<Y>(null, null);
   }
   static <Y> List<Y> cons(Y h, List<Y> t) {
       return new List<Y>(h, t);
   }
}
...
List<Integer> li = nil();
List<String> ls = cons("1", List.<String>nil());
```

Figure 3.2: Method type inference in action

`cons()` turns out to be more problematic: the type of `Y` is inferred to have type `Integer`, as we are passing an argument of type `String` where an `Y` is expected. Since type-inference in argument position is not supported [JGSB05, SC08], the nested call to `nil()` would yield the type `List<Object>`, as no assignment context is given here — which would result in a type-error since a `List<Object>` cannot be passed where a `List<String>` is expected. This problem can be solved by explicitly providing the actual type to be replaced for `Y` in the nested call to `nil()`.

1.3 Wildcards

There are some situations in which only partial knowledge about the instantiation of a type-variable is required, hence no instantiation of it is a good choice. Suppose that a method `appendList(List<X> l)` is to be added to class `List<X>`, which takes another list `l` and adds its element to the receiver.

```
class List<X> {
  ...
  List<X> appendList(List<X> l) { ... }
}
```

When this method is invoked on a receiver with type `List<Number>`, only another list of type `List<Number>` can be passed as argument, though it is easy to recognise that also instances of `List<Integer>` and `List<Float>`

could in principle be passed — as both `Integer` and `Float` are subtypes of `Number`. Generalising, any list whose type is `List<T>` can accept lists of type `List<Z>`, where `Z` is a subtype of `T`, but this is not possible using standard generics for they are *invariant* — e.g. `List<Integer>` is *not* a subtype of `List<Number>`:

```
List<Number> ln = ...
List<Integer> li = ...
ln.appendList(ln); //ok
ln.appendList(li); //error
```

This problem is addressed by integrating parametric polymorphism (generics) and inclusive polymorphism (subtyping) [IV06], as developed in the wildcards mechanism introduced in J2SE 5.0 [THE+04]. After a generic class of the kind `List<X>` has been defined, one can use a type of the kind `List<? extends X>`, called a *bounded wildcard (parameterised) type*. The type `List<? extends E>` can be use in place of any type of the kind `List<T>`, where `T <: E`. Hence, the method `appendList()` can be redefined as follows:

```
List<X> appendList(List<? extends X> l) { ... }
```

Thanks to covariant subtyping, `appendList()` in `List<Number>` can now be applied to arguments of the kind `List<Integer>`, `List<Float>`, and so on. The following example, shows another kind of wildcard:

```
class List<X> {
 ...
 void addTo(List<? super X> l) { ... }
}
...
List<Integer> li=...;
List<Number> ln=...;
li.addTo(ln); //ok
ln.addTo(li); //error
```

Here, the `addTo()` method adds all the elements in the receiver list to the list `l` passed as argument. Instead of declaring the type of `l` as being `List<X>`, it is more useful to use a wildcard type of the kind `List<? super T>`: in fact, any instance of a type `List<S>`, where `T <: S`, can be safely passed to the

method — dually to the case '`? extends T`' — e.g. a list of `Number` accepts elements from a list of `Integer`. The last example of wildcard type is the unbounded version `List<?>`, literally meaning *any* `List<T>`, which is used when the actual type of the list element is either unknown or not relevant, as in a method of the kind:

```
class List<X> {
  ...
  static int size(List<?> c) { ... }
}
...
List<Integer> li=...;
List<String> ls=...;
int s1 = size(li); //ok
int s2 = size(ls); //ok
```

All such new types find an extensive use in the Java Collection Framework (see [Mica]) to flexibly define constraints on the parameterisation of collections.

Wildcards cannot be used to create objects in `new` expressions — an instance creation expression of the kind `new List<?>(..)` is disallowed; rather, they can be thought of as sort of interfaces over standard generic types. Wildcards can in fact be understood as a generalisation of standard generic types, where the type parameter is not a concrete type, but rather a set of types, similar to a sort of *interval*; subtyping between wildcards can be intuitively expressed in terms of inclusion of such intervals — a more formal characterisation of subtyping between generic types is given in Section 2.3.

2 Design of Java Generics

In this section we provide an in depth analysis of the technical details involved in the design of Java generics; our goal is to give the reader an idea of the complexity of the underlying type-system by which genericity is enabled in the Java programming language; more specifically we discuss advanced features such as type-inference in method calls, capture conversion and subtyping between generic types. Finally, we show how Java generics allows for a smooth transition from non-generified to generified libraries, thanks to *raw types*.

```
interface I1 { }
interface I2 { }
class A implements I1, I2 { }
class B implements I1, I2 { }
public class C {
   static <Z> Z choose(Z th, Z that) { return th; }
   void main() {
      A z = choose(new A(), new B());
   }
}
```

(a) Method type inference and intersection types

```
class A<X> { }
class B extends A<B> { }
class C extends A<C> { }
class D {
   static <Z> Z choose(Z th, Z that) { return th; }
   void main() {
      choose(new List<B>(), new List<C>());
   }
}
```

(b) Method type inference and infinite types

Figure 3.3: Method type inference: two corner cases

2.1 Method Type Inference

In Java, method type parameters might be left unspecified at the call site; in fact, the Java compiler can statically infer the actual types to be replaced for method type-variables following a variant of the Hindley-Milner algorithm for local type inference [Mil78, PT98]. Java method type-inference is a two step process that can be described as follows:

Inference from actual arguments During this phase, the compiler collects a set of constraints of the kind $\overline{\texttt{T}} <: \texttt{X}$, where the types in $\overline{\texttt{T}}$ are derived from the types of the actual arguments supplied in the generic method call; this is done for each type-variable `X` of the generic method being called. The type of `X` is then assumed to be the *least upper bound*

of the types in $\overline{\texttt{T}}$ — this ensures that the inferred type is the least type that makes each formal argument be greater than the corresponding actual argument type.

Inference from declared bounds/assignment context In the case one or more variables have been left uninferred during the previous step, another round of type-inference is applied. This time the compiler collects a set of constraints of the kind `X <:` $\overline{\texttt{B}}$, where the types in $\overline{\texttt{B}}$ are derived from the types of the declared bounds of the variables $\overline{\texttt{X}}$ of the generic method being called. Additional constraints of the kind `X <:` $\overline{\texttt{T}}$ are added, where the types in `T` are derived from the type of the assignment context in which the method call occurs (if any). Once all such constraints have been collected, each previously uninferred type-variable is inferred to be the *greatest lower bound* of the types in $\overline{\texttt{B}}$ and $\overline{\texttt{T}}$ — this ensures that *(i)* the inferred type is the greatest type that makes the method return type be smaller than the expected type and that *(ii)* the inferred type for `X` is compatible with the declared bounds of `X`.

This apparently simple inference scheme is able to infer correct and sound answers in most practical cases. Unfortunately, such flexibility comes at a price, as the definition of the least upper bound function is perhaps one of the most complex part of the Java Language Specification [JGSB05]; for instance, there are situations in which the compiler can infer types that are not *expressible* in Java — types that a programmer cannot write down for they are not part of the actual Java language. In Figure 3.3a, the method `choose()` is called by passing as arguments an object of type `A` and an object of type `B`, respectively. In this situation the inference process leads to an *intersection type* (see [JGSB05]), namely `Object&I1&I2` — that is, the least upper bound between `A` and `B` is the greatest subtype of all the common supertypes between `A` and `B`, namely `Object`, `I1` and `I2`.

Additional problems might arise when the types supplied to the least upper bound function are generic types; despite, in most cases, wildcards can be fruitfully exploited for improving the quality of the output of the

```
class ListUtils {

   public static List<?> clone(List<?> l) {
      return doClone(l);
   }

   private static <T> List<T> doClone(List<T> l) {
        List<T> newList=new List<T>();
        newList.head=l.head;
        newList.tail=l.tail;
        return newList;
   }
   ...
}
...
List<?> l = new List<String>();
List<?> l2 = ListUtils.clone(l);
```

Figure 3.4: Capture conversion in method calls

type-inference scheme — this can be regarded as one of the most remarkable properties of Java wildcards [THE+04] — the interaction between wildcards and method type inference can lead to very subtle issues. In Figure 3.3b, the method `choose()` is supplied two arguments, of type `List<B>` and `List<C>`, respectively. Under such circumstances, the inference process yields an infinite type, namely `List<? extends A<? extends A<? extends A<...>>>>` — that is, the common supertype between `B` and `C` is some instantiation of the generic class `A`, namely `A<X>`, where `X <: A<X>`[1].

2.2 Capture Conversion

Java wildcards provide a means by which subtyping (inclusive polymorphism) can better integrate with generics (parametric polymorphism). This mechanism, known in literature as *use-site variance* (as opposed to declaration-site variance), has been first introduced by Igarashi and Viroli in [IV06]. This proposal, based on a type-system featuring existential types, turned out to

[1]Since the most widely used Java compilers such as `javac` and `ejc` do not support infinite types yet [Sun], an approximation is used in which the infinite recursion is truncated by an unbounded wildcard as in `List<? extends A<? extends A<?>>>`.

be too constraining in practice — in particular with respect to the problem of the "generification" of the core Java libraries. Hence, Java features a slightly different flavour of use-site variance, where the explicit *open* and *close* operations on existential types have been replaced by the so called *capture conversion* [TEPH05].

Capture conversion essentially amounts at introducing symbolic representatives of the unknown types *hidden* behind wildcards, in the form of *fresh* type-variables generated under the hood by the compiler. Given a class declaration of the kind `class C<`$\overline{\texttt{X}}$ `extends` $\overline{\texttt{B}}$`>`, capture conversion turns a generic type of the kind `C<`$\overline{\texttt{W}}$`>` into a new type `C<`$\overline{\texttt{V}}$`>`, where each type in $\overline{\texttt{V}}$ is obtained by substituting each wildcard type argument `W` in $\overline{\texttt{W}}$ with a fresh type-variable `Z` with certain lower bounds and upper bounds (denoted as Δ^{-}(`Z`) and Δ^{+}(`Z`), respectively). The types in $\overline{\texttt{V}}$ are computed using the interval metaphor as follows:

- if `W` is of the kind `?` (i.e. an unbounded wildcard) then `V` is a fresh type-variable `Z` such that Δ^{+}(`Z`) is `[`$\overline{\texttt{V}}$`/`$\overline{\texttt{X}}$`]B` and Δ^{-}(`Z`) is `<null>`;
- if `W` is of the kind `? extends T` then `V` is a fresh type-variable `Z` such that Δ^{+}(`Z`) is the smallest type between `T` and `[`$\overline{\texttt{V}}$`/`$\overline{\texttt{X}}$`]B` and Δ^{-}(`Z`) is `<null>`;
- if `W` is of the kind `? super T` then `V` is a fresh type-variable `Z` such that Δ^{+}(`Z`) is `[`$\overline{\texttt{V}}$`/`$\overline{\texttt{X}}$`]B` and Δ^{-}(`Z`) is `T`;
- if $\texttt{W}_\texttt{i}$ is not a wildcard then `V` is `W`.

For instance, we have that capture conversion of `Pair<? extends String, Integer>` yields `Pair<Z,Integer>` where `Z` is a fresh type-variable such that Δ^{+}(`Z`) `= String` and Δ^{-}(`Z`) `= <null>` — note that the second argument `Integer` is not affected by capture conversion, for it is not a wildcard. Capture conversion comes into play under the following circumstances:

Membership check When the members of a given type of the kind `C<`$\overline{\texttt{W}}$`>` need to be accessed, a capture conversion is first applied; this conversion yields a new type `C<`$\overline{\texttt{V}}$`>`, where all toplevel wildcards have been replaced by fresh type-variables — standard membership resolution can thus be

applied. For instance the type of the field `head` in an object of type `List<? extends Number>` is discovered by applying capture conversion — this yields the type `List<Z>`, where Δ^+(`Z`) = `Number`; since the declared type of the `head` in `List<X>` is `X`, it follows that the type of `head` viewed as a member of the generic type `List<? extends Number>` is simply `[Z/X]X = Z`.

Direct supertype The direct supertype of a type `C<`$\overline{\texttt{W}}$`>` containing one or more wildcards type arguments, is defined [JGSB05] as the direct supertype of the type `C<`$\overline{\texttt{V}}$`>`, where `C<`$\overline{\texttt{V}}$`>` is obtained by capturing `C<`$\overline{\texttt{W}}$`>`. Consider a class definition of the kind `class D<X> extends C<C<? extends X>>`. The direct supertype of `D<? super String>` is obtained by capturing `D<? super String>` — this yields a type `D<Z>` where Δ^-(`Z`) = `String`. From the above definition, we have that the direct supertype of `D<? super String>` is the direct supertype of `D<Z>`, namely `[Z/X]C<C<? extends X>> = C<C<? extends Z>>`.

Method calls In a method call of the kind `o.m(`$\overline{\texttt{x}}$`)` where $\overline{\texttt{X}}$ are the types of the actual arguments supplied to the method, capture conversion is first applied to the types $\overline{\texttt{X}}$. Consequently, the method call (and method type-inference, if necessary) is handled the usual way — regardless of whether some types in $\overline{\texttt{X}}$ are wildcard types. As shown in the example in Figure 3.4, the type `List<? extends X>` is first captured, yielding the type `List<Z>`, where Δ^+(`Z`) = `X`; then method-type inference proceed as usual, and the method type-variable `X` is instantiated to the type `Z`.

2.3 Subtyping and Decidability

The subtyping algorithm is obtained by a combination of three main ingredients: capture conversion, standard inheritance, and type-argument containment as shown in [TEPH05, CD09]. Let `S` and `T` be two correctly formed class types of the kind `class C<`$\overline{\texttt{U}}$`>` and `class D<`$\overline{\texttt{V}}$`>`, respectively; the algorithm decides whether `S <: T` in the two steps (subtyping rules are discussed in greater details in Figure 3.5).

First, the direct supertype of `S` is accessed until `C`$\neq$`D`. This ensures that `S` is *lifted* to a type of the kind `D<`$\overline{\texttt{U'}}$`>`. Consequently, `S <: T` if the intervals induced

Syntax:

`S, T`	::=	`class C<`$\overline{\texttt{A}}$`>`	class types
	|	`X`	type-variables
	|	`<null>`	bottom type
`A`	::=	`? extends T`	type arguments
	|	`? super T`	
	|	`?`	
	|	`T`	

Subtyping:

$$\texttt{T} \texttt{ <: } \texttt{Object} \quad \text{S-Top} \qquad \texttt{<null> <: T} \quad \text{S-Bot} \qquad \frac{\overline{\texttt{T}} \leq \overline{\texttt{A}}}{\texttt{C<}\overline{\texttt{T}}\texttt{> <: C<}\overline{\texttt{A}}\texttt{>}} \quad \text{S-Cla}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{> extends D<}\overline{\texttt{Y}}\texttt{>} \qquad \overline{\texttt{K}} = \mathit{capture}(\overline{\texttt{X}})}{\texttt{C<}\overline{\texttt{S}}\texttt{> <: [}\overline{\texttt{K}}\texttt{/}\overline{\texttt{X}}\texttt{]D<Y>}} \quad \text{S-Sub}$$

$$\frac{\Delta^{+}(\texttt{X}) = \texttt{T}}{\texttt{X <: T}} \quad \text{S-Upp} \qquad \frac{\Delta^{-}(\texttt{X}) = \texttt{T}}{\texttt{T <: X}} \quad \text{S-Low}$$

Type-containment:

$$\texttt{T} \leq \texttt{T} \quad \text{C-Ref} \qquad \frac{\texttt{S <: T}}{\texttt{S} \leq \texttt{? extends T}} \quad \text{C-Ext} \qquad \frac{\texttt{T <: S}}{\texttt{S} \leq \texttt{? super T}} \quad \text{C-Sup}$$

Figure 3.5: Subtyping rules in Java

by type-arguments in $\overline{\texttt{U'}}$ are smaller than the ones induced by type-arguments in $\overline{\texttt{V}}$, written $\overline{\texttt{U}} \leq \overline{\texttt{V}}$.

As discussed in Section 1.3, wildcard types are hence handled as "intervals" between the lower bound and upper bound, while non-wildcard types are singletons: the three type-containment rules in Figure 3.5 basically amounts at checking interval containment. Wildcard types of the kind `? super` feature contravariant subtyping, as the type-containment relation swaps the two terms in the subtyping test. For instance, the type `List<? super Number>` is a subtype of `List<? super Integer>` since the type-argument `? extends Number` is contained by the type argument `? super Integer`, as `Integer <: Number`.

```
class A<X> { }
class B extends A<A<? super B>> {
   public A<? super B> cast(B b) {
      return b;
   }
}
```

(a) Java code

$$\dfrac{\dfrac{\dfrac{}{\texttt{B} <: \texttt{A<? super B>}}}{\texttt{A<A<? super B>>} <: \texttt{A<? super B>}} \qquad \texttt{B} <: \texttt{X}}{\texttt{B} <: \texttt{A<? super B>}}$$

(b) Subtyping derivation

Figure 3.6: A simple example of non-termination

Decidability of the Java subtyping algorithm — that is, whether subtyping terminates in a finite number of steps for any two given types `S` and `T` — has been the subject of several studies [KP06, KREY06, CD09, MZ06, WT09]. In [KP06] the problem of subtyping decidability is formally characterised with respect to the so called declaration-site variance setting, that is, where variance annotations associated with type-variables are given in a generic class/method declaration, like in C# and Scala — rather than at use-site, as with wildcards. Under this assumption subtyping decidability can be viewed as the result of the interplay between *(i)* contravariance, *(ii) non-finitary inheritance* — `extends`/`implements` clauses possibly leading to non-finite sets of direct supertypes — and *(iii)* multiple instantiation inheritance — implementing several instantiations of the same generic interface (e.g. `Comparable<Integer>` and `Comparable<String>`).

The program in Figure 3.6a shows a basic example of non-termination of the subtyping algorithm; the code essentially triggers a subtyping test of the kind `B <: A<? super B>` — this test should be performed in order to check conformance with respect to the declared return type of the enclosing method `m()`. This recursively leads to the same subtyping test after few algorithmic steps, as shown in Figure 3.6b. This non-termination problem

```
class A<X> { }
class B<X> extends A<A<? super B<B<X>>>> {
   public A<? super B<Object>> m(B<Object> b) {
      return b;
   }
}
```

(a) Java code

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{B<Object>} <: \texttt{A<? super B<Object>>}}{\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<? super B<Object>>}}}{\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<X>} \quad \texttt{B<Object>} <: \texttt{X}}}{\texttt{B<Object>} <: \texttt{A<? super B<B<Object>>>}}}{\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<? super B<B<Object>>>}}}{\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<Y>} \quad \texttt{B<B<Object>>} <: \texttt{Y}}}{\texttt{B<B<Object>>} <: \texttt{A<? super B<B<Object>>>}}}{\texttt{B<B<B<Object>>>} <: \texttt{A<? super B<B<B<Object>>>>}}$$

(b) Subtyping derivation

Figure 3.7: A more convoluted example of not termination

can actually be easily prevented by detecting loops when performing the subtyping test, e.g. by exploiting a *subtyping cache* that keeps track of all the pending subtyping tests. However, in the general case the subtyping cache cannot prevent non-termination; for instance, the code in Figure 3.7a triggers a more convoluted subtyping test of the kind `B<Object> <: A<? super B<Object>>`. Note that caching is, per se, not sufficient to detect this kind of non-termination since this subtyping test recursively leads to the more complicated expression $\texttt{S}_\texttt{i} <: \texttt{T}_\texttt{i}$ where both types `B<Object>` and `A<? super B<Object>>` are nested in `S` and `T`, respectively (`i` is the nesting level), as shown in Figure 3.7b.

In conclusion, decidability of Java subtyping is still an open debate. On the one hand, Java forbids multiple instantiation inheritance which, according to the study in [KP06], would allow for decidable subtyping. On the other hand, as wildcards relies on use-site variance, which is known to be more expressive and powerful with respect to declaration-site variance, there could

exist forms of non-termination that have not been characterised yet — this problem is discussed in [WT09].

2.4 Raw Types

The design of Java generics aimed at achieving the so called *migration compatibility* [Gaf04, BOSW98] — that is the ability of leveraging generic libraries from non-generic (pre JDK 5.0) clients. Such interoperability is ensured by raw-types — a generic type without any type arguments, like e.g. `List`. A raw type `C` is assignment compatible with all the generic instantiations of the kind `C<`$\overline{\texttt{T}}$`>`; hence, raw types greatly simplify the task of interfacing with non-generic code. On the other hand, it is possible to exploit raw types in a potentially unsound way, as shown below:

```
List l = new List<Integer>(new Integer(1), null);
l.tail = new List(new String("two"), null);
List<String> ls = l;
String s = l.head; //CCE
```

The code above creates an object of type `List<Integer>` and then assigns it to a variable of type `List`; since the actual type parameter for the type-variable `X` is statically unknown (`List` is a raw type), it is possible, for instance, to end up with an heterogeneous list containing two elements of type `Integer` and `String`, respectively. Worse, it is possible to assign an object of type `List<Integer>` to a variable — namely `ls` — of a different generic type, namely `List<String>`. This phenomenon, called *heap pollution* [JGSB05], causes the code to unpredictably fail during execution — in this case, the explicit type-conversion added by the compiler (see Section 3.1) fails, as an element of type `Integer` is retrieved when one of type `String` is expected.

In order to prevent unexpected runtime failures, the compiler generates an unchecked warning whenever an object of a raw type `C` is converted into a generic type of the kind `C<`$\overline{\texttt{X}}$`>`. In the above example, a warning is emitted when `l`, whose type is `List`, is assigned to a variable of type `List<String>`; this assignment is said to be *unchecked*, as the correctness of this conversion cannot be guaranteed statically.

3 Implementation of Java Generics

Generics are implemented using a lossy translation scheme named *type-erasure* [JGSB05, BOSW98] that literally erases generic types and wildcards during the compilation process; hence, generic types and wildcards are mainly introduced as compile-time abstractions to enforce type-safety. There are, however, some instructions — such as instance test (`instanceof` operator) or type-conversions — whose semantics depends on the runtime type of an object — we call such operations *type-dependent operations*. Because of type-erasure, generic types and wildcards never enter the runtime domain of the Java Virtual Machine (JVM); as a result, type-dependent operations involving generic types are subject to some unavoidable restrictions — this problem is known in literature as lack of reification of generic types [Nin07, AR08, CAF04]. The aim of this section is to illustrate how type-erasure works, and to discuss the main restrictons imposed by this implementation technique.

3.1 Type-erasure

Java implements generics through an homogeneous translation scheme called type-erasure, which has been first described by Bracha et al. in [BOSW98]. The core idea of type-erasure is to provide an automatic translation from Java code using generics and wildcards into its morally equivalent, non-generic counterpart exploiting the homogeneous generic idiom (see Section 1). The details of this translation process are reported below:

- A generic class of the kind `class C<`$\overline{\texttt{X}}$ `extends` $\overline{\texttt{B}}$`>` (resp. a generic method of the kind `<`$\overline{\texttt{X}}$`>m(`$\overline{\texttt{T}}$`)`) is translated into its *monomorphic* — i.e., non-generic — version `C`, where all the occurrences of the type parameters $\overline{\texttt{X}}$ are replaced with their declared bounds $\overline{\texttt{B}}$ (or `Object`, if no bound is provided).
- When a member `m` of a generic type of the kind `C<`$\overline{\texttt{T}}$`>` is accessed, the compiler automatically adds an explicit type-conversion — provided that the type of `m` has changed under erasure. This cast is required in order to enforce correctness of the generated code, by preventing e.g.

that an element retrieved from a list of type `List<Integer>` is assigned to a variable of type `List<String>` (see Section 2.4).

- When the type of a class member changes under erasure, the compiler emits a special classfile attribute called `Signature` [Micb]. This attribute contains the full generic signature of the erased member and is used by the compiler to reconstruct exact type-information when e.g. a class is accessed from a library.
- It is possible that the erased signature of the overriding method is not *override-equivalent* [JGSB05] with respect to the erased signature of the overridden method; under such circumstances, the compiler generates a special method, called *bridge method* [BOSW98], in order to preserve the semantics of overriding.

For instance, the generic class `List<X>` is translated into a non generic class `List`, where all the occurrences of the type-variable `X` have been replaced by `Object` — this lead to a code which is indeed very similar to the one shown in Figure 3.1a.

The only computational overhead added by type-erasure is due to the insertion of down-casts; for instance the following statements:

```
List<List<String>> lls = new List<List<String>>(...);
String s = lls.tail.head.head;
```

are translated as follows:

```
List lls = new List(...);
String s = (String)((List)((List)lls.tail).head).head;
```

On the other hand, such casts are unavoidable also when using the homogeneous generic idiom, so it can be safely assumed that type-erasure does not significantly alter the application performance. Moreover, since generic code is translated into non-generic code before code-generation, type-erasure ensures that generic code can be understood and executed by a legacy JVM — that is, no runtime extension is required in order to support Java generics.

`A`	:=	`T` | `? extends T` | `? super T` | `?`	Argument types
`T`	:=	`X` | `C` | `C<`$\overline{\texttt{A}}$`>` | `T[]`	Reference types
`R`	:=	`C` | `C<?>` | `R[]`	Reifiable types
`K`	:=	`C` | `C<`$\overline{\texttt{T}}$`>` | `R[]`	Types of objects

Figure 3.8: Syntax of reference types in Java

3.2 Consequences of Type-erasure

In Java there is a subtle distinction between *reifiable* and *non-reifiable* types [JGSB05]. Consider the syntax of Java reference types, reported in Figure 3.8; reference types include generic classes of the kind `C<`$\overline{\texttt{T}}$`>`, non generic classes of the kind `C` (either unparameterised types or raw types), arrays of the kind `T[]` and type-variables of the kind `X`. As it can be seen, not all such types can be used to create objects — only types in `K` can occur in an instance creation expression; in fact, as already mentioned, a generic type of the kind `C<`$\overline{\texttt{T}}$`>` can be used in an instance creation expression, provided that none of its type parameters $\overline{\texttt{T}}$ is a wildcard.

Additional restrictions apply to arrays (the problem of generic arrays is discussed in greater detail in Section 3.2.2): only arrays whose element type is reifiable — either a class (or interface) type with zero type arguments or with the unbounded wildcards everywhere — can be instantiated.

Reifiable types `R` are the only types which can be used as the target type of a type tests using the `instanceof` operator; in other words, such types are the only types the runtime system can "see" when inspecting an object. This is a crucial point: in Java there is a mismatch between those types `K` that are available at compile-time to create objects and types `R` that are available at runtime for inspection.

3.2.1 Unchecked Cast

Despite Java generics virtually eliminate the need of using down-cast, there are situations in which explicit type-conversions are still useful. For instance, the programmer may still need to manage a heterogeneous collection of elements,

```
List<Integer> li = new List<Integer>(new Integer(1), null);
List<String> ls = new List<String>("two", null);
List<List<?>> ll = new List<List<?>>(li,new List<List<?>>(ls,null));
...
List<String> ls2 = (List<String>)ll.tail.head;
List<Integer> li2 = (List<Integer>)ll.tail.head; //?
...
Integer i = li2.head; //CCE
```

(a) Generic code

```
List li = new List(new Integer(1), null);
List ls = new List("two", null);
List ll = new List(li, new List(ls, null));
...
List ls2 = (List)ll.tail.head;
List li2 = (List)ll.tail.head; //ok
...
Integer i = (Integer)li2.head; //CCE
```

(b) Erased code

Figure 3.9: An example of unsound cast conversion

and to retrieve elements from such a collection with their exact type — this can be safely done only by exploiting some form of explicit type-conversion, as shown in Figure 3.9a. First, we create a list of heterogeneous lists (of type `List<List<?>>`. We then add two lists of type `List<Integer>` and `List<String>` respectively. At some later stage, we want to retrieve an element of type `String` from the second list stored in `ll`; this is accomplished by inserting an explicit type-conversion (to `List<String>`), as the original type of the list is *hidden* behind the wildcard type `List<?>`.

Note that a wildcard type of the kind `List<?>` is a common supertype of all possible generic instantiations of `List<X>`, such as `List<Integer>`, `List<String>` and so on; consequently, the semantics of a cast conversion from `List<?>` to `List<Integer>` can only be enforced during execution, when the exact type of the object being converted is known. Unfortunately, type-erasure maps all generic instantiations of a given generic class — as `List<String>`, `List<Integer>` — into the same erased runtime type `List` (see Figure 3.9b). Consequently, the runtime support cannot assert the validity of this cast — which is in fact translated as a simple cast to the

```
List<Integer>[] li_arr = new List<Integer>[]{ ... };
Object[] o_arr = li_arr;
List<String> ls = new List<String>(...);
...
o_arr[0] = ls; //?
List<Integer> li = li_arr[0];
Integer i = li.head; //CCE
```

(a) Generic code

```
List[] li_arr = new List[]{ ... };
Object[] o_arr = li_arr;
List ls = new List(...);
...
o_arr[0] = ls; //ok
List li = li_arr[0];
Integer i = (Integer)li.head; //CCE
```

(b) Erased code

Figure 3.10: An example of unsound usage of generic arrays

erased type `List`.

Allowing potentially unsafe type-conversions leads, again, to heap pollution problems; in this case, we assign an object of type `List<String>` to a variable of a different generic type — namely `List<Integer>`; this is accomplished by using an unsafe type-conversion to the type `List<Integer>`. Such a cast is said to be *unchecked* (and will result in a compile-time warning), as its semantic cannot be enforced, neither statically — as usual, since it is a down-cast — nor dynamically — because of type-erasure.

3.2.2 Generic Arrays

Java arrays feature covariant subtyping — that is, `Integer[] <: Object[]`. While covariant subtyping rules lead to a relatively intuitive and predictable behaviour, they also introduce an hole in the type-system, as semantics of assignments involving arrays must be enforced during execution:

```
Integer[] iarr = new Integer[]{1, 2, 3};
Object[] oarr = iarr;
oarr[2] = "Three"; //ASE
Integer i = iarr[2];
```

Here, we create an array of type `Integer[]` and we assign it to a variable of type `Object[]`. This is allowed, as `Integer <:Object`. We then are free to overwrite an array element with i.e. an element of type `String`, as `String <: Object`. This would be problematic, as we subsequently retrieve an element of type `String` where an `Integer` is expected. The JVM provides a routine that enforces the correctness of array store operations during execution. In the above example, such routine promptly issues a runtime error — namely `ArrayStoreException` — as it detects an attempt to store an object of type `String` into an array of type `Integer[]`.

Unfortunately, the array store check routine cannot be leveraged to prevent bad assignments involving generic arrays. Consider the code in Figure 3.10a; first we assign an array of type `List<Integer>[]` to an array of type `Object[]` — this is correct, since `List<Integer> <: Object`. We then insert an object of the wrong type — namely `List<String>` — into the original array, exploiting the aliased reference `o_arr`; this eventually leads to a runtime error when we retrieve an element from a list in the original array, as an object of type `String` is found, where one of type `Integer` is expected. Note that there is no way to detect the bad array store, as the runtime type of the array `li_arr` is simply `List[]` (see Figure 3.10b). Worse, no unchecked warning can be issued here, as the code above does not rely — neither explicitly nor implicitly — upon any unchecked conversion. Hence, in order to preserve soundness, the creation of generic arrays is forbidden in Java.

4 Alternatives to Type-erasure

Several solutions have been studied to address the lack of reification of generic types [SA98, AFM97, SC06, MBL97, Vir05, CV08b]. Existing approaches address this problem by defining new translation techniques which allow for a runtime representation of generics and wildcards. Such approaches can be classified into two main categories:

Compile-time Compile-time approaches tackle the problem of reification by introducing an alternate, more sophisticated translation scheme that allows exact type-information to be dynamically reconstructed

and used when executing type-dependent operation involving generic types. Following a classification introduced in [OW97], we distinguish between *homogeneous* translations [Vir05, CV08b] — where all generic instantiations of a given type `C<`$\overline{\texttt{T}}$`>` are mapped into a single Java class — and *heterogeneous* translations [SC06] — where each generic instantiation of the kind `C<`$\overline{\texttt{T}}$`>` translates into a different specialised class. Compile-time approaches do not require changes to the runtime environment, as reification is typically achieved by introducing ad-hoc compile-time artifacts.

Runtime Runtime solutions achieve reification by extending the runtime environment; this is done by defining a custom class loader, as in [AFM97], or by redesigning the JVM to directly represent generic types [MBL97]. Note that, though JVM-based approaches typically leads to a better-engineered solution (with greatest performance and coherence) — as developed e.g for the .NET framework [SK01] — they inevitably pose additional problems, as in [MBL97] where reified generics are supported through a true language extension (*where clauses*) so that backward compatibility is compromised.

In this section we focus on two compile-time approaches that have been the subject of several studies, namely NextGen [SC06] and EGO [Vir05, CV08b]. More specifically, we provide an in depth analysis of the latter approach, as it paves the way to the runtime approach that will be discussed in Chapter 4 — one of the main contributions of this work.

4.1 The NextGen Translator

The NextGen compiler addresses the problems introduced by type-erasure by defining an heterogeneous translation scheme where the relationships between generic classes and their instantiations are encoded in a non-generic class hierarchy. For each parametric class, the NextGen translator creates an homogeneous abstract class; this class is indeed very similar to its untranslated counterpart — the only difference being that the class is now marked as `abstract`.

```
abstract class List<X> {
  Object head;
  List<X> tail;
  List(Object head, List<X> tail){
    this.head=head;
    this.tail=tail;
  }
}

interface $List$_String_${}

class $$List$_String_$ extends List<String>
                       implements $List$_String_$ {
  $$List$_String_$(Object head, List<String> tail) {
     super(head,tail);
  }
}
```

Figure 3.11: Translation with NextGen of code in Figure 3.1b

Each time a client class uses a new instantiation of a generic type — say `List<String>` — the translator creates a small wrapper subclass extending the abstract class `List<X>`, and a *marker* interface implemented by this subclass — e.g. `$List$_String_$` (see Figure 3.11). Hence, type-dependent operations such as cast and `instanceof` are expressed in terms of operations involving a more specialised, type-dependent subclass representing a given instantiation `C<`$\overline{\texttt{T}}$`>` of a generic class of the kind `C<`$\overline{\texttt{X}}$`>`.

For example, an instance creation expression is translated as follows:

`new List<String>("1", null)` → `new $$List$_String_$("1",null);`

An instance test involving e.g. the type `List<String>` can be translated as an instance test whose target type is `$List$_String_$`:

`obj instanceof List<String>` → `obj instanceof $List$_String_$`

NextGen heterogeneous translation scheme does not significantly affect performance: it has been shown in [SC06] that the code generated by NextGen is almost as fast as the one obtained through type-erasure, as most type-dependent operations are simply translated in terms of method calls involving

specialised subclasses — such calls are handled effectively by the HotSpot JVM through *method inlining* [KWM+08]. On the other hand, NextGen requires a new class for each new instantiation of a generic class type of the kind `C<`$\overline{\texttt{X}}$`>`. Even though these classes are in general small, their number can increase as the library of generic classes is used by different applications, so that the global size of the library can grow without bounds. It has been shown in [SC06] how this problem can be tackled effectively, by introducing a modified class loader that creates wrapper classes on the fly.

Moreover, heterogeneous translations such as the one proposed by NextGen doesn't scale particularly well to wildcards: in fact, types of the kind `List<? super T>` are contravariant, hence, the set of their supertypes is not closed: for any newly defined class `C` such that `C<:T`, type `List<? super C>` should be a supertype of `List<? super T>`. Therefore, whether such an approach would ever be able to support subtyping is still an open issue.

4.2 The EGO Compiler

The EGO compiler (Exact Generics on-Demand) is the result of a project developed in collaboration with Sun Microsystems with the goal of evaluating a compile-time reifying support to Java generics, which would not require changes to the JVM or to any other component of the Java platform. The solution conceived and developed is a sophisticated translation technique based on the type-passing style [VN00, Vir03b], where runtime type information is automatically created on a by-need basis, and cached for future utilisation.

In EGO's translation scheme, the generic type used to create an object is reified to an actual further argument (called *descriptor*) that is passed to the generic class constructor. Each generic class is augmented with an additional field in which this descriptor gets automatically stored for later accesses: each instance of a generic class points to its exact generic type. Such an information then accessed when necessary, e.g. when a cast operation occurs, when executing a type test, or when serialising the object.

Several critical issues had to be tackled in order to make this general idea a fully-fledged solution, including performance, compatibility, and so on. In

particular, EGO compiler has been developed with the following features:

Laziness Descriptors are created only the first time they are required, preventing any interference with usual Java class loading dynamics, and avoiding the problem of infinite polymorphic recursion [VN00];

Completeness The type-passing translation scheme is applied not only to generic classes, but also generic methods, generic inner classes, interfaces, and arrays; moreover in [CV08b] it has been shown how this scheme can be extended in order to support reification of wildcards;

Effectiveness A number of bridging techniques were introduced to deal with effectiveness issues such as interoperability between legacy and generic Java code and support to separate compilation — this is a crucial aspect of translation as the code generated by the EGO compiler might be executed by legacy clients;

Efficiency The need to obtain good performance results of the translated code pervasively affected all the aspects of the EGO translation scheme; this led to a sophisticated double-caching mechanism in which descriptors are stored in a global dictionary, called *descriptor registry*, but also cached into static fields of generic client classes for ensuring fast retrieval when performing type-dependent operations.

Performance measures executed over large-size benchmarks, like the `javac` compiler itself, have demonstrated the effectiveness of the EGO approach; such benchmarks report a general execution speed overhead within 10%, memory overhead within 5% and a class-size overhead within 15% [Vir05]. In the remainder of this section we provide a brief overview of the translation scheme exploited by the EGO compiler.

4.2.1 Type Descriptors in EGO

The EGO compiler represents the runtime type associated with a generic type of the kind `C<`$\overline{\texttt{T}}$`>` by means of a specialised data-structure called *class descriptor*. A class descriptor is implemented in the EGO runtime in terms of a class called `Cla`, whose definition is reported in Figure 3.12. More specifically, the class `Cla` is used for representing the runtime type-information associated

```
class Cla extends Desc {
   Class<?> theClass;
   Desc[] params;
   int[] annotations;
   Cla[] bounds;
   Cla super;
}
```

Figure 3.12: Class descriptor in EGO

with a generic type of the kind $\texttt{C<}\overline{\texttt{T}}\texttt{>}$, where `C` is a class declaration of the kind $\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{B}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{V}}\texttt{>}$. A class descriptor is structured in five main parts: *(i)* a `Class` object which stands for the "erased" class type `C`; *(ii)* an array of descriptors, used to keep track of the type parameters, containing descriptors for the types in $\overline{\texttt{T}}$; *(iii)* an array of integer values encoding the variance annotations associated with each type in $\overline{\texttt{T}}$ — as one or more types in $\overline{\texttt{T}}$ could be a wildcard argument; *(iv)* an array of descriptors representing the actual bound types $\texttt{[}\overline{\texttt{T}}\texttt{/}\overline{\texttt{X}}\texttt{]}\overline{\texttt{B}}$; and *(v)* a reference to the descriptor for $\texttt{[}\overline{\texttt{T}}\texttt{/}\overline{\texttt{X}}\texttt{]D<}\overline{\texttt{V}}\texttt{>}$ — the direct supertype of $\texttt{C<}\overline{\texttt{T}}\texttt{>}$.

For instance, EGO represents the type `List<? extends String>` by the class descriptor where: *(i)* the base type is the `Class` object respresenting the class type `List`; *(ii)* the type parameters array contains one element, namely, the type descriptor for `String`; *(iii)* the variance annotations array contains one integer element whose value is 1 (as 1 means '`? extends`'); the bounds array contains one descriptor for the type-variable bound, namely `Object`; *(v)* the super descriptor points to the top descriptor `Object`.

Other abstractions like raw types, generic inner classes, generic interfaces, generic methods and generic arrays are implemented through proper kinds of descriptors, which here are not discussed in detail for the sake of simplicity — e.g. the raw type for `List` is represented by a type descriptor of the kind `List<Any>`, where `Any` is a special descriptor used to represent an *unknown* type.

EGO provides a hash-consing mechanism to quickly store and retrieve descriptors [SK01]. When a descriptor is required, it is first searched in a

```
class List<X> {

   X head;
   List<X> tail;

   List(X head, List<X> tail){
      this.head = head;
      this.tail = tail;
   }

   static <Z> List<Z> nil() {
      return new List<Z>(null, null);
   }

   static <Z> List<Z> cons(Z head, List<Z> tail) {
      return new List<Z>(head, tail);
   }

   public static void main(String[] args) {
      Object o = List.cons(1, List.<Integer>nil());
      boolean res = o instanceof List<String>;
   }
}
```

Figure 3.13: A simple list class

global *descriptor registry* (an hashtable-like data structure): if such descriptor is not found in the registry, meaning that it is the first time that such descriptor is used in a type-dependent operation, a new descriptor will be allocated and registered there. Moreover, the reference to a descriptor is also stored locally to where it has been used, e.g. in a static field of the client class, leading to a particularly space- and time-efficient double-caching mechanism. The details of this kind of management are encapsulated into the `$crCLA()` method — this method is automatically added by the EGO compiler to a generic class of the kind `C<`$\overline{\texttt{X}}$`>`. This method is supplied a set of descriptors corresponding to the actual type parameters $\overline{\texttt{T}}$ of the generic type of the kind `C<`$\overline{\texttt{T}}$`>` for which a descriptor has to be retrieved; the method automatically handles all the tasks related to the creation of a new class descriptor of the kind `C<`$\overline{\texttt{T}}$`>`, such as creating and setting the parent descriptor, computing the descriptors for the actual bound types, and registering the descriptor.

```
class List<X> implements EGO.Parametric {
  protected Desc.Cla $d; // Instance descriptor
  static Desc[] $descs = new Desc[6]; //Local descriptor cache
  X head;
  List<X> tail;
  // Constructor (for backward compatibility)
  List(Object head, List tail) {
    this((Desc.Cla)$C(0), head, tail);
  }
  List(Desc.Cla $d, Object head, List<X> tail) {
    this.$d = $d;
    this.head = head;
    this.tail = tail;
  }
  static <Z> List<Z> nil(Desc.Meth $md) {
    return new List<Z>($B$D($md,0), null, null);
  }
  static <Z> List<Z> cons(Desc.Meth $md, Z head, List<Z> tail) {
    return new List<Z>($B$D($md,1), head, tail);
  }
  public static void main(String[] args) {
    Object o = List.cons($C(5), 1, List.<Integer>nil($C(4)));
    boolean res = $C(2).isInstance(o);
  }
  // Facility method to register descriptors
  public static Cla $crCLA(Cla[] params, int[] annotations) {
    Cla $v = Cla.reg(List.class, new Cla[]{params[0]});
    $v.setTypeVarBounds(new Cla[]{Desc._Object});
    Cla $cap = $v.capture();
    $v.setFath(Desc._Object);
    return $v;
  }
  // Facility method for retrieving closed descriptors
  private static Desc $C(int id) {
    if ($descs[id] != null) return $descs[id];
    switch (id) {
       case 0: return $descs[id] = $crCLA(new Cla[]{Desc._Any}, new int[]{0});
       case 1: return $descs[id] = Cla.reg(String.class);
       case 2: return $descs[id] = $crCLA(new Cla[]{$C(1)}, new int[]{0});
       case 3: return $descs[id] = Cla.reg(Integer.class);
       case 4: return $descs[id] = Meth.reg("nil",new Cla[]{$C(3});
       case 5: return $descs[id] = Meth.reg("cons",new Cla[]{$C(3});
     }
     return null;
  }
  // Facility method for retrieving open descriptors
  private static Desc $B$D(Desc d, int id) { ... }
}
```

Figure 3.14: Translation with EGO of code in Figure 3.13

4.2.2 Type-passing Technique in EGO

Figure 3.13 reports an example of generic class `List<X>` with standard `nil()` and `cons()` constructors, and Figure 3.14 its corresponding translation in EGO. An argument of type `Cla` is added to the constructor of `List<X>`, representing the generic type under instantiation. Its content will be stored in the EGO-generated field called `$d`: this is meant to contain information about the runtime type of the current instance, passed from the client that invokes the constructor. Note that the legacy constructor is kept to support compatibility with legacy code: there, the new constructor is called by passing a special descriptor for the raw type of `List`, namely, `List<Any>`.

The reification scheme exploited in an instance creation expression is of the general kind:

```
new List<T>(<args>) → new List<X>(/*Desc for List<T>*/,<args>)
```

namely, an appropriate expression — which is in charge of efficiently creating/retrieving the descriptor — is added as first argument of a generic class' constructor.

Descriptors can be of two different kinds: they can be independent of the current generic instantiation, such as e.g. type `List<String>`, which we call *closed descriptors*, or they may include type-variables defined in the enclosing scope, such as `List<Z>` in method `List.<Z>nil()`, which we call *open descriptors*. These two kinds of descriptor require different management [Vir05], delegated respectively to methods `$C()` and `$B$D()`, as shown in Figure 3.14: independently of their details, these methods are in charge of implementing the first caching level. For instance, method `$C()` looks first for the required descriptor in the static field `$descs` — otherwise a new descriptor is created and registered through method `$crCLA()`. Generic methods are handled similarly as shown for `<Z>cons()`: a method descriptor (instance of class `Desc.Meth`) is passed as first argument in the invocation, carrying information about the instantiation of the method type parameters.

A type-dependent operation involving a generic type exploits the runtime type information stored in the `$d` field. For instance, let `v` stand for the

expression used to access the descriptor for `List<String>` — e.g. `$C(2)` as in method `main()` — we have the following translations:

```
o instanceof List<String> → v.isInstance(o)
(List<String>)o → (List<String>)v.cast(o)
```

Methods `isInstance()` and `cast()` (of class `Cla`) simply try to access `o`'s descriptor: if this is possible it means the objects has been created from a generic class, hence they simply check whether such a descriptor corresponds to a descriptor for any supertype of `List<String>` — this is accomplished by performing a dynamic subtyping test. Other kinds of runtime introspection, such as e.g. those required to support persistence, are implemented in a similar fashion.

Chapter 4

Reified Generics in the Java Virtual Machine

The J2SE 5.0 platform lacks a true runtime support for both generic types and wildcards; those types are in fact translated into legacy Java types during compilation by a process called *type-erasure*. This leads to problems with many existing Java technologies such as *Reflection*, *Serialization*, *Java Beans* and *RMI*. In this thesis we develop[1] a sophisticated type-passing technique for addressing the problem of reification of generic types in the Java programming language; this approach — first pioneered by the so called EGO translator [Vir05] (see Section 4.2) — is here turned into a full-blown solution which reifies generic types inside the Java Virtual Machine (JVM) itself, thus overcoming both the performance penalties and the compatibility issues of the original EGO translator. The JVM used for building our prototype is the CVM (version 1.0.2), which is part of the CDC (Connected Device Configuration) configuration of the J2ME platform[2] — hence the name GCVM (Generic CVM).

Our goal is to provide a full-blown reification support for generics and wildcards which allows for a more coherent and safe version of the Java programming language. Most noticeably, in the GCVM there is no distinction

[1] This research project has been funded by Sun Microsystems.

[2] The reference implementation discussed throughout this chapter is the CVM version 1.0.2. Our aim was to use a CVM implementation without Just-In-Time compilation support (JIT), which would make the development of the reified CVM prototype an harder task.

beetwen reifiable and non-reifiable types (see Section 3.2): all types (but type-variables) are *reifiable* — consequently, the GCVM must provide runtime support for execution of type-dependent operations involving generic types, such as generic cast, generic `instanceof`, generic array creation, and so forth.

```
Object cs = new Cell<String>("One");
Cell<Integer> ci = (Cell<Integer>)cs; //CCE
boolean res = cs instanceof Cell<String>;
Object[] oarr = new Cell<? super Integer>[5];
oarr[2] = new Cell<String>("Two") //ASE
```

Note that the the semantics of type-dependent operations involving generic types is here enforced *during execution*: the GCVM issues a runtime error whenever e.g. an erroneous type-conversion — as in `(Cell<Integer>)cs` — or a bad array store — as in `oarr[2] = "Hello!"` — is detected; hence, the language deployed by the GCVM does not suffer from the heap pollution problem.

1 Architecture Overview

Java generics are implemented through type-erasure (see Section 3.1), which literally erases all generic types signatures from a generic Java program during compilation. As an example, generic class declarations are translated to *monomorphic* class declarations (e.g. `List<E>` is turned into `List`), and every type-dependent operation involving generic types is translated, too (e.g. `new List<Integer>(...)` becomes `new List(...)`). Consequently, generic types and wildcards never enter the runtime domain of the JVM, so that type-dependent operations involving generic types are subject to some unavoidable restrictions.

The first problem we have to face is to define an extension to the current classfile format [LY99, Micb], so that exact type information required by type-dependent operations can be stored in the classfile; this can be accomplished in two ways, either by extending the JVM instruction set, or by exploiting some form of *bytecode instrumentation*. The former technique, which has been successfully exploited in runtime approaches as in [MBL97], inevitably poses serious compatibility issues, as it typically introduces new bytecode

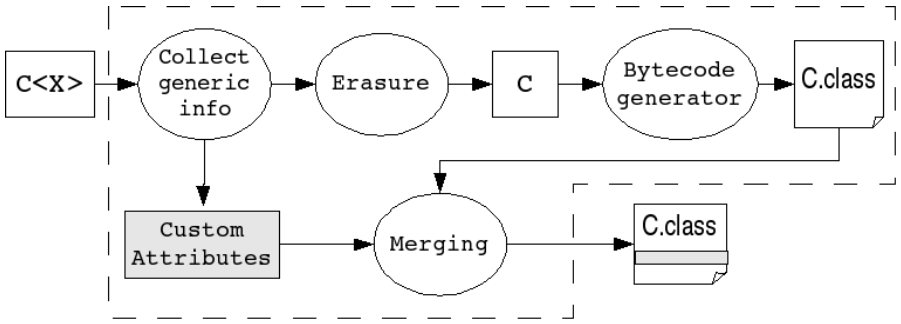


Figure 4.1: Bytecode Instrumentation

instructions that would not be understood by legacy JVMs. Our goal is to encode full generic type signatures in a backward compatible fashion — this can be accomplished by making generic type signatures available via custom *classfile attributes*. In fact, as stated in [LY99], non-custom bytecode attributes are simply skipped by a legacy JVM; consequently, legacy JVMs would still be able to execute generified bytecode — this is also the way in which generic type signatures for class/method/member declarations were added in J2SE 5.0 (see Section 3.1).

Requirement 1. The reification support must be able to recover the type information that has been lost during type-erasure. Such information must be made available via custom bytecode attributes, so that the generified classfiles might still be executed by legacy (i.e. non-generic) JVMs.

Our compilation strategy is described in Figure 4.1. This process involves the following steps: first, generic types' signatures are collected before the erasure process takes place; a plain Java classfile is thus generated as usual — this classfile contains no generic type signatures (expect for the ones in the `Signature` attributes [Micb]); finally, the previously collected generic types' signatures are merged into the erased classfile as *custom attributes*, thus obtaining a *generified* classfile. Recalling from Section 3.2, a type-dependent operation involving a generic type of the kind $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ is translated into a type-dependent operation involving an erased type `C`. The reification support must be able to reconstruct the original type information associated with a type-

dependent operation, as this information might be required during execution — examples are the type of an instance creation expression of the kind `new List<String>()`, or the target type of an instance test of the kind `(List<? super Integer>)obj`, etc. In our approach, a type-dependent instruction involving a generic type of the kind `C<`$\overline{\texttt{T}}$`>` is decorated with a pointer to a *type descriptor* — used to represent the unerased signature of `C<`$\overline{\texttt{T}}$`>`. As an example, consider the following piece of code:

```
List<String> ls = new List<String>();
```

Which, after erasure, translates to:

```
List ls = new List();
```

Assuming that the type descriptor for `List<String>` is available in the generified classfile, the above code could be rewritten as follows:

`List ls = new List();` → `List<String>'s type descriptor`

Where the arrow denotes a dependency between the `new` instruction and the type descriptor representing the (generic) type of the instance creation expression.

Requirement 2. The reification support must associate each type-dependent opcodes involving a generic type of the kind `C<`$\overline{\texttt{T}}$`>` with a corresponding type descriptor for `C<`$\overline{\texttt{T}}$`>` — such descriptor must be made available in the generified classfile. The type-dependent opcodes are:

- `new` — object allocation;
- `anewarray`, `multianewarray` — (multi)array allocation;
- `aastore` — array store;
- `instanceof` — dynamic instance test;
- `checkcast` — dynamic type-conversion;
- `invokevirtual`, `invokespecial`, `invokestatic` — method calls.

Note that this is essentially the same type-passing strategy implemented in the EGO translator (see Section 4.2). In EGO, type information is encoded in terms of a descriptor object — an instance of the `Cla` class. The descriptor

object is supplied to the constructor of a generic class by the client class, and then stored in a synthetic field of the generic class being instantiated — type-dependent operations are simply implemented in terms of operations provided by the descriptor class.

Conversely, no Java artifact is required here — the additional type information is directly encoded inside custom classfile attributes; such information is then reconstructed inside the GCVM when the generified classfile is first loaded. In particular, the GCVM must be able to recover the exact type information stored in the custom classfile attributes so that such information can be used when executing type-dependent opcodes involving generic types.

For instance, when a new generic object is allocated, the GCVM must be able to recover the unerased generic type of the instance creation expression — this is accomplished by accessing the custom classfile attributes stored in the generified classfile. A runtime representation of such (possibly generic) type is then attached to the newly created instance; this information might be needed at a later stage — e.g. when executing a type-dependent operation (such as a type-conversion or an instance test) on that generic instance.

Requirement 3. The reification support must be able to reconstruct the additional type information stored in a generified classfile. Moreover, each type-dependent operation for which a type-descriptor is available, must be tagged explicitly during class loading.

Requirement 4. The reification support should *(i)* attach to each generic object a type representation of the kind `C<`$\overline{\texttt{T}}$`>` where the types in $\overline{\texttt{T}}$ correspond to the actual type parameters associated with the object's generic type, and *(ii)* provide a means to retrieve the exact type of a generic object e.g. during type-dependent operations.

2 The Generified Classfile Format

In this section we discuss how type descriptors are *stored* into a classfile and how such descriptors can be *linked* to type-dependent instructions. In the following, the structure of classfile attributes is given in term of C-like `struct` data-structures (the same notation is used in [LY99]), where the type tags `u1`, `u2` and `u4` denotes 1, 2 and 4 bytes-wide values, respectively.

```
struct DescriptorTable {
   u2 attribute_name_idx;
   u4 attribute_length;
   u2 descriptors_count;
   {  u1 tag;
      u2 descriptor_length;
      u1 info[descriptor_length];
   } descriptor_info[descriptors_count];
}
```

Figure 4.2: The `DescriptorTable` class attribute

2.1 The `DescriptorTable` Attribute

All type descriptors used by a given class are stored as *entries* in a *class* attribute, called `DescriptorTable`. The `DescriptorTable` attribute (see Figure 4.2) acts as an extended constant pool [LY99], used to store all descriptors entries referred to by type-dependent operations in a given class. The amount of descriptor entries stored in the descriptor table is `descriptors_count` — consequently, a descriptor entry can be stored at a position i, where $0 \leq i \leq$ `descriptors_count`. As shown in Figure 4.2, all descriptor entries share the same *header*. This common header provides hints on the kind (`tag`) and the length in bytes (`descriptor_length`) of the descriptor entry being encoded. There are several kinds of descriptor entries — one for each kind of Java type, such as class, array, methods, type-variables; in the remainder of this section we provide a brief overview of the main kinds of descriptor entries encoded in a generified classfile.

2.1.1 Class Descriptors

A generic class type of the kind `C<`$\overline{\texttt{T}}$`>` is represented by a `ClassDescriptor` entry; class descriptors can be used to encode the type of an instance creation expression, the target type of an instance test/type-conversion, the parameter type of a generic type, the element type of an array and so on. Class descriptors can be used to encode either generic types, such as `List<String>`, `Pair<Z,Integer>`, wildcard types such as `List<? extends`

```
struct ClassDescriptor {
   u1 tag;
   u2 descriptor_length;
   u2 enclosing_idx;
   u2 name;
   u2 params_length;
   u2 params[params_length];
   u1 annotations[params_length];
}
```

Figure 4.3: Structure of a `ClassDescriptor`

`Integer>`, `Pair<?,?>`, or non-generic types such as `Object`, `String`, etc.

A class descriptor entry is made up of the following fields:

enclosing_idx points to a class descriptor entry. This field is used when the class descriptor being encoded is an inner class; in that case `enclosing_idx` points to the descriptor entry for the *innermost* enclosing class/method. If the class type being encoded is a toplevel class, this field is set to `-1`;

name points to a `CONSTANT_Class_info` constant pool entry defining the name of the class type being encoded;

params type parameter descriptors list (whose size is `params_length`). The i-th element points to a descriptor entry for the i-th actual type parameter of the generic class type being encoded. Valid entries are either of kind `ClassDescriptor` or `TypeVarDescriptor`;

annotations the variance annotation array (whose size is `params_length`). The i-th element points to an integer constant, where the values 0, 1, 2 and 3 are used to tag an *invariant* parameter (e.g. `Number`), a *covariant* parameter (e.g. `? extends Number`), a *contravariant* parameter (e.g. `? super Number`) and a *bivariant* parameter (of the kind `?`), respectively.

For example, the wildcard type `List<? super Integer>` is represented by a `ClassDescriptor` entry whose *(i)* `enclosing_idx` is `-1` (since `List` is a toplevel class), *(ii)* `name_idx` points to a constant pool entry for `List`, *(iii)* `params_length` is `1`, *(iv)* `params` contains an index pointing to the class

```
struct MethodDescriptor {
   u1 tag;
   u2 descriptor_length;
   u1 flags;
   u2 name;
   u2 receiver_idx;
   u2 params_length;
   u2 params[params_length];
}
```

Figure 4.4: Structure of a `MethodDescriptor`

descriptor entry for `String` and *(v)* `annotations` contains the annotation value 2 (as `List<? super Integer>` has a contravariant parameter type).

2.1.2 Method Descriptors

A generic method type of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{X}}$`>m()` is represented by a `MethodDescriptor` entry. Method descriptors are mainly used to encode the type of a method in a generic method call, as `List.<String>cons()`, etc. A `MethodDescriptor` entry is made up of the following fields:

flags a general-purpose 8-bits mask, used to record various details about a generic method call. These flags can be used e.g. to encode specific Java modifiers such as `static` or `final`, or to explicitly mark interface or *captured* calls — which need special treatment by the runtime environment, as we shall see in Section 4.3.2;

name_idx points to a constant pool entry defining the name of the method being encoded. Valid constant pool entries are either of kind `CONSTANT_MethodRef_info` or `CONSTANT_InterfaceMethodRef_info`;

receiver_idx points to a class descriptor entry representing the type of the *receiver* in a given generic method call;

params type parameter descriptors list (whose size is `params_length`). The i-th element points to a descriptor entry for the i-th actual type parameter of the generic method type being encoded. Valid entries are either of kind `ClassDescriptor`, or `TypeVarDescriptor`.

```
struct ArrayDescriptor {
   u1 tag;
   u2 descriptor_length;
   u2 element_idx;
   u2 depth;
}
```

Figure 4.5: Structure of a ArrayDescriptor

For example, the generic method List.<Integer>cons() is represented by a MethodDescriptor entry whose *(i)* receiver_idx is -1 (as the method being called is static), *(ii)* name_idx points to a constant pool entry for List.cons, *(iii)* params_length is 1 and *(iv)* params contains an index pointing to the ClassDescriptor entry for Integer.

2.1.3 Array Descriptors

A generic array type of the kind class C<$\overline{\text{T}}$>[] is represented by an ArrayDescriptor entry; array descriptors can be used to encode the type of an array creation expression, the target type of an instance test/type-conversion, the parameter type of a generic type, and so on. Array descriptors can be used to encode either generic array types, such as Pair<Z,? super Integer>[][] or non-generic array types such as Object[], Integer[][], etc. An ArrayDescriptor entry is made up of the following fields:

element_idx an index pointing to a ClassDescriptor entry. This field is used to represent the element type of the array;

depth the array's depth (must be less than $2^{16} - 1$).

For instance, a generic array of the kind List<Float>[][] is represented by an array descriptor entry whose *(i)* element_idx points to the class descriptor entry for List<Float> and *(ii)* depth is set to the value 2.

2.1.4 Type-variable Descriptors

A type-variable of the kind X is represented by a TypeVarDescriptor entry; type-variable descriptors are used to encode the type parameters of a generic type in several contexts, such as instance creation expressions, generic method

```
struct TypeVarDescriptor {
   u1 tag;
   u2 descriptor_length;
   u2 owner;
   u2 slot;
   u2 class_bound_idx;
   u2 interface_bounds_length;
   u2 interface_bounds[interfaces_bounds_length];
}
```

Figure 4.6: Structure of a `TypeVarDescriptor`

calls, array creation expressions, and so on. A type-variable descriptor can also be used to encode the target type of an instance test/type-conversion or the element type of a generic array. Example of usages of type-variable descriptors are `List<X>`, `Pair<? super Z,Integer>[]`, `Z[][]`.

A `TypeVarDescriptor` entry is made up of the following fields:

slot the position in which the type-variable appears in a generic class/method declaration;

owner points to a constant pool entry representing the *owner* of the type-variable whose descriptor is being encoded. Valid entries are either of kind `CONSTANT_ClassRef`, `CONSTANT_MethodRef` or `CONSTANT_InterfaceMethodRef`;

class_bound_idx points to a `ClassDescriptor` entry for the *class bound* associated with a given type-variable;

interface_bounds a list (whose size is `interfaces_bounds_length`) of descriptors for all the *interface bounds* associated with a given type-variable. The i-th element points to a `ClassDescriptor` entry defining the i-th interface bound of the type-variable being encoded[3].

Given the generic class `List<E extends Number & Comparable<E>>`, the type-variable `E` of `List` — written `List#E` — is represented by a type-

[3]Both `class_bound_idx`, `interface_bounds_length` and `interface_bounds` are redundant; in principle, they could be parsed from the `Signature` attribute of the class/method declaration where a type-variable is defined [Micb].

```
struct DescriptorMap {
   u4 attribute_name_index;
   u4 attribute_length;
   u2 maps_count;
   { u2 PC;
     u2 desc_index;
   } map_info[maps_count];
}
```

Figure 4.7: The DescriptorMap method attribute

variable descriptor entry where: *(i)* `owner` points to a constant pool entry for `List`, *(ii)* `slot` is `1` (`E` is the first type-variable of `List`), *(iii)* `class_bound_idx` is an index pointing to the class descriptor entry for `Number`, *(iv)* `interface_bounds_length` is `1` and *(v)* `interface_bounds` contains an index to the class descriptor entry for `Comparable<E>`.

2.2 The DescriptorMap Attribute

In the previous section we discussed how type descriptors can be encoded into a plain Java classfile, as entries of the `DescriptorTable` attribute. In this section we focus our attention on how such descriptors can be linked to type-dependent instructions. This task is accomplished by introducing another custom bytecode attribute called `DescriptorMap`, which is attached to *each* Java method containing one or more type-dependent operations involving generic types.

The structure of a `DescriptorMap` attribute is quite simple (see Figure 4.7); a `DescriptorMap` attribute is essentially a table, used to store entries of the kind *(PC, descIndex)*. Such entries are used to link a type-dependent opcode, whose index inside the method's `code` attribute [LY99] — or program counter — is *PC*, to a type descriptor — whose index inside the `DescriptorTable` attribute is *descIndex*.

The code in Figure 4.8a defines a simple generic class, namely `Pair`, where the two type-variables `X` and `Y` are used to abstract over the concrete types of the fields `x` and `y`, respectively. `Pair` defines a generic method, namely `chgSecond()`, that accepts a value of type `Z` (where `Z` is a method

```
class Pair<X,Y>{
   X x; Y y;

   Pair(X x, Y y) {
      this.x = x;
      this.y = y;
   }
   Pair<Y,X> swap(){
      return new Pair<Y,X>(y,x);
   }
   <Z> Pair<Z,Y> chgFirst(Z z){
      return new Pair<Z,Y>(z,y);
   }
   final <Z> Pair<X,Z> chgSecond(Z z){
      return swap().<Z>chgFirst(z).swap();
   }
}
```

(a) The generic class `Pair`

```
class TestPair{
   public static void main(String[] args){
      String one="one";
      Integer two=new Integer(2);
      String three="three";
      Pair<String,Integer> pair1=new Pair<String,Integer>(one,2);
      Pair<String,String> pair2=pair1.<String>chgSecond(three);
   }
}
```

(b) The client class `TestPair`

Figure 4.8: A simple generic class and its client: `Pair` and `TestPair`

type-variable) and returns a new object of type `Pair<X,Z>` — that is, a new pair where the second element has been replaced. The code in Figure 4.8b defines a client class, namely `TestPair`, that performs some type-dependent operations involving the generic type `Pair<X,Y>`: first, a new object of type `Pair<String,Integer>` is created; secondly, this newly created object is used as a receiver in the generic method call to `chgSecond()` — this yields a new pair object of type `Pair<String,String>`.

Therefore, the `DescriptorMap` attribute associated with the method

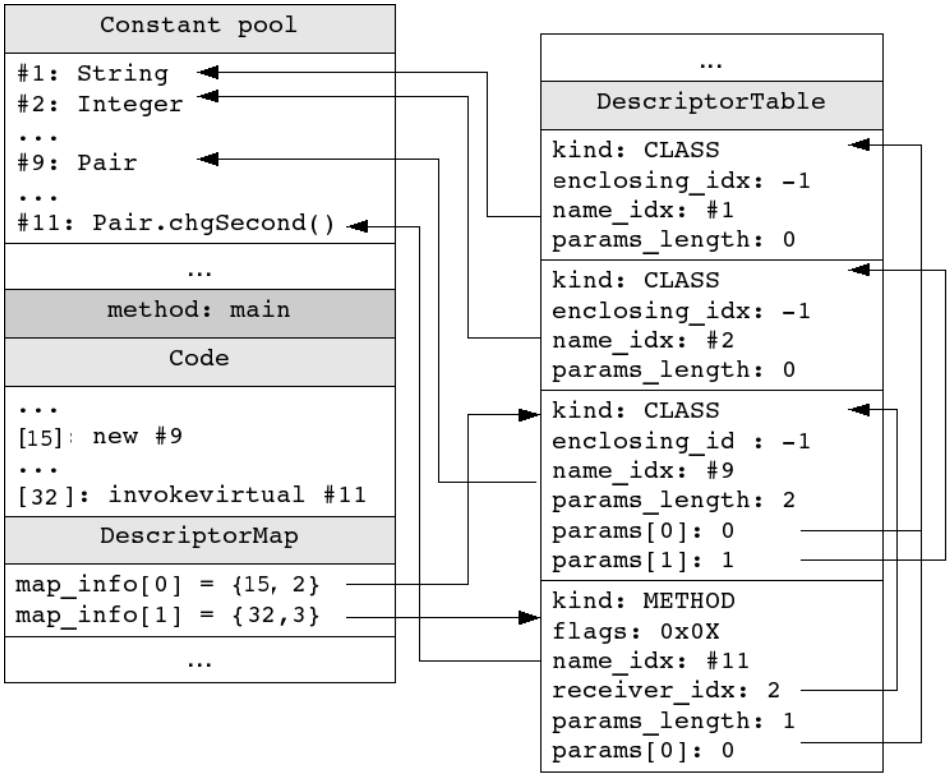


Figure 4.9: `DescriptorMap` in action

`main()` must defines the following two entries (see Figure 4.9):

1. the entry $\{15, 2\}$ is used to associate the `new` opcode (whose program counter value is 15) with the class descriptor entry for `Pair<String,Integer>`, stored in the second slot of `TestPair`'s descriptor table.
2. the entry $\{32, 3\}$ is used to associate the `invokevirtual` opcode (whose program counter value is 32) with the method descriptor entry for `Pair<String,Integer>.<String>chgSecond()`, stored in the third slot of `TestPair`'s descriptor table.

2.3 The `SuperDescriptor` Attribute

A generic class of the kind `C<`$\overline{\texttt{X}}$`>` might have one or more *generic supertypes* of the kind `D<`$\overline{\texttt{Y}}$`>`. It is crucial that the type information associated with the

```
struct SuperDescriptor {
   u4 attribute_name_index;
   u4 attribute_length;
   u2 super_idx;
   u2 interfaces_count;
   u2 interfaces[interfaces_count];
}
```

Figure 4.10: The `SuperDescriptor` class attribute

generic supertypes of a generic class is preserved under type-erasure — that is, given a type-descriptor of the kind $C\langle\overline{T}\rangle$, the reification support must be able to reconstruct some descriptors of the kind $[\overline{T}/\overline{X}]D\langle\overline{Y}\rangle$ — we call such descriptors the *parent* descriptors.

The type information stored in the parent descriptors must be accessed when performing type-dependent operations such as type-conversion or `instanceof`, that are typically implemented in terms of a *dynamic* subtyping test. As an example consider the class `java.util.Vector`, which is part of the Java Collection Framework. This class is declared as follows [Mica]:

```
public class Vector<E> extends AbstractList<E> implements
       List<E>, RandomAccess, Cloneable, Serializable{...}
```

A generic type of the kind `Vector<Integer>` has two generic supertypes — `AbstractList<Integer>` and `List<Integer>`, respectively; those types are obtained from the types in the `extends`/`implements` clauses, where all occurrences of the type-variable `E` have been replaced with the type `Integer`. For instance, consider the following code:

```
Object o = new Vector<Integer>();
if (o instanceof List<Integer>) { ... }
```

In order to execute the instance test, the runtime environment must check that `Vector<Integer> <: List<Integer>` — this intuitively amounts at recursively scanning all the parent descriptors of `Vector<Integer>` until the descriptor for `List<Integer>` (or `Object` if the test fails) is found.

An additional custom attribute, namely `SuperDescriptor`, is available in the generified classfile; this attribute is used to keep track of the parent descriptors associated with a generic class of the kind `C<`$\overline{\texttt{X}}$`>`[4].

A `SuperDescriptor` attribute is made up of the following fields (see Figure 4.10):

`super_idx` points to a class descriptor entry representing the (possibly generic) supertype of the class being encoded. If the class being encoded has either an empty or a non-generic `extends` clause, this field is set to `-1`;

`interfaces` an array of descriptor entries, of size is `interfaces_length`, where the i-th element points to a class descriptor entry for the i-th generic superinterface of the class being encoded.

As an example, the `SuperDescriptor` attribute of the generic class `Vector<E>` has the following layout: *(i)* `super_idx` is the index of the class descriptor for `AbstractList<E>` and *(ii)* `interfaces` is an array containing an index that points to the class descriptor entry for `List<E>` (as `Vector<E>` has only *one* generic superinterface in its `implements` clause).

3 The gCVM Runtime

In this section we develop an extension to the CVM runtime that provides runtime support for generic types/wildcards. This section is structured into three main parts, discussing the following topics:

Classloading We show how the custom classfile attributes described in Chapter 2 are represented in terms of internal structures of the gCVM; more specifically, we focus on the bytecode instrumentation carried out during class loading that ensures an efficient retrieval of type descriptors associated with type dependent opcodes, without requiring a global lookup into the method's `DescriptorMap` attribute.

[4]Such information could in principle be reconstructed from the `Signature` attribute included in JDK5.0 classfiles [Micb]. This is, however, rather expensive, as the signature attribute is essentially a string — which the gCVM would need to parse during class loading.

Representation of generic instances We show how generic types are represented by the GCVM runtime and how the support for generic types affects the object layout in the Java heap.

Interpreter We analyse how type-dependent opcodes are executed by the GCVM interpreter; first we discuss how descriptor entries are resolved and linked to runtime descriptors; then we show how the interpreter executes the *generic* counterparts of some remarkable type-dependent opcodes such as `new`, `invokevirtual`, etc.

3.1 Runtime Overview

Figure 4.11 shows a snapshot of the GCVM while running the `TestPair` class, whose code is shown in Figure 4.8b. Recalling from section 2.2, `TestPair` defines a `main()` method leveraging two type-dependent operations, namely a generic instance creation and a generic method call; the opcodes corresponding to such type-dependent operations are linked — via the `DescriptorMap` attribute of `main()` — to entries in `TestPair`'s descriptor table: first the `opc_new` opcode corresponding to the instance creation expression (`new Pair<String,Integer>()`) points to a descriptor table entry representing the generic type `Pair<String,Integer>`; secondly, the `opc_invoke_virtual` opcode corresponding to the generic method call `pair1.<String>chgSecond()` points to a descriptor table entry representing the generic method type `Pair<String,Integer>.chgSecond<String>()`.

When `TestPair` is first loaded, the GCVM creates an internal representation for `TestPair`'s classfile which is then stored in the GCVM class table; this data-structure, called `CVMClassBlock`, is used to collect information that must be available when executing a method in `TestPair`, such as the virtual method table, the parent class loader, the constant pool, and so forth. The `CVMClassBlock` of `TestPair` points to another internal data-structure, namely `CVMDescriptorTable`, used to map the contents of `TestPair`'s descriptor table attribute — execution of type-dependent opcodes might need to dynamically access the contents of the descriptor table of a given class.

During class loading, all opcodes linked to a a type descriptor via the `DescriptorMap` attribute are *rewritten*, so that a synthetic opcode called

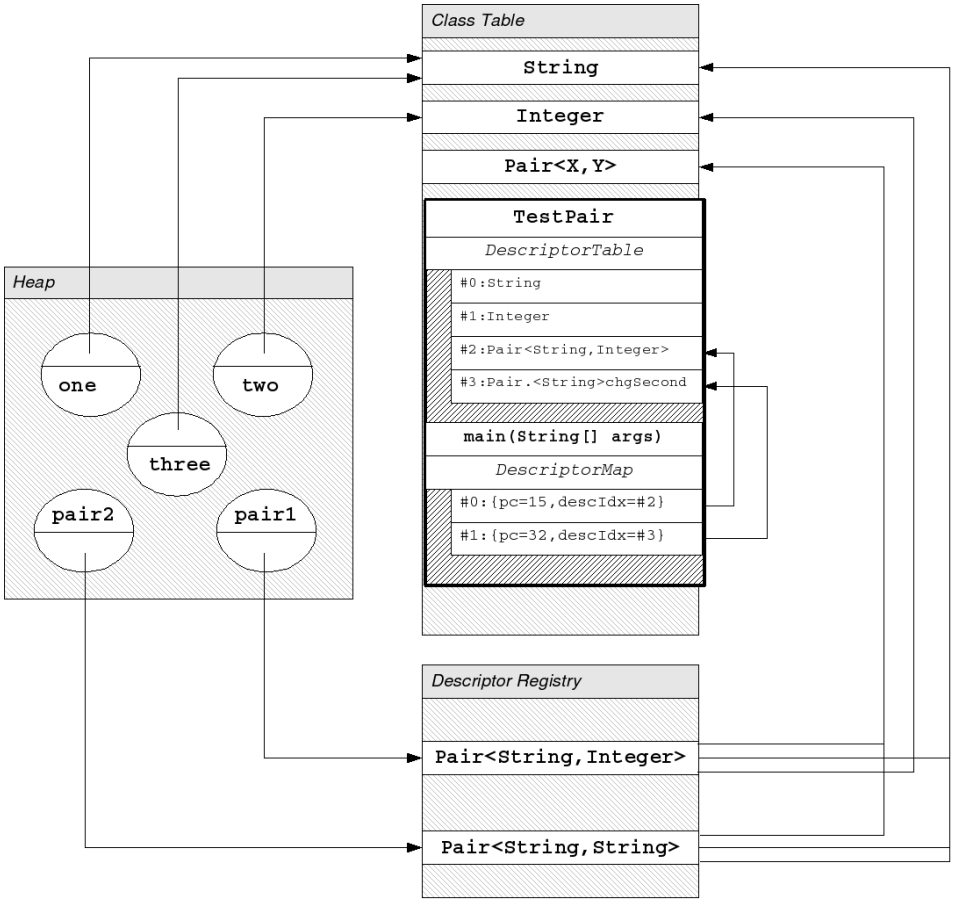


Figure 4.11: The GCVM in action

`opc_load_desc #idx` is prepended, where `#idx` is an index to a descriptor table entry; this special instruction is used to inform the GCVM interpreter that the next opcode to be executed is a type-dependent opcode exploiting the type descriptor stored at position `desc_idx` in the current class' descriptor table. For example, a generic instance creation expression of the kind `new Pair<String,Integer>()` is rewritten into two bytecodes, namely an `opc_load_desc` — which points to the descriptor table entry for the generic type `Pair<String,Integer>` — followed by the original `opc_new`. Consequently, the exact type of the instance creation expression is `Pair<String,Integer>`, rather than the erased type `Pair`.

When a `load_desc` opcode is first executed, the descriptor table entry it

refers to must be *resolved*; this resolution process is quite similar to the one involving constant pool entries for non-generic type-dependent instructions — in fact, a descriptor table can be viewed as an extension to the class' constant pool. The resolution process typically leads to the creation of a *runtime* descriptor, which can thus be used to perform exact subtyping tests (e.g. when executing `opc_instanceof` or `opc_checkcast` opcodes) or to represent the runtime type of a given generic instance. The GCVM exploits a double-caching mechanism similar to the one discussed in Section 4.2: type-descriptors are cached in a shared table called *descriptor registry*. When a descriptor is needed by an application, the registry is first searched for an existing matching descriptor; if none is found, the descriptor is created and added to the registry. The GCVM exploits two different layouts for encoding generic and non-generic instances: the header of a legacy (i.e. non-generic) object points to a `CVMClassBlock` in the GCVM class table; dually, the header of a generic object points to a runtime descriptor in the GCVM descriptor registry, which, in turn, points back to a `CVMClassBlock`. Therefore, class-related information, which might be required when executing a type-dependent operations on generic objects, is still available at the cost of an extra-level of indirection.

In the following sections, the terms "descriptor entry" and "descriptor" are used to mean rather different concepts: by descriptor entry we always mean a compact and *static* symbolic representation of some generic type that can be used as template to generate many different runtime types — we call such representations *runtime descriptors*. It is possible for two distinct classfiles to contain the same descriptor entry; this can happen if two client classes exploit the same generic type e.g. `List<Integer>`. However, during execution, there will be only one (shared) copy of the runtime descriptor for `List<Integer>`. In other words, a descriptor entry is nothing more than a symbolic representation of a type — either a generic type, a wildcard type or a type-variable; as such, it cannot be used *as it is* by the interpreter in order to execute type-dependent opcodes — as a constant pool entry cannot be used to represent the runtime type of a non-generic object.

```
struct CVMDescriptorTable {
  CVMUint16 nentries;
  struct {
    CVMUint8 tag;
  } CVMDescriptorTableEntryHeader* entries[nentries];
};
```

Figure 4.12: The `CVMDescriptorTable` data structure

3.2 Descriptor Table

The GCVM collects class-related information such as constant pool, fields, methods, etc. inside an optimised in-memory data-structure called `CVMClassBlock` — therefore, a `CVMClassBlock` can be thought of as an internal representation of a Java classfile. This process of turning classfile chunks into internal data-structures can be viewed as a necessary optimisation step: the classfile of a given class `C` is accessed and parsed only once, namely when `C` is first loaded; after class loading, the information stored in `C`'s classfile is efficiently retrieved via the `CVMClassBlock` associated with `C`.

Note that the class loading process must handle the custom classfile attributes described in section 2; more specifically, the `CVMClassBlock` of a given class must be associated with an internal data-structure, representing the class' descriptor table — we call this structure `CVMDescriptorTable`. A `CVMDescriptorTable` (see Figure 4.12) is essentially an array of descriptor table entries; all descriptor entries share a common *header*, defining an 8-bit mask which is used to distinguish between different kinds of entries, as well as to encode the entry *state* — a descriptor entry can be either *resolved* or *unresolved*, as we shall see in Section 3.5. In the remainder of this section we focus on class and method descriptor entries — though the GCVM provides an internalised representation for all the kinds of descriptor entries specified by the `DescriptorTable` attribute (see Section 2).

Descriptor entries can assume different layouts depending on whether they are *closed* or *open*; a closed descriptor entry is used to represent a type that does not contain type-variables, such as `List<Integer>` or

```
struct CVMClassEntry{
  CVMDescriptorTableEntryHeader header;
  union {
    CVMClassDescriptor* desc;
    CVMClassBlock* cb;
    CVMUint16 clazz_cpindex;
  } class;
  CVMUint16 enclosingIdx;
  CVMUint16 nparams;
  CVMUint16 params[nparams];
  CVMUint16 annotations[nparams];
};
```

Figure 4.13: The `CVMClassEntry` structure

`Pair<String,Integer>.<String>chgSecond()`. Dually, an open descriptor entry is used to represent a type containing one or more type-variables, such as `List<Y>`, `Pair<String,Z>.<Y>chgSecond()`. There is a fundamental difference between closed and open entries: the runtime type associated with a closed entry does not depend on the runtime type of the class/method in which the entry is used — this allows for a more compact and efficient implementation. On the other hand, a single open descriptor entry might be associated with several runtime types, one for each possible instantiation of the type-variables the entry refers to. As such, open descriptor entries cannot be used to form runtime types: they must first undergo a heavy-weight resolution process where each type-variable in the entry is replaced for an actual type — this is accomplished by looking at the runtime descriptor of the enclosing class/method (see Section 4.1).

3.2.1 Class Entries

A class descriptor entry for a generic type of the kind $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ is parsed into a data-structure called `CVMClassEntry`. Intuitively, a class entry contains a pointer to the erased-type `C`, as well as an array of descriptor entries for the types in $\overline{\texttt{T}}$. The `CVMClassEntry` structure is made up of the following fields:

class a generic class type of the kind $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ — when the entry is unresolved, this field is simply an index to a constant pool entry. When the class

entry becomes resolved, this field stores a pointer to either the runtime descriptor for `C<`$\overline{\texttt{T}}$`>`, or, alternatively, to the `CVMClassBlock` for `C` — depending on whether this descriptor entry is closed or open;

enclosingIdx the index of the enclosing descriptor inside the class' descriptor table. If the type to be represented is a toplevel type, this field is set to `-1`. Valid entries are either of kind `CVMClassEntry` (for member classes) or `CVMMethodEntry` (for local or anonymous classes);

params an array of size is `nparams`, where the i-th element is the index to a descriptor entry for the i-th actual type parameter of the generic type `C<`$\overline{\texttt{T}}$`>`. Valid entries are either of kind `CVMClassEntry` or `CVMTypeVarEntry`;

annotations an array of size is `nparams`, where the i-th element is the annotation value associated with the i-th actual type parameter of the generic type `C<`$\overline{\texttt{T}}$`>`; the encoding for the annotation values is identical to the one discussed in Section 2.1.1.

For instance, a closed, unresolved descriptor entry for the generic type `Pair<String,Integer>` is a `CVMClassEntry` structure where: `enclosing_idx` is set to `-1` (`Pair` is a toplevel class); `class` points to a constant pool entry for `Pair`; `nparams` is `2` (as `Pair` has two type-variables); `params` contains two indices pointing to the descriptor entries for `String` and `Integer`, respectively; and `annotations` is a two-element array containing two 0 — as both type parameters `String` and `Integer` are *invariant*. When this entry becomes resolved, the constant pool pointer in `class` is replaced with a pointer to the runtime descriptor for `Pair<String,Integer>`.

3.2.2 Method Entries

A method descriptor entry for a generic method call of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()` is parsed into a data-structure called `CVMMethodEntry`. Intuitively, this entry contains a pointer to the erased method type `C.m`, an array of descriptor entries for the types in $\overline{\texttt{T}}$, and an index to a descriptor entry for the receiver type `C<`$\overline{\texttt{S}}$`>`. The `CVMClassEntry` structure is made up of the following fields:

```
struct CVMMethodEntry{
  CVMDescriptorTableEntryHeader header;
  union {
    CVMMethodDescriptor* desc;
    CVMMethodBlock* mb;
    CVMMethodTypeID method_index;
  } method;
  CVMUint8 flags;
  CVMUint16 pos;
  CVMUint16 receiver;
  CVMUint16 nparams;
  CVMUint16 params[nparams];
};
```

Figure 4.14: The `CVMMethodEntry` structure

flags a 8-bits mask which reflects the contents of the bit mask in the method descriptor entry stored in the generified classfile (see Section 2);

class the type of a generic method call of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()` — when the entry is unresolved, this field is simply an index to a constant pool method entry. When the descriptor method entry becomes resolved, this field stores a pointer to either the runtime descriptor for `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()` or, alternatively, to the `CVMMethodBlock` for `C.m()` — depending on whether this descriptor entry is open or closed;

receiver the index of the class descriptor entry for the receiver type `C<`$\overline{\texttt{S}}$`>` associated with a generic method call of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`;

pos an index used to handle *virtual* generic method calls in an efficient fashion (see Section 4.2 for further details);

params an array of size `nparams`, where the i-th element is an index to a descriptor entry for the i-th actual type parameter of the generic method type `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`. Valid entries are either of kind `CVMClassEntry` or `CVMTypeVarEntry`.

For instance, a closed, unresolved descriptor entry for a generic method call of the kind `Pair<String,Integer>.<String>chgSecond()` is a

```
struct CVMTypeDescriptor{
   CVMUint32 size;
   CVMUint8  kind;
   CVMUint32 hash;
};
```

Figure 4.15: The `CVMTypeDescriptor` data structure

`CVMMethodEntry` structure where: `receiver` is an index to the descriptor entry for `Pair<String,Integer>`; `method` points to the constant pool method entry for `Pair.chgSecond()`; `nparams` is `1` (as the `chgSecond()` defines one type-variable); and `params` contains an index pointing to the class descriptor entry for `String`. When this entry becomes resolved, the constant pool pointer in `method` is replaced with a pointer to the runtime descriptor for `Pair<String,Integer>.<String>chgSecond()`.

3.3 Runtime Descriptors

The runtime type information associated with generic objects and methods is encoded in specialised data-structures, called runtime descriptors. A runtime descriptor is used in several ways: to represent the exact, non-erased runtime type of a generic object; to perform type-dependent operations such as instance-tests and type-conversions; and, finally, to represent the exact type of a method in a generic method call.

All runtime descriptors have a common *header* (see Figure 4.15) that provides hints on the size (in bytes), the kind (e.g. class, method, array, etc.) and the hashcode of a given runtime descriptor. This information is used to efficiently store and retrieve runtime descriptors to and from the so called descriptor registry (see Section 3.4). In the remainder of this section we discuss two kinds of runtime descriptors, namely `CVMClassDescriptor` and `CVMMethodDescriptor` — though, for completeness, the GCVM must also support array descriptors and captured type-variable descriptors (see Section 4.3).

```
struct CVMClassDescriptor {
  CVMTypeDescriptor      header;
  CVMClassBlock*         class;
  CVMClassDescriptor*    super;
  CVMUint16              ninterfaces;
  CVMClassDescriptor*    interfaces[ninterfaces];
  CVMTypeDescriptor*     outer;
  CVMUint16              nparams;
  CVMTypeDescriptor*     params[nparams];
  CVMUint8               annotations[nparams];
};
```

Figure 4.16: The `CVMClassDescriptor` structure

3.3.1 Class Descriptors

Class descriptors are used to represent both generic types and wildcards of the kind `C<`$\overline{\texttt{T}}$`>`, such as `Pair<String,Integer>` and `Pair<?,? super Integer>`. The information associated with a runtime class descriptor for `C<`$\overline{\texttt{T}}$`>` — the erased type `C`, the descriptors for the type-parameters in $\overline{\texttt{T}}$, etc. — is stored in a structure called `CVMClassDescriptor`, made up of following fields (see Figure 4.16):

class the `CVMClassBlock` representing the erased type `C`;

super the class descriptor representing the generic supertype of `C<`$\overline{\texttt{T}}$`>`;

interfaces a descriptor array, containing the class descriptors for the generic interface types implemented by `C<`$\overline{\texttt{T}}$`>`;

outer the runtime descriptor representing the (possibly generic) enclosing type of `C<`$\overline{\texttt{T}}$`>`. Valid descriptors are either of kind `CVMClassDescriptor` or `CVMMethodDescriptor`;

params a descriptor array, containing the runtime descriptors for the type parameters in $\overline{\texttt{T}}$. Valid descriptors are either of kind `CVMClassDescriptor` or `CVMArrayDescriptor`;

annotations an array of integer values containing the variance annotations associated with the parameter types in $\overline{\texttt{T}}$; the encoding for the annotation values is identical to the one discussed in Section 2.1.1.

```
struct CVMMethodDescriptor {
  CVMTypeDescriptor        header;
  CVMMethodBlock*          method;
  CVMClassDescriptor*      receiver;
  CVMUint16                nparams;
  CVMTypeDescriptor*       params[nparams];
};
```

Figure 4.17: The `CVMMethodDescriptor` structure

For instance, the runtime type of the object `pair1` in Figure 4.11 (`Pair<String,Integer>`) is represented by a class descriptor where: `class` is the `CVMClassBlock` of `Pair`; `super` is the class descriptor for `Object`; the `interfaces` array is `null` — as `Pair` does not implement any generic interface; `outer` is also `null` — as `Pair` has no generic enclosing type; `params` is a two-element array containing the class descriptors for `String` and `Integer`, respectively; and `annotations` is a two-element array containing two 0 — as both type parameters `String` and `Integer` are *invariant*.

3.3.2 Method Descriptors

Method descriptors are used to represent the runtime type associated with a generic method call of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`, such as `Pair<String,Integer>.<String>chgSecond()`. The information associated with a runtime method descriptor for `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()` — the erased type of `m()`, the descriptors for the type-parameters in $\overline{\texttt{T}}$, the receiver type `C<`$\overline{\texttt{S}}$`>` — is stored in a structure called `CVMMethodDescriptor`, made up of the following fields (see Figure 4.17):

method the `CVMMethodBlock` representing the erased type of `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`;

receiver a pointer to the class descriptor for the (possibly generic) receiver type `C<`$\overline{\texttt{S}}$`>`;

params a descriptor array, containing the runtime descriptors for the type parameters in $\overline{\texttt{T}}$. Valid descriptors are either of kind `CVMClassDescriptor` or `CVMArrayDescriptor`.

```
struct CVMDescriptorRegistryEntry {
   CVMTypeDescriptor * value;
   struct CVMDescriptorRegistryEntry * nextEntry;
};

struct CVMDescriptorRegistry {
   CVMUint32 nbuckets;
   CVMUint32 nentries;
   CVMUint32 lowerRehashBound;
   CVMUint32 upperRehashBound;
   CVMfloat32  ratio;
   struct CVMDescriptorRegistryEntry** buckets;
};
```

Figure 4.18: The `CVMMethodDescriptor` structure

For instance, the runtime type associated with the generic method call `pair1.<String>chgSecond(three)` in Figure 4.9 is represented by a method descriptor where: `method` is the `CVMMethodBlock` for `Pair.chgSecond()`; `receiver` is the class descriptor for `Pair<String,Integer>`; and `params` contains the class descriptor for `String`.

3.4 The Descriptor Registry

There are several situations in which the same runtime descriptor is used more than once: the same generic opcode can be executed several times; different opcodes in the same class might refer to the same descriptor entry; again different opcodes in different classes might refer to *equivalent* descriptor table entries (though such entries are in different descriptor tables). The GCVM exploits an advanced caching technique in order to prevent unnecessary creation of runtime descriptors: when a runtime descriptor is first created, it is stored inside a specialised data-structure called *descriptor registry*; subsequent accesses to the same descriptor will simply fetch the existing descriptor from the registry — thus no time (and space) is wasted to create the same runtime descriptor multiple times. This approach has been introduced in the design of the EGO compiler [Vir05]. The descriptor registry is essentially an hashset-like data-structure (see Figure 4.18); each registry entry contains a runtime

descriptor and a pointer to the sibling entry. Descriptor registry entries are grouped in *buckets*; the descriptor hashcode is used to disperse entries uniformly among the buckets — this ensures constant-time lookups. The registry is also self-resizing: when a bucket contains too many (or too few) entries (this depends on `ratio`, `lowerRehashBound` and `upperRehashBound` fields), the bucket array is resized (and all the entries are reallocated). The process of creating a runtime descriptor consists in the following steps:

1. A *dummy* descriptor is created; such descriptor is only used to compute the hashcode of the registry entry that has to be searched;
2. The registry is searched for an entry whose content is similar to the entry computed at step 1;
3. If such an entry is found, the existing entry is returned — the creation of a new runtime descriptor is thus not required;
4. If no matching entry is found, a new runtime descriptor is created and then stored inside the registry. The creation of a new runtime descriptor may, in turn, trigger additional lookups: for instance, when a class descriptor has to be created, its parent descriptor — the descriptor for its supertype — has to be registered first.

The registry provides all the necessary operations for creating and registering all the kinds of runtime descriptors supported by the GCVM. For instance, runtime class descriptors of the kind `C<`$\overline{\texttt{S}}$`>.D<`$\overline{\texttt{T}}$`>` are created through the `CVMregistryAddClass` routine shown in Figure 4.19; this routine accepts the enclosing descriptor for `C<`$\overline{\texttt{S}}$`>`, the `CVMClassBlock` for `D`, an array of descriptors for all the type-parameters in $\overline{\texttt{T}}$ and returns the runtime descriptor associated with the generic runtime type `C<`$\overline{\texttt{S}}$`>.D<`$\overline{\texttt{T}}$`>`. For instance, in order to create the runtime class descriptor of the kind `Pair<String,Integer>`, the `CVMregistryAddClass` routine must be supplied the following arguments: a `null` value, as `Pair` is a toplevel class, the `CVMClassBlock` for `Pair`, and an array containing two class descriptors for `String` and `Integer`, respectively.

```
CVMTypeDescriptor* registry_add_class
  outer : CVMTypeDescriptor,
  cb : CVMClassBlock
  nparams : CVMUint16
  params : CVMTypeDescriptor*[nparams]
begin
fake_class_desc := a fake class descriptor
fake_class_desc.outer := outer
fake_class_desc.cb := cb
fake_class_desc.nparams := nparams
fake_class_desc.params := params
cached_desc := lookup(fake_class_desc)
if cahed_desc is null
  begin
  cahed_desc := create_class_desc(outer,cb,nparams,params)
  super_index := super_index_in_descriptor_table(cb)
  cahed_desc.super := resolve_entry(cb,super_index)
  for i := 0 to cb.ninterfaces
    begin
    interface_index := interface_index_in_descriptor_table(cb,i)
    cached_desc.interfaces[i] := resolve_entry(cb,interface_index)
    end
  end
return cahed_desc
end
```

Figure 4.19: Registering a class descriptor

The descriptor is then created following the procedure described below:

1. A *dummy* descriptor for `Pair<String,Integer>` is created; the only initialised fields of the dummy descriptors are the `CVMClassBlock` and the type-parameters array, as those fields are used to compute the descriptor's hashcode;
2. The registry is searched for an entry containing the descriptor for `Pair<String,Integer>`;
3. If a match is found, the previously stored descriptor is returned — this means that a class descriptor for `Pair<String,Integer>` has already been registered;

```
CVMTypeDescriptor* resolve_entry
  cb : CVMClassBlock,
  index : CVMUint16
begin
desc_table_entry := get_desc_table_entry(index)
if desc_table_entry.kind is CVMClassEntry
  begin
  class_entry := desc_table_entry
  if class_entry.state is resolved
    return class_entry.desc
  outer := resolve_entry(cb,class_entry.outer)
  class_cb := class_entry.cb)
  if class_entry.nparams is not 0
    begin
    params := new CVMTypeDescriptor*[class_entry.nparams]
    for i := 0 to class_entry.nparams
      params[i] := resolve_entry(cb,class_entry.params[i])
    end
  result := registry_add_class(outer, class_cb,class_entry.nparams,params)
  class_entry.desc := result
  return class_entry.desc
  end
...
end
```

Figure 4.20: Resolution of a `CVMClassEntry`

4. If no suitable registry entry is found, a new runtime descriptor for `Pair<String,Integer>` is created; then the registry performs a recursive lookup in order to retrieve the parent descriptor, namely the class descriptor for `Object`. Once the parent descriptor has been retrieved, the descriptor for `Pair<String,Integer>` is finally stored inside the registry.

3.5 Resolution of Descriptor Table Entries

When the GCVM executes an opcode that refers to a constant pool entry, such an entry must be resolved. The resolution of a constant pool entry consists in replacing the symbol stored in that entry with a pointer to some internal representation that is more suitable for execution. For instance, a

class constant pool entry is typically initialised with a class name; when the entry is resolved, the class name is replaced with the corresponding `CVMClassBlock` — this operation could involve class loading, in case the `CVMClassBlock` to be fetched is not available in the GCVM class table. Once a constant pool entry has been resolved, an opcode referring to that entry can be executed *atomically* with respect to class loading — the information needed during the execution is available in the resolved constant pool entry.

Analogously, when the GCVM executes an opcode that refers to a descriptor table entry, a similar resolution process must take place. The resolution of a descriptor table entry consists in creating (or fetching it from the registry, if one is already available) a runtime descriptor. The resolution process must take place before actual execution; in fact, the resolution of a descriptor table entry might trigger class loading (e.g. if a parameter type refers to a class that has not been loaded yet); or it can involve subtle operations, such as looking up the actual runtime types to be replaced for the type-variables in an open descriptor table entry (see Section 4.1.2); finally the resolution process might result in the allocation of a new data-structure, e.g. if no matching descriptor is found inside the descriptor registry — this can, in turn, trigger the resolution of other descriptor table entries.

A descriptor table entry starts off in the unresolved state; this means that no type-dependent instructions referring to that descriptor entry has been executed yet. An unresolved descriptor entry cannot be used as it is by the GCVM interpreter: in fact, it might contain — directly or indirectly — references to unresolved constant pool entries. When a type-dependent operation involving an unresolved descriptor entry is first executed, the resolution process ensures that all constant pool entries the entry refers to are resolved — this might involve class loading; then, a descriptor entry can be marked as resolved. Moreover, if the class descriptor entry to be resolved is closed, a runtime descriptor is retrieved and cached for later use.

The code in Figure 4.11 contains a generic instance creation expression that points to the third slot of `Pair`'s descriptor table — this is the descriptor table entry for the generic type `Pair<String,Integer>`. This entry must be resolved, so that the runtime descriptor for `Pair<String,Integer>` is avail-

able when executing the subsequent `opc_new` instruction — such descriptor represents the runtime type of the generic object that is to be allocated. To resolve the above descriptor entry, the following conditions must be satisfied:

- each constant pool entry referred (either directly or indirectly) by a descriptor table entry must be resolved — in this case, the constant pool entries for `Pair`, `String` and `Integer` and the corresponding `CVMClassBlock` are retrieved;
- each descriptor table entry referred (either directly or indirectly) by a descriptor table entry must also be resolved — in this case the descriptor table entries for `String` and `Integer` are resolved and the corresponding runtime descriptors are retrieved;
- the runtime descriptor for `Pair<String,Integer>` is created and then cached inside a field of the class descriptor entry. The following two cases are given:
  - The runtime descriptor for `Pair<String,Integer>` has not been registered yet; in this case a new runtime descriptor is created and then cached inside the descriptor entry to be resolved;
  - A runtime descriptor for `Pair<String,Integer>` is available in the registry, but such descriptor is not available in the descriptor entry cache (the `class` field, see Section 3.2.1) — this can happen if e.g. another class performed some type-dependent instruction involving the same generic type `Pair<String,Integer>`. In this case the runtime descriptor is fetched from the registry and then cached inside the descriptor entry to be resolved;
  - A runtime descriptor for `Pair<String,Integer>` is directly available in the descriptor entry cache — this can happen if e.g. the same opcode is being executed several times, or if another instruction involving the generic type `Pair<String,Integer>` has been executed in the same class. In this case the resolution process is simply skipped, as the entry is already in the resolved state.

```
struct CVMObject {
    volatile CVMObjectHeader hdr;
    volatile CVMUint32       fields[];
};

struct CVMObjectHeader {
  union {
    CVMClassBlock       *clas;
    CVMTypeDescriptor   *desc;
  } type;
  transient CVMUint32      various32;
};
```

Figure 4.21: The `CVMObject` data structure

3.6 The Object Layout

Any object in the GCVM has an *header* (`hdr`), which points to the object's runtime type, and a 32-bit values array, that is used to store the values for all the fields defined in the object's class (see Figure 4.21). An object header is a 64-bit data-structure called `CVMObjectHeader` that contains a pointer to the data-structure representing the runtime type of the object, as well as a 32-bit mask that is used for various purposes — e.g. to tag objects during garbage collection (GC), to lock objects awaiting for a monitor, etc.

The GCVM exploits two different layouts for encoding generic and non-generic instances. The header of a legacy, non-generic object always points to the object's class — there is a one-to-one correspondence between classes and non-generic types; as an example, the header of the object `one` in Figure 4.11 points to the `CVMClassBlock` for the class `String`. Dually, the header of a generic object always points to a *runtime descriptor* which, in turn, points back to the object's class; as an example, the header of the object `p1` in Figure 4.11 points to the runtime descriptor for the type `Pair<String,Integer>` which points back to the `CVMClassBlock` for the class `Pair`. This extra-level of indirection is crucial, as it ensures that all the class-related information, which might be required during execution, can be accessed uniformly on both

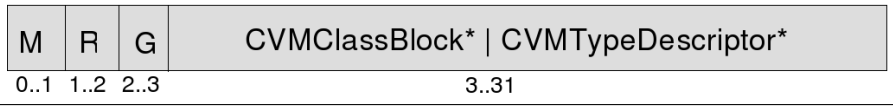


Figure 4.22: The `CVMObjectHeader`'s `type` bits

non-generic and generic instances. In order to implement transparent access to the `CVMClassBlock` of a given object, generic instances must be marked with a special *generic flag*. When such a flag is set, an additional step of indirection is required in order to retrieve the `CVMClassBlock` associated with the object's class. This flag could, in principle, be stored in one of the unused bits of the `various32` word. Unfortunately this technique is problematic, as the contents of this bit mask are flushed each time an object is moved across the heap during a GC round. Instead, we have chosen to store the generic flag directly inside the third lowest bit of the `type` pointer (see Figure 4.22). This is safe, as GCVM data-structures (including both `CVMClassBlock`s and runtime descriptors) are *byte aligned* — the 3 lowest bits are always set to 0. Moreover, this technique ensures that the generic flag is not touched during a GC round.

3.7 The GCVM Interpreter

Type-dependent opcodes can reference a descriptor table entry via the `DescriptorMap` attribute (see Section 2.2). However, the process of distinguishing between generic and non-generic opcodes by looking up into a method's `DescriptorMap` could be very inefficient — in most cases this lookup is likely to fail without retrieving any suitable entry — and lead to severe performance problems, especially if the code being executed contains several type-dependent opcodes. This problem could be partially addressed by resorting to a more efficient data-structure for storing descriptor map entries — so that a constant-time lookup is guaranteed; or we could cache the most frequently accessed `DescriptorMap` entries.

A more tempting alternative is to perform the lookup *once* and then explicitly mark an opcode with a flag indicating whether the opcode has an associated descriptor map entry, as shown in Figure 4.23b. Since the

```
public static void main(java.lang.String[]) {
   ...
   15:  opc_new → Pair
   ...
}
```

(a) A fragment of `Pair`'s bytecode

```
public static void main(java.lang.String[]) {
   ...
   15:  opc_new_generic → Pair<String,Integer>
   ...
}
```

(b) Full rewriting

```
public static void main(java.lang.String[]) {
   ...
   15:  opc_load_desc → Pair<String,Integer>
   17:  opc_new → Pair
   ...
}
```

(c) Partial rewriting

Figure 4.23: Bytecode rewriting

`opc_new` opcode has been replaced with the `opc_new_generic` opcode, no further lookup is required during execution. The drawback of this rewriting scheme is that it requires too many additional opcodes — one for each type-dependent opcode. In particular, we would need at least nine new opcodes (see section 1), but this is not possible, as there are only five unused slots[5].

Instead, we have chosen a different rewriting scheme, which consists in prepending an additional synthetic instruction, called `opc_load_desc`, to the original type-dependent opcode, as shown in Figure 4.23c. This approach requires just one additional opcode for *all* type-instructions opcodes that could refer to descriptor table entries. The `opc_load_desc` instruction has one operand, a 16-bit index pointing to the descriptor table entry that needs to be accessed during the execution of the subsequent type-dependent opcode.

Note that the bytecode in Figure 4.23c cannot be generated during com-

[5]In the GCVM opcodes are encoded using 8-bit strings.

```
curr_cb := current CVMClassBlock
curr_desc := current runtime descriptor
desc_idx := operand of the load_desc
switch curr_pc
...
case opc_load_desc
begin desc_table_entry := get_desc_table_entry(desc_idx)
if desc_table_entry is not resolved
  desc_table_entry.desc := resolve_entry(curr_cb,desc_idx)
curr_desc := desc_table_entry.desc
end
...
```

Figure 4.24: Executing the `opc_load_desc` opcode

pilation, as this would lead to backward compatibility issues; thus, the only possibility is to generate `opc_load_desc` instructions during class loading: first the contents of a `DescriptorMap` attribute is parsed in order to determine the set of type-dependent opcodes requiring instrumentation; such opcodes are then decorated with a corresponding `opc_load_desc` instruction.

A complete implementation of the bytecode rewriting strategy discussed in this section must address some subtle issues, such as e.g. to dynamically adjust the offset of a *branch*-like instructions. For the sake of brevity we do not discuss such details here — even though they are fully implemented in the GCVM.

The execution of a `load_desc` opcode triggers the resolution of the descriptor entry in the current class' descriptor table; the runtime descriptor retrieved during the resolution step is thus stored in a shared variable of the interpreter, namely `curr_desc`, as shown in Figure 4.24. In the remainder of this section we discuss two opcodes, namely `opc_new` and `opc_invokevirtual`; our aim is to show how type-dependent opcodes are executed by the GCVM interpreter.

3.7.1 Instance Creation Expressions

The interpreter routine for executing the `opc_new` opcode is reported in Figure 4.25; each time a new object has to be created, the interpreter must check

```
curr_desc := current runtime descriptor
curr_pc := current program counter value
cp_idx := operand of the opc_new
switch curr_pc
...
case opc_new
begin
cp_entry := get_cp_entry(cp_idx)
class_block := get_class_block(cp_entry)
new_object := allocate_new_object(class_block)
if curr_desc is not null
  begin
  new_object.header := curr_desc
  curr_desc := null
  end
end
...
```

Figure 4.25: Executing the `opc_new` opcode

whether some descriptor has been set in the `curr_desc` state variable. If no descriptor is found, the instance creation expression is executed the usual way — a new instance of type `C` is allocated, whose header points *directly* to the `CVMClassBlock` for `C`.

Conversely, if the `curr_desc` state variable contains a runtime class descriptor of the kind `C<`$\overline{\texttt{T}}$`>`, a new generic instance must be created. This is accomplished in three steps: first, a new non-generic instance is allocated; the header of the newly created instance is then set to the runtime descriptor in `curr_desc` — the class descriptor for `C<`$\overline{\texttt{T}}$`>`; finally, the state variable `curr_desc` is unset — this resets the state of the GCVM interpreter.

3.7.2 Method Calls

The interpreter routine for executing the `opc_invokevirtual` opcode is reported in Figure 4.25; again, the interpreter must check whether some descriptor has been set in the `curr_desc` state variable. If no descriptor is found, the method call is executed the usual way — a new method frame [LY99] is allocated on top of the interpreter stack and the program counter value is updated so that the interpreter will *jump* at the first instruction of the

```
curr_desc := current runtime descriptor
curr_pc := current program counter value
curr_frame := current frame being executed
curr_frame.desc := current method descriptor
curr_frame.method := current CVMMethodBlock
curr_frame.class := current CVMClassBlock
receiver_obj := the receiver of this method call
meth_name_idx := operand of the opc_invokevirtual
switch curr_pc
...
case opc_invokevirtual
begin
meth_name_entry := get_cp_entry(meth_name_idx)
method_block := get_method_block(meth_name_entry)
curr_frame.class := obj_get_class(receiver_obj)
curr_frame.method := method_block
if curr_desc is not null
  begin
  curr_frame.desc := curr_desc
  curr_desc := null
  end
end
...
```

Figure 4.26: Executing the `opc_invokevirtual` opcode

method being called.

Conversely, if the `curr_desc` state variable contains a runtime method descriptor of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`, a generic method call must be executed. This is accomplished in three steps: first, a new method frame is allocated on top of the interpreter stack; the value of the `curr_desc` state variable is then saved in the `desc` field of the new method frame — this is required in order to keep track of the instantiation environment associated with the generic method call; finally, the state variable `curr_desc` is unset and normal execution is resumed.

4 Advanced Features

A complete reification approach should take into account subtle issues, such as efficient management of open descriptors, dynamic dispatching of generic

method calls and runtime support for wildcard types. First, suppose we want to add a method in `Pair` that returns a new pair where the original elements are reversed:

```
class Pair<X,Y> {
   ...
   Pair<Y,X> swap()  new Pair<Y,X>(x, y);
   ...
}
```

The method `swap()` contains a type-dependent operation, namely an instance creation expression of the kind `new Pair<Y,X>()`. Note that the type of the instance creation expression is expressed in terms of the type-variables defined in `Pair`. Consequently, the runtime type of the object to be created depends on the actual instantiation of the type-variables `X` and `Y`, respectively — this is different from e.g. an instance creation expression of the kind `new Pair<String,Integer>`, where the runtime type of the object can always be resolved statically. For instance, a method call of the kind `Pair<Double,Integer>.swap()` yields a result of type `Pair<Double,Integer>`.

The reification support must also provide support for dynamic method dispatching; we discussed how the interpreter handles non-virtual generic method calls — for the sake of simplicity, the method `chgSecond()` in Figure 4.8a has been deliberately marked as `final`. In such cases the receiver is said to be *monomorphic*, that is, its type is always determined statically; this is possible since a `final` method defined in `C<`$\overline{\texttt{X}}$`>` cannot be overridden by subclasses of `C<`$\overline{\texttt{X}}$`>`.

Conversely, when a generic method call involves a non-final method, such as `chgFirst()` in Figure 4.8a, the receiver type is not known until execution — that is, the call-site is said to be *polymorphic*:

```
Pair<String,Integer> psi = ...
psi.<Float>chgFirst(1.0f);
```

Here, the runtime type of `psi` could be *any* subtype of `Pair<String,Integer>`. Thus, the execution of a virtual generic method call is inherently more complex, as the actual receiver type must

be resolved *during* execution; the GCVM greatly reduces the overhead associated with this dynamic resolution process, by leveraging a specialised data-structure called *Virtual Parametric Method Table* (VPMT). This structure is used to minimise the amount of runtime descriptors that need to be registered in order to handle virtual generic method calls — this is accomplished by introducing an highly sophisticated caching technique.

Wildcards introduces many subtleties in the Java programming language (see Section 2), such as capture conversion, type-containment and captured calls — these issues must addressed in order to provide a full-fledged reification support. For instance, the GCVM interpreter must be able to *capture* a runtime descriptor of the kind `C<`$\overline{\texttt{W}}$`>`, where one or more type parameters in $\overline{\texttt{W}}$ is a wildcard — this is required e.g. for executing subtyping tests between wildcard types, or for computing the parent descriptor for `C<`$\overline{\texttt{W}}$`>`. The GCVM also supports captured calls — a generic method call where one ore more method type-variables are replaced with captured type-variables; during the execution of a captured call, the interpreter must be able to dynamically introspect the runtime types of the actual arguments supplied to a generic method, in order to discover the "real" runtime types associated with the method type-variables.

4.1 Open Descriptor Entries

An open descriptor entry is used to represent a type containing one or more type-variables, such as `List<Y>`, `Pair<String,Z>.<Y>chgSecond()`. There is a fundamental difference between open and closed entries: the runtime type associated with a closed entry does not depend on the runtime type of the class/method in which the entry is used. This allows for a compact and efficient implementation, as the runtime descriptor associated with a closed entry can be *cached* inside the descriptor table, so that subsequent access are immediate (see Section 3.5). Conversely, open descriptor entries must undergo a heavy-weight resolution process, where each type-variable `X` in the entry is replaced with an actual type `T` obtained from the runtime descriptor of the enclosing class/method.

The diagram in Figure 4.27 illustrates how open descriptor entries are

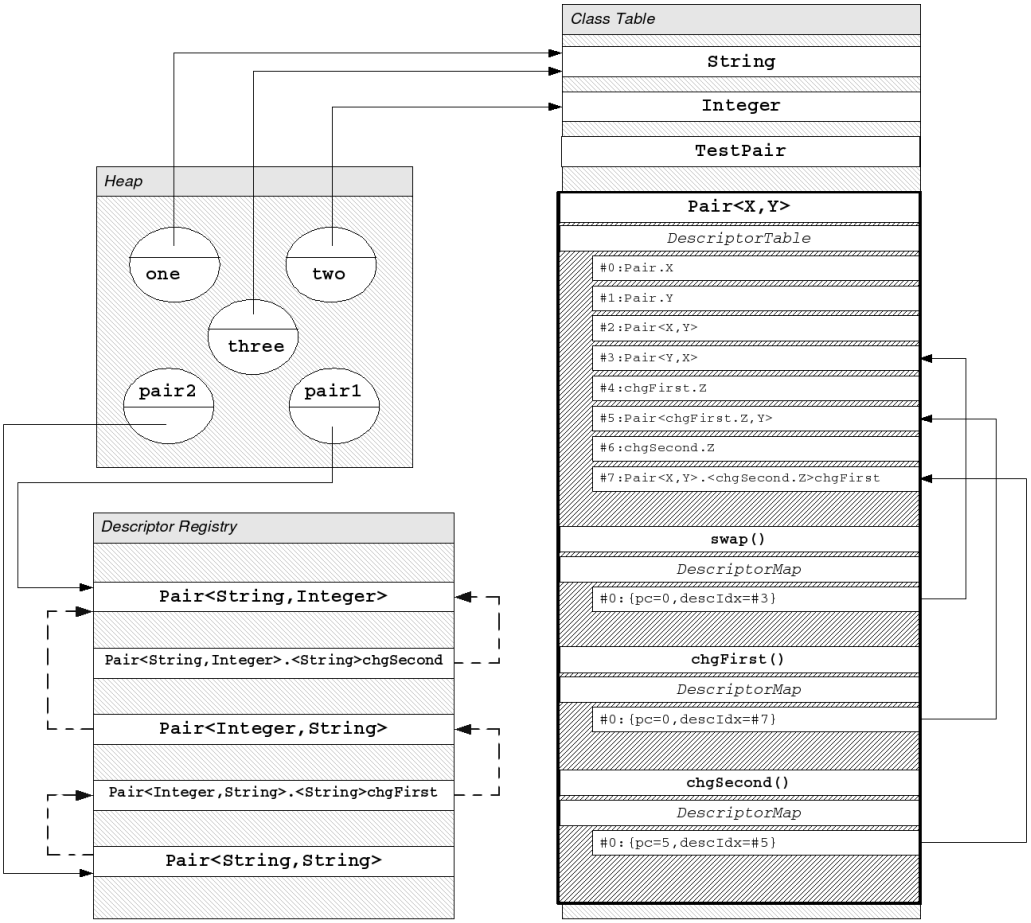


Figure 4.27: GCVM and open descriptors. Dotted arrows express dependencies between runtime descriptors

handled by the GCVM; the `swap()` method in `Pair` refers to the open descriptor entry for the generic type `Pair<Y,X>`, where each type-variable type is represented by a special descriptor table entry — a type-variable entry. In order to resolve a type-variable entry, the GCVM interpreter must gather the actual types `U` and `V` associated with the type-variables `X` and `Y`, respectively; such types are then used to form a new runtime descriptor of the kind `[U/X,V/Y]Pair<Y,X>`. For instance, given a method call of the kind `Pair<String,Long>.swap()`, the resolution process yields a runtime descriptor of the kind `[String/X,Long/Y]Pair<Y,X>`, namely `Pair<Long,String>`;

similarly, given a method call of the kind `Pair<Float,Float>.swap()`, the resolution process yields a different runtime descriptor of the kind `[Float/X,Float/Y]Pair<Y,X>`, namely `Pair<Float,Float>`. In principle, each call to `swap()` could lead to a new runtime descriptor, as the number of instantiation of a generic type of the kind $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ is not bounded.

It is clear that a caching technique such as the one discussed for closed entries cannot be applied here, as it would fail to provide the correct result if e.g. the same open descriptor entry is resolved twice with two different instantiations for `X` and `Y`. Instead, access to open descriptor entries can be optimised by exploiting the runtime *dependencies* between runtime descriptors: the key idea is that a descriptor of the kind `Pair<S,T>`, for any `S` and `T`, has a sibling descriptor of the kind `Pair<T,S>` where the type-parameters are reversed — we call such a descriptor a *friend* descriptor.

In Figure 4.27, friend descriptors are linked together by dotted arrows; for instance, the runtime descriptor for `Pair<String,Integer>` (the runtime type of `p1`) has two friend descriptors — namely a method descriptor of the kind `Pair<String,Integer>.<String>chgSecond()` and a class descriptor of the kind `Pair<Integer,String>`, respectively. In the following we discuss how these runtime dependencies can be leveraged in order to provide a sophisticated and efficient caching scheme.

4.1.1 `CVMTypeVarEntry` and `CVMTypeVarBlock`

An additional kind of entry, namely `CVMTypeVarEntry`, is used to represent both class and method type-variables. For instance, the open descriptor entry for `Pair<X,Y>` has a type-parameters array containing two indices, pointing to the type-variable entries for `Pair#X` and `Pair#Y`, respectively. A type-variable entry always points to a `CVMTypeVarBlock`, a data-structure used to represent type-variable declarations of the kind `T#X`; hence, type-variable entries are not resolved in a standard fashion. The `CVMTypeVarBlock` for a type-variable of the kind `C#X` is created when $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ is first loaded; at this stage, the `CVMClassBlock` of $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ is also augmented with an array of `CVMTypeVarBlock` (one for each type-variable in $\texttt{C<}\overline{\texttt{X}}\texttt{>}$).

```
struct CVMTypeVarEntry{
  CVMDescriptorTableEntryHeader header;
  CVMTypeVarBlock*  tb;
};

struct CVMTypeVarBlock{
  CVMUint8* slot;
  union {
     CVMClassBlock class_owner;
     CVMMethodBlock method_owner;
  } owner;
  CVMUint16    class_bound_idx;
  CVMUint16    ninterface_bounds;
  CVMUint16*   interface_bounds[ninterface_bounds];
};
```

Figure 4.28: The `CVMTypeVarEntry` and `CVMTypeVarBlock` structures

The `CVMTypeVarBlock` structure is made up of the following fields:

slot a numeric value encoding the position in which the type-variable occurs in a generic class/method declaration;

owner the internal data-structure representing the type-variable's owner, either a `CVMClassBlock` — for class type-variables of the kind `C#X` — or a `CVMMethodBlock` — for method type-variables of the kind `m()#X`;

class_bound_idx an index to the class descriptor entry for the type-variable's class bound;

interface_bounds an array of size is `ninterface_bounds`, where the i-th element is an index to the class descriptor entry for the i-th interface bound associated with the type-variable (if any).

For example, a type-variable of the kind `Pair#Y` is represented by a `CVMTypeVarBlock` where: `slot` is set to `1`, since `Pair#Y` is the second type-variable declared by `Pair`; `owner` is a pointer to `Pair`'s `CVMClassBlock`; `class_bound_idx` is set to `-1`, as `Pair#Y` has no class bound; analogously, `interface_bounds` is an empty array, given that `Pair#Y` has no interface bounds.

```
CVMTypeDescriptor* resolve_entry
  cb : CVMClassBlock,
  index : CVMUint16,
  bounding_desc : CVMTypeDescriptor
begin
desc_table_entry := get_desc_table_entry(cb,index)
if desc_table_entry.kind is CVMTypeVarEntry and
    bounding_desc.tag is CVMClassDescriptor
  begin
  tvar_entry := desc_table_entry
  tb_to_find := tvar_entry.tb
  bounding_cb := bounding_desc.class
  if bounding_cb == tb_to_find.owner
    return bounding_desc.params[tb_to_find.slot]
  else
    return resolve_entry(cb,index,bounding_desc.outer)
  end
...
end
```

Figure 4.29: Resolution of a `CVMTypeVarEntry`

4.1.2 Resolution of Type-variable Entries

The resolution process of a type-variable entry (shown in Figure 4.29) typically consists in finding the actual runtime descriptor associated with a given type-variable of the kind `T#X`. This is accomplished by looking at the so called *bounding descriptor* — the runtime descriptor used to keep track of the current instantiation context. A bounding descriptor can be either a class descriptor (if the interpreter is executing a non-generic method) or a method descriptor (if the interpreter is executing a generic method call). In the general case, the instantiation context can be *nested*, as both class and method descriptors might optionally define an enclosing descriptor. The resolution of a type-variable entry of the kind `C#X` consists in the following steps (in the remainder of this section we discuss the case where the bounding descriptor is class descriptor):

1. First, the `CVMClassBlock` of the bounding descriptor is retrieved — the bounding descriptor is stored in a state variable of the interpreter;

2. If the retrieved `CVMClassBlock` is the owner of the type-variable entry to be resolved, the resolution process yields the runtime descriptor associated with the actual type of `C#X` in the bounding descriptor's type parameters array;
3. Otherwise, the type-variable owner is some enclosing descriptor of the current bounding descriptor; in this case, another resolution process is triggered recursively, where the bounding descriptor is replaced by its enclosing descriptor.

Consider the open descriptor entry for `Pair<Y,X>` referred to by the `Pair.swap()` method; this entry refers to two type-variable entries, for `Pair#Y` and `Pair#X`, respectively. Such entries must be resolved — that is, the actual types for for `Pair#Y` and `Pair#X` must be determined — so that a runtime descriptor for `Pair<Y,X>` can be registered. In order to resolve the above type-variable entries, the resolution process must be supplied a valid bounding descriptor; since the interpreter is executing a non-static, non-generic method of the kind $\texttt{C<}\overline{\texttt{S}}\texttt{>.m()}$, the runtime descriptor for the actual receiver type $\texttt{C<}\overline{\texttt{S}}\texttt{>}$ is assumed to be the current bounding descriptor. For instance, given a method call of the kind if `Pair<String,Integer>.swap()`, the bounding descriptor is the runtime descriptor for `Pair<String,Integer>`.

The bounding descriptor supplied to the resolution routine is thus used for retrieving the runtime descriptors corresponding to the actual types associated with `Pair#Y` and `Pair#X`, respectively; in the former case, we have that the owner of `Pair#Y` is the `CVMClassBlock` for `Pair` — the erased type of the current bounding descriptor `Pair<String,Integer>`. Therefore, the actual runtime descriptor for `Pair#Y` is retrieved by accessing the second slot of the bounding descriptor's type parameters array — this yields the runtime descriptor for `Integer`. The remaining type-variable entry for `Pair#X` is resolved in a similar way — this time the resolution process yields the runtime descriptor for `String`. Once both type-variable entries have been resolved, the open descriptor entry for `Pair<Y,X>` can be resolved following the standard procedure described in Section 3.5; this yields a new runtime descriptor for `Pair<Integer,String>`.

4.1.3 Open Descriptors and Caching

The resolution process of open descriptor entries poses several performance issues: the type-variable entries associated with a given open descriptor entry must undergo an heavy-weight discovery process that amounts at inferring one or more runtime descriptors from a given instantiation context — the bounding descriptor supplied to the resolution routine. Worse, the same open descriptor entry could be resolved several times against different bounding descriptors, so that each time a new runtime descriptor is returned; therefore, a caching technique similar to the one exploited for closed descriptors (see Section 3.5) would fail to provide the correct result.

Note that if an open descriptor entry is resolved several times against the same bounding descriptor, the resolution routine described in Figure 4.29 always yields the same runtime descriptor. This suggests the idea that a runtime descriptor associated with an open descriptor entry can be cached in the bounding descriptor used during the resolution process.

Each runtime descriptor is equipped with an array of *friend* descriptors, used to cache the *dependant* runtime descriptors. For instance, the size of the friends array of a class descriptor of the kind `C<`$\overline{\texttt{T}}$`>`, where `C` is a generic class of the kind `C<`$\overline{\texttt{X}}$`>`, is given by the number of the open descriptor entries in `C`'s descriptor table referring to type-variable entries of the kind `C#`$\overline{\texttt{X}}$. Similarly, the size of the friend array of a method descriptor of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`, where `m()` is a generic method of the kind `<`$\overline{\texttt{X}}$`>m()`, is given by the number of the open descriptor entries in `C`'s descriptor table referring to type-variable entries of the kind `m()#`$\overline{\texttt{X}}$. Each open descriptor entry is given a *unique* index, used to determine the position of a resolved runtime descriptor inside the bounding descriptor's friends array.

The complete resolution routine for open class descriptor entries is shown in Figure 4.30; when an open descriptor entry of the kind `C<`$\overline{\texttt{T}}$`>` must be resolved, the friends array of the bounding descriptor is first accessed; if a cached runtime descriptor is found, that descriptor is immediately returned and the resolution process terminates. If no such descriptor is found, the resolution process takes the slow route — in this case a new descriptor is

```
CVMTypeDescriptor* resolve_entry
  cb : CVMClassBlock,
  index : CVMUint16
  bounding_desc : CVMTypeDescriptor
begin
desc_table_entry := get_desc_table_entry(cb,index)
if desc_table_entry.kind is CVMClassEntry
  begin
  class_entry := desc_table_entry
  if class_entry.state is resolved
    if class_entry is open
      cached_desc := bounding_desc.friends[class_entry.index]
    else
      cached_desc := class_entry.desc
  if cached_desc is not null
    return class_entry.desc
  outer := resolve_entry(cb,class_entry.outer)
  class_cb := class_entry.cb
  if class_entry.nparams is not 0
    begin
    params := new CVMTypeDescriptor*[class_entry.nparams]
    for i := 0 to class_entry.nparams
      params[i] := resolve_entry(cb,class_entry.params[i])
    end
  result := registry_add_class(outer, class_cb,class_entry.nparams,params)
  if class_entry is open
    bounding_desc.friends[class_entry.index] := result
  else
    class_entry.desc := result
  return result
  end
...
end
```

Figure 4.30: Resolution of a `CVMClassEntry` revised

```
class Triple<X,Y,Z> extends Pair<Y,Z> {
   X x;
}
...
Pair<String,Integer> tsif = new Triple<Float,String,Integer>
Pair<Integer,String> tsfi = tsif.swap();
```

Figure 4.31: Open descriptor and subtyping

retrieved and then stored in the `idx`th slot of the friends array of the bounding descriptor, where `idx` is the unique index associated with the open descriptor entry to be resolved.

For instance, the descriptor table of the generic class `Pair` (see Figure 4.27) has four open descriptor entries in position 2, 3, 5 and 7, respectively. The entries stored at position 2 and 3 are bounded by the generic class type `Pair<X,Y>`; consequently, the friends array of a class descriptor of the kind `Pair<S,T>` contains 2 elements. When an open descriptor entry of the kind `Pair<Y,X>` is first resolved, a new descriptor must be retrieved — this is accomplished by following the steps reported in Figure 4.29; assuming that the bounding descriptor is a class descriptor of the kind `Pair<String,Integer>`, the resolution process yields a new descriptor of the kind `Pair<Integer,String>`. This descriptor is stored in in the second slot of the bounding descriptor's friends array — the open descriptor entry for `Pair<Y,X>` is given the unique index 2. Hence, assuming that the open descriptor entry for `Pair<Y,X>` is resolved multiple times against the same bounding descriptor, namely `Pair<String,Integer>`, the resolution process simply yields the previously cached descriptor.

4.1.4 Open Descriptors and Subtyping

Subtle issues arise when open descriptors are used in conjunction with subtyping; suppose we define a subclass of `Pair`, namely `Triple<X,Y,Z>`, as shown in Figure 4.31. The generic class `Triple<X,Y,Z>` has a generic supertype, namely `Pair<Y,Z>`; hence, the `extends` clause implicitly defines a mapping between the type-variables in `Triple` — `Triple#Y` and `Triple#Z` — and the

type-variables in `Pair` — `Pair#X` and `Pair#Y`, respectively. For instance, the supertype of `Triple<Float,String,Integer>` is `Pair<String,Integer>`; such type is obtained by replacing the actual types of `Triple#Y` and `Triple#Z` for the type-variables `Pair#X` and `Pair#Y` in the generic type `Pair<X,Y>`.

In a method call of the kind `Triple<Float,String,Integer>.swap()`, an open descriptor entry of the kind `Pair<Y,X>` must be resolved, where the bounding descriptor is the runtime descriptor for `Triple<Float,String,Integer>`. In other words, we are trying to resolve an open descriptor entry of the kind $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ against a bounding descriptor of the kind $\texttt{D<}\overline{\texttt{S}}\texttt{>}$, where `D <: C`. It is thus necessary to apply an additional resolution step, that essentially amounts at *lifting* the bounding descriptor to the same *depth* of the descriptor entry to be resolved. That is, the resolution process must recursively access the bounding descriptor's parent in order to retrieve a runtime descriptor whose base class is `C` — where `C` is also the owner of the the descriptor table defining the open descriptor entry to be resolved. In our example, the bounding descriptor `Triple<Float,String,Integer>` is lifted to the runtime descriptor for `Pair<String,Integer>`, as we are resolving an open descriptor entry in `Pair`'s descriptor table; the resolution process goes as usual, where the lifted descriptor is used as the new bounding descriptor.

The process of lifting a runtime descriptor can involve multiple steps; in order to address this problem efficiently, the `CVMClassBlock` for a generic class of the kind $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ is decorated with a *depth* field; this field univocally determines the the position of $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ in the inheritance hierarchy. For instance, the depth of `Pair<X,Y>` is simply 0, as `Pair<X,Y>` has no generic supertype; on the other hand, the depth of `Triple<X,Y,Z>` is 1, as `Triple<X,Y,Z>` has a generic supertype, namely `Pair<Y,Z>`.

Moreover, a runtime class descriptor of the kind $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ must provide an array in which all supertype descriptors are stored, ordered by ascending *depth* values; the first element of the supertypes array is always the root of a given generic hierarchy, while the last element of the supertypes array is the bottom of the subtyping hierarchy — i.e. the current descriptor. For example, the supertypes array of a runtime descriptor of the kind `Triple<Float,String,Integer>` stores three elements: *(i)* the top descriptor

```
CVMTypeDescriptor* check_bounding_desc
  cb : CVMClassBlock,
  bounding_desc : CVMTypeDescriptor
begin
expected_depth := cb.depth
actual_depth := bounding_desc.class.depth
if actual_depth is greater than expected_depth
  return bounding_desc.supertypes[expected_depth]
else
  return bounding_desc
end
```

Figure 4.32: Choosing the right `bounding_desc`

for `Object`, *(ii)* the runtime descriptor for `Pair<String,Integer>`, and *(iii)* the runtime descriptor for `Triple<Float,String,Integer>` itself, orderly.

Hence, the problem of choosing the right bounding descriptor is efficiently addressed by checking the depth of the actual bounding descriptor supplied to the resolution routine against the *expected* depth — the depth of descriptor table defining the open descriptor entry to be resolved. If a mismatch is found, the bounding descriptor is lifted to a parent descriptor that matches the expected depth. For example, when the open descriptor entry for `Pair<Y,X>` is resolved against a bounding descriptor of the kind `Triple<Float,String,Integer>` the resolution process needs to adjust the bounding descriptor, as we are attempting to resolve an entry in a descriptor table whose depth is 1 (`Pair`) against a bounding descriptor with depth 2 (`Triple`). More specifically, the bounding descriptor must be replaced by its parent descriptor, as shown in Figure 4.32; this yields a new bounding descriptor with the correct depth, namely `Pair<String,Integer>`. The resolution process is then resumed, and the class descriptor for `Pair<Integer,String>` is retrieved the usual way — either by creating a new descriptor, or by returning a previously cached descriptor in the bounding descriptor's friends array.

```
class Pair<X,Y> {
   void <Z> chgFirst(Z z){ ... }
   ...
}
class Triple<X,Y,Z> extends Pair<Y,Z> {
   void <V> chgFirst(V v){ ... }
   ...
}
...
boolean b = ...
Pair<String,Integer> psi = b ?
   new Pair<String,Integer>(...) :
   new Triple<Float,String,Integer>(...);
psi.<String>chgFirst("1");
```

Figure 4.33: The problem of dynamic dispatching in generic method calls

4.2 Dynamic Dispatching and Generic Methods

The Java programming language, as most Object-Oriented languages, supports dynamic method dispatching, that is, the ability of dynamically binding a method call to the most specific implementation available for that method, based on the runtime type of the receiver. Dynamic dispatching is typically handled efficiently by exploiting a specialised a data-structure called *Virtual Method Table* (VMT). A VMT is a list of pointers to the methods defined in a given class hierarchy. The key property of VMT is that objects belonging to the same inheritance hierarchy have VMTs with a similar layout: the pointer to a method defined in a class `A` is stored in the same VMT slot across all subclasses of `A`. VMTs are typically stored inside class-related data-structures (as `CVMClassBlock`, in the case of the GCVM), so that each object implicitly keeps a reference to the VMT associated with its runtime type. When the interpreter executes a virtual method call, the receiver's VMT is accessed using the *static* index of the method being called; this automatically yields the most specific implementation available for that method.

Combining together dynamic dispatching and generic method calls can be quite problematic performance-wise: this leads to pathological cases in

```
struct CVMVpmtEntry{
   CVMUint16               max_capacity;
   CVMUint16               size;
   CVMMethodDescriptor*    descs[size];
} CVMVpmt[];
```

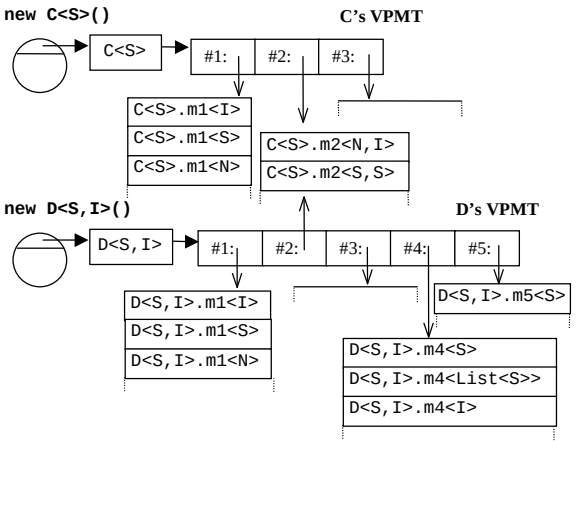
Figure 4.34: The `CVMVpmt` data structure

which the runtime descriptor associated with a given method call cannot be determined until execution [Vir05]. In the example in Figure 4.33 the subclass `Triple<X,Y,Z>` redefines a method in the superclass, namely `chgFirst()`. Note that the runtime type of `psi` is either `Pair<String,Integer>` or `Triple<String,Integer,Float>`, depending on the value assumed by `b`. Consequently, the method descriptor exploited in the generic method call to `chgFirst()` is either `Pair<String,Integer>.<String>chgFirst()` or `Triple<String,Integer,Float>.<String>chgFirst()`.

4.2.1 Virtual Parametric Method Tables

The problem of dynamic dispatching in generic method calls can be addressed by means of a specialised data-structure called *Virtual Parametric Methods Table* (VPMT)[Vir03a]. This structure is similar to a VMT, but instead of retrieving method implementions, VPMTs are used to retrieve runtime method descriptors. The VPMT features a correspondence property similar to the one discussed for VMT: the position of a method descriptor of the kind $\texttt{C<}\overline{\texttt{S}}\texttt{>.<}\overline{\texttt{T}}\texttt{>m()}$ in a given VPMT is *independent* from the runtime descriptor associated with the receiver type $\texttt{C<}\overline{\texttt{S}}\texttt{>}$. Therefore, this positional information can be used by the interpreter in order to efficiently retrieve the runtime descriptor associated with a virtual generic method call.

A VPMT is essentially an array of entries used to store runtime method descriptors (see Figure 4.34). The *width* of the VPMT is *fixed*: each generic method in a given class is assigned a different VPMT entry; the *height* of a VPMT entry can vary: each time a generic method is called with a new instantiation context, a new method descriptor is added to the corresponding VPMT entry. More specifically, the VPMT entry for a generic method of the




```
class C<X>{
  public <Z> void m1(){..}
  public <Z,W> void m2(){..}
  protected <Z> void m3(){..}
  private <Z> void m4(){..}
}
...
class D<X,Y> extends C<X>{
  public <Z> void m1(){..}
  protected <Z> void m3(){..}
  public <Z,W> void m4(){..}
  public <Z> void m5(){..}
}
...
Object o1=new C<String>();
o1.<Integer>m1();
o1.<String>m1();
...
Object o2=
new D<String,Integer>();
o2.<Integer>m1();
o2.<String>m5();
...
```

Figure 4.35: VPMTs in action

kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{X}}$`>m()` stores method descriptors of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`, where each method descriptor corresponds to a new instantiation of the generic method `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{X}}$`>m()`.

Different overriding versions of the same generic method are associated with the same VPMT slot (see Figure 4.35). For instance, the class descriptors for both `C<String>` and `D<String,Integer>` points to a VPMT whose first entry stores method descriptors of the kind `<`$\overline{\texttt{X}}$`>m1()`. Such entries are said to be *correspondent*. A crucial property of correspondent VPMT entries is that they share the same *height*: method descriptors of the same kind are stored at the same height across all correspondent VPMT entries. For instance, the second slot of the VPMT entry associated with `m1()` in `C<String>` is a method descriptor of the kind `C<String>.<String>m1`; similarly, the second slot of the VPMT entry associated with `m1()` in `D<String>` is a method descriptor of the kind `D<String>.<String>m1` — hence, the *height* of a method descriptor in a given VPMT entry is independent from the actual receiver type.

```
CVMMethodDescriptor* get_virtual_method_desc
  mb: CVMMethodBlock
  params: CVMTypeDescriptor[]
  top_desc: CVMClassDescriptor
  curr_rec: CVMClassDescriptor
  vpmt_idx: CVMUint16
  top_vpmt: CVMVpmt
  curr_ vpmt: CVMVpmt
begin
height := lookup(top_vpmt[vpmt_idx],params)
if height is valid
  desc := top_vpmt[vpmt_idx][height]
else
  desc := registry_add_method(mb, top_desc, params)
height := append(top_vpmt[vpmt_idx], desc)
if mb is not overriden
  curr_vpmt[vpmt idx][height] := desc
else
  begin
  ov_rec := find_overriding_desc(mb, curr_rec)
  ov_vpmt := ov_rec.vpmt
  desc := r_vpmt[vpmt idx][height]
  if desc is not found
    begin
    desc := registry_add_method(mb, ov_rec, params)
    r_vpmt[vpmt idx][height] := desc
    end
  if ov_rec is not curr_rec
    curr_vpmt[vpmt_idx][height] := desc
  end
return desc
end
```

Figure 4.36: Preserving VPMT's consistency

4.2.2 Consistency of VPMTs and Caching

The correspondence property of VPMTs must be satisfied through incoming registrations of new class descriptors and method descriptors. Suppose that we want to register a runtime method descriptor of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`; we call *top receiver* the runtime descriptor associated with the supertype of `C<`$\overline{\texttt{S}}$`>` in which the generic method is first defined. First, the VPMT entry associated with the generic method `<`$\overline{\texttt{X}}$`>m()` in the top descriptor's VPMT is accessed; the resolution routine then iteratively scans this VPMT entry to find a suitable method descriptor of the kind `<`$\overline{\texttt{T}}$`>m()`. If a matching descriptor is found, its height is cached inside the descriptor method entry (see the `pos` field discussed in Section 3.2.2), so that it can be re-used during subsequent generic method calls; if no match is found, a new method descriptor is registered and stored inside the top descriptor's VPMT entry. Note that the receiver's VPMT must be updated accordingly: if the class associated with the runtime type of the receiver does not override `<`$\overline{\texttt{X}}$`>m()`, the entry in the receiver descriptor's VPMT is filled with a pointer to a method descriptor in the top descriptor's VPMT. Conversely, if the receiver class overrides `<`$\overline{\texttt{X}}$`>m()`, a new method descriptor with a different receiver type must be registered and then stored in the receiver descriptor's VPMT entry.

Hence, in order to efficiently retrieve the runtime descriptor associated with a virtual generic method call of the kind `C<`$\overline{\texttt{S}}$`>.<`$\overline{\texttt{T}}$`>m()`, the VPMT of the receiver descriptor — a class descriptor of the kind `C<`$\overline{\texttt{S}}$`>` — must be accessed with an *index* to the VPMT entry associated with the generic method `<`$\overline{\texttt{X}}$`>m()`; such entry must then be accessed at the correct height, so that a method descriptor of the kind `<`$\overline{\texttt{T}}$`>m()` is retrieved — this descriptor corresponds to a suitable instantiation of the generic method `<`$\overline{\texttt{X}}$`>m()`.

If both the index and the height are available during execution, the runtime method descriptor associated with a virtual generic method call is retrieved as follows:

```
rec_desc.vpmt[vpmt_index].descs[height]
```

Therefore, it is crucial that both the index and the height associated with a given virtual generic method call are made available as soon as possible: this

ensures that the resolution routine for retrieving a suitable method descriptor is executed only once — namely, the first time a generic method of the kind `<`$\overline{\texttt{T}}$`>m()` is called on a top receiver of the kind `C<`$\overline{\texttt{S}}$`>`. Note that, in the general case where the method descriptor entry associated with a virtual generic method call is open, the height value cannot be cached inside the descriptor entry; instead, such value is cached into a slot of the bounding descriptor's friends array (see Section 4.1.2).

Finally, there are cases in which virtual generic method calls cannot be optimised using the technique discussed in this section: for instance, no caching is possible when the receiver is either an interface type or a wildcard type. In such cases, the top descriptor associated with a virtual generic method call cannot be safely determined — e.g. because interfaces support multiple inheritance — and, consequently, the correspondence property of VPMT cannot be guaranteed; this leads to severe performance issues, as discussed in Section 5.

4.3 Capture Conversion

Capture conversion (see Section 2.2), is a process that takes a generic type of the kind `C<`$\overline{\texttt{S}}$`>` — possibly containing one or more wildcard type-arguments — and turns it into a generic type `C<`$\overline{\texttt{T}}$`>`, where all occurrences of wildcard type-arguments have been replaced by fresh type-variables. Despite capture conversion is mainly a static mechanism that extends the applicability of e.g. membership checks ad method type-inference to wildcard types, the reification support must be equipped with a routine for capturing class descriptors; such a routine must be exploited when registering a runtime class descriptor of the kind `C<`$\overline{\texttt{S}}$`>`, where one or more variance annotations associated with the descriptors in $\overline{\texttt{S}}$ denote a wildcard type-argument. In such case, the registration routine must apply capture conversion to the descriptor for `C<`$\overline{\texttt{S}}$`>` — this yields another descriptor of the kind `C<`$\overline{\texttt{T}}$`>`, where the variance annotations associated to the descriptors in $\overline{\texttt{T}}$ are all set to 0 — recalling from Section 2.1.1, the variance annotation 0 is used to denote *invariant* type-arguments. Moreover all descriptors in $\overline{\texttt{S}}$ associated with a variance annotation other than 0 are replaced by *fresh type-variable* descriptors. Since

```
CVMClassDescriptor* kap_desc
  desc : CVMClassDescriptor*
begin
cb := desc.class
n_params := desc.nparams
new_params = new CVMTypeDescriptor[n_params]
for i := 0 to n_params
  begin
  tb := cb.tvars[i]
  bound := resolve_entry(tb.class_bound_idx,desc,cb)
  switch desc.annotations[i]
    begin
    case 0: new_params[i] := desc.params[i]
    case 1: new_params[i] := registry_add_ftvar(desc.params[i],null)
    case 2: new_params[i] := registry_add_ftvar(bound,desc.params[i])
    case 3: new_params[i] := registry_add_ftvar(null,null)
    end
  end
return registry_add_class(desc.outer,cb,n_params,new_params)
end
```

Figure 4.37: The `kap_desc` routine

the captured descriptor is invariant, the registration process is then executed the usual way (see Section 3.4).

The internal representation of a fresh type-variable descriptor is given in Figure 4.38; a fresh type-variable descriptor contains two descriptors, namely `low_bound` and `upp_bound`, associated with the type-variable lower and upper bound, respectively. Such bounds are computed by the capture conversion routine described in Figure 4.37. For example, when a class descriptor of the kind `Pair<? extends String, Integer>` is captured, a new descriptor of the kind `Pair<Z,Integer>` is retrieved, where `Z` is a fresh type-variable descriptor whose upper bound is the class descriptor for `String` and whose lower bound is the special bottom descriptor `<nulltype>` — note that the second type-argument `Integer` is not affected by capture conversion, as its variance annotation is 0.

```
struct CVMFreshTVarDescriptor{
   CVMTypeDescriptor     header;
   CVMClassDescriptor*   low_bound;
   CVMClassDescriptor*   upp_bound;
};
```

Figure 4.38: The `CVMFreshTVarDescriptor` structure

4.3.1 Subtyping

The subtyping algorithm is obtained by a combination of standard inheritance and type argument containment (see Figure 4.39). In order to determine whether a class descriptor of the kind `D<`$\overline{\texttt{T}}$`>` is a subtype of another class descriptor of the kind `C<`$\overline{\texttt{V}}$`>`, the reification support must first lift the the descriptor for `D<`$\overline{\texttt{T}}$`>` to a parent descriptor of the kind `C<`$\overline{\texttt{U}}$`>` — this is accomplished by iteratively accessing the parent descriptor for `D<`$\overline{\texttt{T}}$`>` until a suitable class descriptor is found. If no such descriptor is found, the test fails immediately — the two class descriptors belong to unrelated inheritance hierarchies.

Once the descriptor has been lifted to the desired depth, a type-argument containment test is executed; this test consists in checking the intervals associated with the descriptors for the types in $\overline{\texttt{U}}$ and $\overline{\texttt{V}}$, respectively. In particular, for any given pair of descriptors of the kind `U`, `V`, the type-containment test ensures that `U` $\leq$ `V`. This is accomplished by recursively triggering a subtyping test on the upper/lower bounds of `U` and `V`, depending on the variance annotation associated with `V`. If `V` is invariant, the test checks that `U == V`; if `V` is covariant, the test checks that Δ^{+}(`U`) `<:` Δ^{+}(`V`); dually, if `V` is contravariant, the routine checks that Δ^{-}(`V`) `<:` Δ^{-}(`U`); finally, if `V` is bivariant, the type-containment test succeeds for any `U`. If the type-containment test finds a pair of type-arguments $\texttt{U}_\texttt{i}$, $\texttt{V}_\texttt{i}$ such that $\texttt{U}_\texttt{i} \not\leq \texttt{V}_\texttt{i}$, the subtyping test fails. If no such pair is found, the subtyping test succeeds.

For instance, the descriptor for `Triple<Integer,Number,String>` is a subtype of the descriptor for `Pair<? super Integer,?>`. In fact, the descriptor for `Triple<Integer,Number,String>` can be lifted to a parent descriptor

```
bool* is_subtype
  c1 : CVMClassDescriptor*
  c2 : CVMClassDescriptor*
begin
sup_desc := c1
while sup_desc.cb not equal to c2.cb   sup_desc := sup_desc.parent if sup_desc is
null
return false
if sup_desc is equal to c2
return true
for i := 0 to sup_desc.params.length
  begin
  ui := sup_desc.params[i]
  vi := c2.params[i]
  is_contained := false
  switch c2.annotations[i]
    begin
    case 0: is_contained := ui is equal to vi
    case 1: is_contained := is_subtype(upper(ui), upper(vi))
    case 2: is_contained := is_subtype(lower(vi), lower(ui))
    case 3: is_contained := true;
    end
  if not is_contained
    return false
  end
return true
end
```

Figure 4.39: The `is_subtype` routine

of the kind `Pair<Number,String>`; moreover we have that `Number` $\leq$ `? super Integer` — as `Integer <: Number` — and also that `Integer` $\leq$ `?`.

4.3.2 Captured Calls

In order to improve interoperability between wildcards and generic methods, Java allows to invoke a generic method passing as argument a wildcard type. Under such circumstances, one or more type-variables of the generic method are inferred with *fresh* type-variables — recalling from Section 2.2, the types of the actual arguments in a method call are subject to capture conversion.

The GCVM must handle captured calls, so that, given a generic method

```
struct CVMKapVarEntry{
   CVMDescriptorTableEntryHeader  header;
   CVMMethodBlock*                mb;
   CVMUint16                      pos;
   CVMUint16                      arg_idx;
};
```

Figure 4.40: The `CVMKapVarEntry` structure

call of the kind `<T>m()`, a dynamic inference process is applied in order to retrieve the actual types associated with the method type-variables can be retrieved — this inference step is unavoidable, as one or more types in $\overline{\texttt{T}}$ might be "hidden" behind wildcard types. The GCVM defines a special kind of descriptor entry, namely `CVMKapVarEntry`, used to represent the hidden parameter types in a generic method call. A `CVMKapVarEntry` is made up of the following fields (see Figure 4.40):

mb points to the `CVMMethodBlock` associated with the generic method being called;

pos the position of the actual argument to be used during the dynamic inference process;

arg_idx a class descriptor entry for the formal argument type to be used during the dynamic inference process.

The code in Figure 4.41 shows a generic method `copyFst()` that accepts an argument of type `Pair<U,V>` and returns a new pair (an object of type `Pair<U,Y>`), where the first value of the pair is copied from the pair passed as argument; this method is then supplied an argument of type `Pair<?,String>`. Note that the actual instantiation context for the generic method's type-variable `U` cannot be known until execution; in fact, such type is hidden behind the wildcard type `Pair<?,String>`. The method descriptor entry for the above captured call refers to a captured descriptor entry where: `mb` points to the `CVMMethodBlock` for `Pair.copyFst()`; `pos` is set to `1`, as the the actual argument — namely, `p` — to be exploited during the runtime

```
class Pair<X,Y> {
   ...
   <U,V> Pair<U,Y> copyFst(Pair<U,V> p) {
      return new Pair<U,Y>(p.x, y);
   }
   ...
}
...
Pair<String, Double> psd = ...
...
Pair<?,String> parg = new Pair<Integer,String>(1,"two");
psd.copyFst(parg);
```

Figure 4.41: A captured call

inference process is also the first argument in the generic method signature; `arg_idx` is an index to an open descriptor entry of the kind `Pair<U,V>`.

Given a captured call involving a generic method of the kind `<`$\overline{\texttt{X}}$`>m()`, the captured call inference process can be seen as a function that takes as input the runtime descriptors associated with the types of the actual arguments supplied to `m()` and yields the inferred runtime descriptors for the type-variables in $\overline{\texttt{X}}$ (see Figure 4.42). Let i be the position stored in the captured descriptor entry, and E be the descriptor entry associated with the i_{th} formal argument type of `<`$\overline{\texttt{X}}$`>m()`; the inference routine amounts at recursively scanning the descriptor table entry E until a suitable type-variable entry of the kind `m()#X` is found. If such a descriptor entry is found, the runtime descriptor for `X` can be accessed from the runtime descriptor associated with the type of the i_{th} actual argument, following the path discovered during the previous inference process.

In the example above, `infer_kap_desc` takes as input the runtime descriptor for `Pair<Integer,String>` (the runtime type of the `p` argument) and matches it against the formal argument type in the method signature `<U,V>copyFst(Pair<U,V>)`; consequently, the captured descriptor entry for `Pair.copyFst()#U` is resolved to the class descriptor for `Integer`, and the captured call is associated with a method descriptor of the kind `Pair<String,Double>.<Integer>copyFst()`.

```
CVMClassDescriptor* infer_kap_desc
  tb_to_infer : CVMTypeVarBlock*
  desc : CVMClassDescriptor*
  formal_desc_idx : CVMUint16
begin
entry := get_desc_table_entry(formal_desc_idx)
if entry.tag is CVMTypeVarEntry
  begin
  if entry.tb is tb_to_infer
    return desc
  else
    return null
  end
if entry.tag is CVMClassEntry
  begin
  for i := 0 to entry.nparams
    begin
    result := infer_kap_desc(tb,desc.params[i],entry.params[i])
    if result is not null
      return result
    end
  end
end
```

Figure 4.42: The `infer_kap_desc` routine

A full-blown support for captured calls must take into account `null` values passed as arguments, actual argument types that are subtypes of the formal types in the generic method signature, and the interplay between captured calls and open descriptors. For the sake of brevity, we do not cover such subtle issues here — though they are fully implemented in the GCVM.

5 Benchmarks

The GCVM introduces three kinds of overhead in the execution of generic code: execution speed overhead, memory overhead and classfile size overhead. Execution speed overhead is mainly due to the need to manage runtime descriptors when executing type-dependent operations involving generic types. Memory overhead is caused by the descriptor registry, used to keep track of all the runtime descriptors used by an application. Finally, classfile size overhead

is due to the additional bytecode attributes available in the generified classfiles. Note that a reification support is essentially a runtime infrastructure for a new language, namely, a slight extension of Java where generics and wildcards are treated as first-class types — seamlessly usable in type-dependent operations. Such a language is currently not deployed, hence it is very difficult to gather large-size source code upon which performing correctness/performance tests.

In the remainder of this section we discuss the performance of the GCVM with respect to small-size synthetic programs specifically designed to measure the execution speed overhead associated with type-dependent operations involving generic types, such as generic instance creation expressions and generic method calls — our analysis takes into account both monomorphic and polymorphic call-sites. We then conclude this section by illustrating a real world benchmark: the GJ compiler [Mic01].

5.1 Microbenchmarks

Our first set of benchmarks consist of "microbenchmarks" designed to measure the performance of code involving generic instance creation expressions and generic method calls (see Figure 4.1). Each microbenchmark consists in repeatedly executing the same generic operation several times; we dropped the first iteration of each benchmark from the computed average because it deviated significantly from the remaining 20 iterations. We presume that the source of this deviation is the overhead associated with the GCVM startup, possibly affected by other operating-system dependent factors such as caching, etc. — which we don't want to discuss here. After dropping the first run, the variance among iterations for each benchmark was less than 1%.

Our goal was to measure the overhead associated with the handling of type descriptors — e.g. resolution of descriptor table entries, descriptor registry lookup, virtual method table management, and so forth; recalling from Section 4.1, there is a fundamental difference between open and closed entries: the runtime type associated with a closed entry does not depend on the runtime type of the class/method in which the entry is used. This allows for a compact and efficient implementation, as the runtime descriptor corresponding to a closed entry can be *cached* (see Section 3.5) inside the descriptor table, so that

	NC	NO	MC	MO	PC	PO
CVM	1104 ms	1095 ms	383 ms	384 ms	435 ms	435 ms
GCVM	1234 ms	1272 ms	466 ms	530 ms	709 ms	713 ms
Overhead	11.74%	16.25%	21.66%	38.10%	63.02%	63.70%

(a) Benchmark results: (NC) `new` closed, (NO) `new` open, (MC) closed generic method call with monomorphic call-site, (MO) open generic method call with monomorphic call-site, (PC) closed generic method call with polymorphic call-site, (PO) open generic method call with polymorphic call-site

	IMC	IMO
CVM	446 ms	437 ms
GCVM	2203 ms	2825
Overhead	394%	547%

(b) Benchmark results: (IMC) closed virtual method call involving interface or wildcards, (IMO) open virtual method call involving interface or wildcards

Table 4.1: Microbenchmark results

subsequent accesses are immediate. On the other hand open descriptor entries must undergo a heavy-weight resolution process (see Section 4.1.2), where each type-variable `X` in the descriptor entry is replaced for an actual type `T` by looking at the so called *bounding descriptor*. This distinction is reflected in the results in Figure 4.1a, as type-dependent operations involving closed descriptor entries are significantly faster than their counterparts involving open descriptor entries.

Another important distinction is between monomorphic and polymorphic call sites; recalling from Section 4.2, generic method calls featuring dynamic dispatching pose severe performance issues, as the runtime descriptor associated with a given virtual method call cannot be determined until execution. In order to overcome this problem, the GCVM supports a sophisticated caching technique (see Section 4.2.1) that significantly reduces the overhead associated with the handling of runtime method descriptors; however, as

	Execution time	Memory (peak)	Classfile size
CVM	4523 ms	14216 KByte	340.7 KByte
GCVM	4594 ms	14586 KByte	353.4 KByte
Overhead	1.57%	2.48%	3.7%

(a) Overall

Kind	Amount
Class	77
Method	11
Array	6

(b) Runtime descriptors

Opcode	Amount
`new`	260048
`new_array`	1483
`new_multiarray`	0
`invoke_xxx`	11573
`instanceof`	30729
`checkcast`	19742

(c) Rewritten opcodes

Table 4.2: The GJ benchmark

shown in Figure 4.1a, execution of virtual method calls is still significantly slower compared to the execution of non-virtual method calls — we believe this difference is mainly related to the routines required to maintain the consistency of the VPMT (see Section 4.2.2). However it is important to notice that in cases where the VPMT cannot be leveraged — e.g. in virtual method calls involving interface/wildcard types — the execution is even slower (see Figure 4.1b); this corresponds to the case limit where no caching is possible, so that the runtime descriptor associated with the virtual method call must be dynamically resolved upon each new call.

5.2 Real World Benchmark: GJ

The overhead introduced by our approach highly depends on the relative amount of generic features used by an application; therefore, the only significant measurement results can be obtained over real world application of

medium/large size. In our benchmarks we considered the GJ compiler, which largely relies on generics and performs several type-dependent operations involving generic types and wildcards, such as allocation of generic objects, generic virtual method calls and legacy-style type conversions to unbounded generics such as `C<?>`. Our benchmark consisted in running the GJ compiler in order to compile a fixed set of classes; the results of the GJ benchmark are summarised in Figure 4.2.

Note that the execution-time overhead introduced by the GCVM when executing the GJ benchmark is not significant (∼1.5%) (see Figure 4.2a; this result has been obtained in spite some operations involving wildcards, such as virtual generic method calls, captured calls and subtyping, are intrinsically more complex than their non reified counterparts. An important point is that, however, such type-dependent operations are likely to be infrequent and they do not affect real world benchmarks — indeed, in currently deployed applications, generics and wildcards remain a mainly static mechanism for type-safety.

Moreover, as shown in Figure 4.2b, the descriptor registry plays a crucial role in minimising the number of runtime descriptors that need to be created during execution — and, consequently, to reduce dynamic memory footprint. The number of runtime descriptors created while executing the GJ benchmark is relatively low compared to the overall number of type-dependent opcodes that have been instrumented with the special `opc_load_desc` instruction, shown in Figure 4.2c.

Chapter 5

Multi-paradigm Integration with Generics, Wildcards and Annotations

In this chapter we discuss a framework called PatJ [CV07, CV08a] which promotes seamless exploitation of Prolog programming in Java[1]. Integrating Object-Oriented and logic programming has been the subject of several researches and corresponding technologies; such proposals come in two flavours, either attempting at joining the two paradigms as in [Esp06, ON94], or simply providing an interface library for accessing Prolog declarative features from a mainstream Object-Oriented languages such as Java as in [tuP02, swi, Min, k-p, JLo02]. Both solutions have however drawbacks: in the case of hybrid languages featuring both Object-Oriented and logic traits, such resulting language is typically too complex, thus making mainstream application development an harder task; in the case of library-based integration approaches there is no true language integration, and some "boilerplate code" has to be implemented each time to fix the paradigm mismatch.

Our aim is to introduce a novel approach that combines the expressive power of Java generics and the flexibility of Java annotations, in order to define a precise mapping between Object-Oriented and logic programming features. PatJ defines a hierarchy of classes where the bidirectional semantics of Prolog terms is modeled directly at the level of the Java generic type-

[1] PatJ is available for download at the URL `http://trac.alice.unibo.it/trac/pj/`.

system — this API is a noticeably sophisticated application of Java generics and wildcards. On top of this generic API, PatJ provides custom Java *annotations* [JGSB05, Mica] to be used for embedding Prolog theories within Java classes, so as to specify Prolog code as a possible implementation of given Java methods or fields.

The idea of using annotations for extending the Java language is not new: for instance, *AspectJ*/*AspectWerkz* [BKG+06, Bon04], which are very popular aspect-oriented extensions of the Java programming language, use Java annotations for declaring aspects, pointcuts, and advices. Similarly, in [ANMM06], a framework is described that supports *pluggable type systems* in the Java programming language. Other remarkable applications of Java annotations include: simplifying code of enterprise Java applications [Sun09], associating rich semantic assertions to Java code [LBR06], building frameworks for detecting anomalies (e.g. deadlocks) in concurrent Java programs [GHS05], and enforcing the static type-checking of the Java language for e.g. detecting nullability constraint violations [PAC+08] or detecting immutable references [ZPA+07].

Other than Java-Prolog integration, we believe the work discussed in this Chapter provides general hints on how generic programming can successfully turn libraries into smooth language extensions, making Java a flexible platform for customising the programming model according to the application needs.

1 Object-Oriented vs. Logic Programming: a Comparative Study

Object-Oriented programming and logic programming are two very complimentary programming paradigms. On the one hand, in the Object-Oriented paradigm, computation can be viewed in terms of *messages* that are exchanged between entities, called *objects*. Operations performed by objects are typically expressed in an *imperative* style, as a sequential flow of *instructions* that change the state of a program.

By contrast, in logic programming, a program is structured as a set of axioms and inference rules (a *theory*), that are used in order to assert the

validity of a given logic predicate, called *goal*. Consequently, logic programming is intrinsically *declarative*, as it gives the programmer the ability to express the *logic* of a computation without expressing its control flow — this is accomplished by focussing on *which* goals the program should accomplish, rather than *how* to accomplish them.

An Object-oriented programming language allows the programmer to define software entities — namely objects — which closely model real world artifacts, so that the complexity of the solution is typically reduced [GHJV95, Mey89]. A key concept of Object-Oriented programming is that there exists a clear distinction between the set of operations defined by an object (its *interface*) and the object's internal representation (its *implementation*). This simplifies the task of making minor changes e.g. in the data representation or the procedures of an object — in class-based Object-Oriented languages, this is typically accomplished by changing the code of the object's class — without affecting other parts of a program: inter-class consistency is guaranteed, as long as the object's public interface remains the same. The availability of reusability mechanisms such as subtype and parametric polymorphism (see Section 1) makes it easy to add new features on top of existing ones so that the same software entities can be used in several contexts.

On the other hand, logic programming languages are considered to be well-suited for expressing complex problems because most of the low-level machinery (memory management, pointers, etc.) are *hidden* to the programmer — they are left to the computational engine. Logic programming allows for a more natural representation of the problem's domain [Kow74, VEK76], usually offering the opportunity to represent data both *extensionally* — as explicit *facts* of the kind "*Nodes A and B are connected*" — and *intensionally* — as *inference rules* which implicitly describes how to obtain valid assertions from existing ones, such as "*if, given pair of connected nodes X and Y, Y is connected to a third node Z, then X and Z are also connected*".

A multi-paradigm integration allowing interoperability between Object-Oriented and logic programming would allow applications to take advantage of all the features discussed above: such a framework would allow for strong object-based encapsulation, thus maximising the opportunity for code-reuse;

at the same time it would allow applications to take advantage of key assets of logic programming, such as adaptiveness and non-determinism.

In the remainder of this section, we discuss some of the key differences between Object-oriented and logic programming; we grouped these differences into two main categories: *data-binding* and *execution semantics*. The former focusses on the data-types available in a given programming paradigm and on how custom data-types can be defined; the latter focusses on how computation is expressed in a given programming paradigm. Our discussion focusses on two programming languages such as Java and Prolog — these are perhaps the most popular choices in the Object-Oriented and logic programming domains, respectively.

1.1 Object-Oriented Programming in Java

In this section we provide a brief overview of how computation is expressed in the Java Programming language. More specifically, we discuss the builtin data-types available in Java, and, most importantly, the powerful `class` keyword, that allows programmers to define custom abstract data-types — a Java class is used as a template for building objects. A class defines a set of variables, or *fields*, that are used to model the state of an instance of that class; moreover a class provides a set of operations, or *methods*, that can be used by clients in order to *manipulate* the state of an object of that class. Computation in Java is thus accomplished through *message passing* — that is, by calling methods on objects.

1.1.1 Builtin Types and Classes

The syntax of Java values is reported in Figure 5.1a; a value in Java is either a *primitive*, such as a numerical value (either integral, as `25`, `0x1f` or floating-point, as `14.2`, or `1.234e2`), a boolean (constant values `true` `false`) and a character literal, enclosed in single quotes (e.g. `'$'`, `'\u0220'`) or, alternatively, a *reference* to a Java object. Java objects are allocated using the `new` operator (e.g. `new Foo()` where `Foo` denotes a Java class).

Java is equipped with two kinds of builtin classes: arrays and strings. Arrays are instances of the class `java.lang.Array`; they can be initialised

`v`	::=	`p`	primitive values
	|	`new C(`$\overline{\texttt{v}}$`)`	objects
	|	`new C[] {` $\overline{\texttt{v}}$ `}`	arrays

(a) Syntax of Java values

`t`	::=	`a`	atoms
	|	`n`	numbers
	|	`V`	variables
	|	`f(`$\overline{\texttt{t}}$`)`	compound

(b) Syntax of Prolog terms

Figure 5.1: Builtin data-types in Java and Prolog

using braces (e.g. `{1,2,3}` denotes an array of `int`) and array elements can be accessed using the '`[]`' operator (e.g. `a[2]` retrieves the third element of the array `a`). Strings are instances of the class `java.lang.String`; as for arrays, Java provides special syntax shortcuts for creating strings - the string literal `"Hello!"` can be regarded as a syntactic sugar for the expression `new java.lang.String("Hello")`.

Java defines an extensive set of class libraries [Mica], that can be regarded as additional builtin types; for instance, the Java Collection Framework defines general purpose containers like lists (e.g. `java.util.ArrayList<E>`), stack (e.g. `java.util.Stack<E>`), dictionaries (e.g. `java.util.Map<K,V>`). Class libraries typically lacks the syntactic sugar available for builtins Java classes such as arrays or strings. For instance, a collection object is created using the `new` operator — as for any other user-defined class —, while e.g. retrieving an element from a collection object is accomplished by calling specific a method on that object (e.g. `get()`).

1.1.2 Defining Custom Classes

The `class` keyword is used to define custom abstract data-types. Java classes can define one ore more *fields*; a field is an object variable that can be used to represent the *internal state* of the abstract-data type. The code in Figure 5.2a shows a simple Java class representing binary trees. This class defines three fields, namely `value` — an integer value attached to each tree node —

```
class BinaryTree {
   BinaryTree left, right;
   Integer value;

   BinaryTree(Integer value) {
      this.value = value;
   }
   BinaryTree(Integer value, BinaryTree left, BinaryTree right) {
      this(value);
      this.left = left;
      this.right = right;
   }
}
```

(a) A Java class for encoding binary trees

```
tree(H,X,Y) :- number(H), valid(X), valid(Y).
valid(tree(H,X,Y)) :- number(H), valid(X), valid(Y).
valid(nil).
```

(b) A Prolog definition of a binary tree

Figure 5.2: Binary trees in Java and Prolog

`left` and `right` — the subtrees of a given node. Objects are created using the keyword `new`, followed by the name of the class to be used as template, optionally followed by an argument list (the actual values to be supplied to the class constructor):

```
BinaryTree bt = new BinaryTree(1,
                               new BinaryTree(2),
                               new BinaryTree(3));
System.out.println(bt.left.head); //prints 2
```

The above code creates a binary tree where: `value` is the integer value `1`; the left subtree is a binary tree object of the kind `new BinaryTree(2)`; the right subtree is a binary tree object of the kind `new BinaryTree(3)`. Fields can be accessed using the '`.`' operator; for instance, the expression `bt.left.head` is used to select the value of the first subtree of the binary tree object `bt`.

In addition to fields, Java classes can define *methods*. A Java method is a piece of code that can manipulate one or more class fields in order to

perform some computation; for example, the code in Figure 5.3a shows a method, namely `count()`, that computes all the occurrences of a given value in a binary tree object. This method accepts one argument of type `Integer`, namely `elem`; the method body recursively calls `count()` on both subtrees (if they are non-null values) so that `elem` is recursively found in the graph induced by a binary tree object. Let $\texttt{count}_{\texttt{left}}$ and $\texttt{count}_{\texttt{right}}$ be the number of occurrences of `elem` in the left and right subtrees, respectively, and let `i` be an integer value that is set to `1` if the head of the binary tree is equal to `elem`, and `0` otherwise; at each recursion step the method yields the sum between $\texttt{count}_{\texttt{right}}$, $\texttt{count}_{\texttt{right}}$ and `i`:

```
bt.count(1); → 1
bt.count(3); → 1
bt.left.count(3) → 0
```

We conclude this brief overview by noting that Java is a strongly typed language; as such, the compiler statically checks that method/constructor calls are well-formed — that is a method/constructor must be supplied the correct number of actual arguments, where the type of each argument must be *convertible* [JGSB05] to the expected one. Any failure to do so will result in a compile-time failure:

```
BinaryTree bt2 = new BinaryTree(""); //type-mismatch
bt.count(1,2); //too many parameters
```

1.2 Logic Programming in Prolog

In this section we discuss how computation is expressed in the Prolog programming language. More specifically, we show the builtin data-types available in Prolog and we show how such data can be manipulated by Prolog facts and rules. The main goal of this section is to illustrate the key differences between Java programming and Prolog programming; such analysis will come handy at a later point when we will discuss the requirements that must be matched in order to bridge the semantic gap between Object-Oriented and logic programming.

```
class BinaryTree {
   ...
   int count(Integer elem) {
      return value == elem ? 1 : 0 +
             left != null ? left.count(elem) : 0 +
             right != null ? right.count(elem) : 0;
   }
}
```

(a) A Java method

```
count(nil, E, 0).
count(tree(H, X, Y), H, R) :- count(X, H, R1),
                              count(Y, H, R2),
                              R is 1 + R1 + R2.
count(tree(H, X, Y), E, R) :- E =\= H,
                              count(X, E, R1),
                              count(Y, E, R2),
                              R is R1 + R2.
```

(b) A Prolog predicate

Figure 5.3: Manipulating binary trees in Java and Prolog

1.2.1 Terms

Every data value in Prolog is a *term* expressed by the syntax shown in the lower part of Figure 5.1; a term `t` is either an *atom* `a` (an unstructured literal, optionally enclosed in single quotes, as `car`, `'Bob'`, etc.), a number `n` (either an integer as 42, or a floating point 15.2), a logic variable `V` (a variable that can be bound to a value during computation, expressed as a literal starting with a capital letter), or a compound term of the kind $\texttt{f(}\overline{\texttt{t}}\texttt{)}$, where `f` is the functor name and each $\overline{\texttt{t}}$ denotes a list of terms.

A Prolog list is a term of the kind $[\texttt{t}_1, \ldots, \texttt{t}_n]$ (or $[\texttt{t}_h|\texttt{t}_t]$ where $\texttt{t}_h$ is the head and $\texttt{t}_t$ is the tail), which Prolog implementations handle as special cases of compound terms. In fact, all Prolog lists are represented as binary compound terms, whose functor is '.' and whose first and second arguments are the list's head and tail respectively (e.g. the Prolog list `[1,2,3]` is represented by the compound term `'.'(1,'.'(2,'.'(3, [])))` where the special atom `[]` denotes the empty list.

1.2.2 Facts and Rules

Prolog has no builtin mechanism to define custom data-types. Instead, special compound terms called *clauses* can be fruitfully exploited to define structured data. A clause is term of the kind '`H :- B`', where `H` denotes the head of the clause, and `B` denotes the clause body. In the following, the term "fact" is used to denote a Prolog clause with an empty body; dually the term "rule" is used to denote a Prolog clause with a non-empty body. Prolog clauses are internally represented as binary compound terms, whose functor is '`:-`' and whose first and second arguments are the clause head and body, respectively.

A possible Prolog implementation of a binary tree is shown in Figure 5.2b; the Prolog code actually goes far beyond the mere definition of a data-structure - it actually defines a ternary *relation* between a value, and two subtrees, called *fact*; the distinction between structure and behaviour is here completely blurred, as a Prolog fact is also a full-fledged computational artifact. Note that, in order to preserve the semantics of the Java binary tree representation, the code needs to perform some checks on the kinds of subterms associated with a given fact of the kind `tree(X,Y,Z)` — these checks are required as Prolog is not a strongly typed language. For instance, we need to ensure that e.g. the value of a binary tree is a number, and that the subtrees are either the special atom `nil` (used to encode the empty binary tree) or compound terms of the kind `tree(X,Y,Z)`:

```
tree(1,tree(2,nil,nil),tree(3,nil,nil)). //ok
tree(a,nil,nil). //no - number(a) is not true
```

In order to understand the semantic differences between Java and Prolog, consider the simple Prolog predicate `count(T,E,R)`, which holds when `R` is the number of all occurrences of a given value `E` inside a binary tree `T` (the behaviour of this Prolog predicate is equivalent to the Java method `count()` shown in Figure 5.3a). A possible implementation for the `count` predicate is given in Figure 5.3b.

In Prolog, computation is expressed in a declarative fashion, by specifying a set of *rules*. For instance, the first rule of the `count` predicate states that the `count` relation is defined whenever `T` is the empty tree — namely an atom of

```
?-count(tree(1,tree(2,nil,nil),nil),4,1). → no
?-count(tree(1,tree(2,nil,nil),nil),2,X). → yes, X/1
?-count(tree(1,tree(2,nil,nil),tree(2,nil,nil)),E,1). → yes, E/1
?-count(T,2,R). → yes, T/nil;R/0,
                       T/tree(2,nil,nil);R/1,
                       T/tree(2,nil,tree(2,nil,nil));R/2,
                       T/tree(2,nil,tree(2,nil,tree(2,nil,nil)));R/3,
                       ...
```

Figure 5.4: Different ways of exploiting the `count/2` predicate

the kind `nil` — and `R` is `0`, regardless of `E` (that is, the number of occurrences of *any* value in an empty tree is 0). The second and the third rules are more complex; they state that, if `R1` and `R2` are the number of occurrences of the value `E` in the subtrees `X` and `Y`, respectively, then the number of all the occurrences of `E` in a a tree of the kind `tree(H,X,Y)` is either `1 + R1 + R2` — if $\texttt{H} = \texttt{E}$ — or simply `R1 + R2` — (if $\texttt{H} \neq \texttt{E}$). Therefore, Prolog predicates do not correspond to a concrete execution flow — rather, they are full-fledged declarative entities upon which the Prolog engine can reason, make assertions, etc.

For instance, the above `count` predicate can be exploited in several different ways, as shown in Figure 5.4. More specifically, the user can ask *(i)* whether the number of occurrences of the value `4` in a binary of the kind `tree(1,tree(2,nil,nil),nil)` is `1` (Prolog replies no), *(ii)* what is the number of occurrences of the value `2` in a binary tree of the same kind (Prolog replies with primitive value 2), *(iv)* what are the elements that have no duplicates in a binary tree of the kind `tree(1,tree(2,nil,nil),tree(2,nil,nil))` (Prolog replies with the value 1), and finally *(iv)* for a binary tree containing the value `2` an unspecified number of times (Prolog iteratively provides all the binary trees of depth n containing the value `2`, for increasing values of n).

Hence, the arguments of the `count` predicate are truly *bidirectional*: they can act either as inputs or outputs, depending on whether a variable term or a ground term is supplied. The ability of supporting this peculiar feature is not a mere programming mechanism of Prolog, but it is a core difference

between Object-Oriented and logic programming models.

1.3 Prolog Predicates vs. Java Methods

Prolog terms are easily mapped into an Object-Oriented hierarchy of classes whose root is the abstract class/interface `Term` — this stems from the fact that every value in Prolog is implicitly a term. Subclasses are then defined for each concrete Prolog term, such as `Atom`, `Int`, `Struct` (for compound terms), `Var` (for logic variables) and so on — this approach is successfully exploited in almost all library-based integration approaches.

In such a setting, one might be tempted to leverage the Java term hierarchy to e.g. define a Java method modelling a Prolog predicate; suppose we want to define a Java method mapping the following Prolog predicate `lenght`:

```
length([], 0).
length([_|T], S):- length(T, S2), S is S2 + 1.
```

The predicate `length(L,S)` holds whenever `L` is a list containing exactly `S` elements. At first, such a predicate can be viewed as a Java method that, given a list (of type `Struct` — as Prolog lists are a special case of compound terms) simply returns its size (a term of type `Int`):

```
Int length(Struct s)
```

The signature above, however, does not fully capture the semantics of the original Prolog predicate; more specifically, bidirectionality is lost, since an input/output role is implicitly assigned to each variable in the corresponding Prolog predicate `length(L,S)`. In other words, the above method can be understood as a specific *instance* of a Prolog predicate of the kind `length(L,S)`, where `L` acts as an input — it appears in the method argument list — while `S` acts as an output — the return value of the method.

There are two possible mappings that preserve the bidirectional semantics of the original predicate: an *heterogeneous* mapping, which requires several method signatures — one for each distinct configuration of the predicate variables — and an *homogeneous* mapping, where the type `Term` is used to abstract over the concrete types of the terms in the method signature.

A possible heterogeneous translation of the Prolog predicate `length` is reported below:

```
boolean length(Struct l, Int s) //L input, S input
boolean length(Var l, Int s) //L output, S input
boolean length(Struct l, Var s) //L input, S output
boolean length(Var l, Var s) //L output, S output
```

Each overloaded version of `length()` corresponds to a different configuration of the variables `L` and `S`, respectively; for instance, the method `length(Struct,Var)` can be used to retrieve the size of a given list; similarly, the method `length(Struct,Int)` can be used to check as to whether a list has a given size, and so forth. This solution does not scale particularly well: in the general case, given a predicate of the kind $\texttt{p}(\overline{\texttt{t}})$ of arity n, the mapping defined by such heterogeneous translation scheme requires 2^n different Java signatures — one for each possible input/output configuration of the subterms in $\overline{\texttt{t}}$.

By contrast, the homogeneous translation scheme unifies all possible usages of the subterms `L` and `S` into a single method signature accepting two objects of type `Term` — this is possible since every concrete Prolog term is an instance of some subclass of `Term`:

```
boolean length(Term l, Term s)
```

This signature preserves the bidirectionality of the original Prolog predicate: since both `Struct`, `Int` and `Var` are subclasses of `Term`, it is possible to call `length()` with e.g. a ground list (of type `Struct`) and a variable (of type `Var`), or with two variables (of type `Var`), and so forth. This is however problematic: the above signature turns out to be applicable even in cases where the binary relation expressed by the original Prolog predicate is undefined — again, this is possible because the type of the formal arguments is simply `Term`, the root of our term class hierarchy.

Requirement 1. The integration support should map execution of Prolog queries on Java method calls, in order to greatly reduce the semantic gap between logic and Object-Oriented programming. An interesting case is when the Prolog goal term `G` is a predicate of the kind $\texttt{p}(\overline{\texttt{t}})$ with arity n, where the subterms in $\overline{\texttt{T}}$ contain only *one* logic variable. In this case, a straightforward mapping is given so that `G` can be modeled as a Java

```
class Permutation {
   List<Integer> nextPerm(List<Integer> arr) {
      List<Integer> a = new ArrayList<Integer>(arr);
      int n = a.size() - 1;
      int j = n - 1;
      while (a.get(j) > a.get(j+1)) {
         if (j==0) {
            return null; //last permutation
         }
         j--;
      }
      int k = n;
      while (a.get(j) > a.get(k)) k--;
      int tmp = a.get(j); a.set(j, a.get(k)); a.set(k, tmp);
      int r = n;
      int s = j + 1;
      while (r > s) {
         tmp = a.get(r); a.set(r, a.get(s)); a.set(s, tmp);
         r--; s++;
      }
      return a;
   }
   List<List<Integer>> permutation(List<Integer> l) {
      List<List<Integer>> perms = new ArrayList<List<Integer>>();
      while (l != null) {
         perms.add(l);
         l = nextPerm(l);
      }
      return perms;
} }
```

Figure 5.5: Permutations in Java

method `g()` accepting $n-1$ arguments so that a method call of the kind `g(`$\overline{\texttt{a}}$`)` effectively corresponds to a Prolog query of the kind `G(`$\overline{\texttt{a}}$`)`. Moreover, a Java method defined by this mapping should preserve (as much as possible) the semantics of the corresponding Prolog predicate — concepts such as bidirectionality of predicate variables should be supported.

1.4 Prolog in Java: Library-based Integration

In this section we discuss the most common problems that developers have to face when bridging the gap between Object-Oriented and logic programming. Integration is usually accomplished by means of a library that allows e.g. a

```
Term[] termArray = new Term[3];
for (int i = 0;i<list.size();i++) {
   term_array[i]=new Int(list.get(i));
}
Struct pl_list = new Struct(term_array);
Var x = new Var("X");
Struct goal = new Struct("permutation",pl_list,x);
```

(a) Creating the goal term

```
String theory = ... //contains the Prolog theory for permutation/2
Theory t = new Theory(theory);
Prolog engine = new Prolog();
engine.setTheory(t);
```

(b) Setting up the Prolog engine

```
SolveInfo solution = engine.solve(goal);
if (solution.isSuccess()) {
   Struct s = solution.getTerm("X");
   ... //do something with list s
}
solution = engine.solveNext();
while (engine.hasOpenAlternatives()) {
   ...
}
```

(c) Browsing solutions

Figure 5.6: Permutations in tuProlog

Java program to define Prolog terms, and predicates that can be queried upon. As a concrete use case we refer to the tuProlog engine [DOR05, tuP02], a lightweight, full-fledged Prolog engine entirely written in Java — however similar conclusions hold for other library-based integration approaches.

Assume we want to exploit the following Prolog theory for generating all permutations of a list:

```
remove([X|Xs],X,Xs).
remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).
permutation([],[]).
permutation(Xs,[X|Ys]):-remove(Xs,X,Zs),permutation(Zs,Ys).
```

Predicate `remove` takes a list, an element, and the list after removing the

element, while predicate `permutation` takes a list and a permuted version of it — syntax `[X|Xs]` stands for a list with head `X` and tail `Xs` as usual. Though this is just an explanatory example, a Java programmer might enjoy how the permutation algorithm is easily resolved in Prolog — especially if compared with its Java equivalent (see Figure 5.5) — and accordingly be willing to use it in a Java application to compute permutations of Java collections.

As we have seen, Java and Prolog have two fundamentally different ways to represent data; consequently, the first problem a programmer needs to face consists in manually mapping Java values into Prolog terms. This step is a common trait of all library-based integration approaches: executing a Prolog query amounts at building a Prolog term (usually a predicate) containing some variables — the placeholders that will be filled once the query has been solved. Such conversion is typically done by building object-based representations of Prolog terms — in the case of the tuProlog engine, such objects are instances of the `Term` class. Assuming that `list` is an object of type `LinkedList<Integer>` containing the values `1`, `2`, and `3`, the code snippet in Figure 5.6a is used to create a goal of the kind `permutation([1,2,3],X)`, asking for any list `X` which is a valid permutation of `[1,2,3]`.

This conversion code has a repetitive structure — we call such code "boilerplate": Java values (either objects or primitives) are to be converted into a suitable Prolog representation, so that they can be understood by the Prolog engine. This approach does not scale well, as the amount of code that needs to be written in order to build a goal term of the kind `p(`$\overline{\texttt{s}}$`)` grows linearly in the *depth* of the subterms in $\overline{\texttt{s}}$.

Requirement 2. The integration support should allow for an easy, straightforward mapping between Java values (either primitive or reference) and Prolog terms and vice-versa. More formally, an integration framework should supply a marshalling function $m := \mathbb{J} \rightarrow \mathbb{P}$ that, given a Java value $v \in \mathbb{J}$ returns its corresponding representation as a Prolog term $v' \in \mathbb{P}$. Conversely, the framework should also supply an unmarshalling function $u := \mathbb{P} \rightarrow \mathbb{J}$ that, given a Java representation of a prolog term $p \in \mathbb{P}$ returns its corresponding representation as a plain Java value $p' \in \mathbb{J}$, so that $u(m(v)) = v$.

Library-based integration approaches typically provide a Java class modelling a Prolog engine; this class provides the necessary methods for executing Prolog queries. The tuProlog engine provides a class — namely `Prolog` — that can be instantiated (as any other Java class) and initialised with a `Theory` object containing a snippet of Prolog code (usually a Java string where predicates are separated by newline characters). Assuming that the Prolog code for computing permutations is fitted into a Java string, the code in Figure 5.6b is required in order to create and initialise the tuProlog engine class.

Since typically (as in this case) more than one solution is supplied, the Prolog engine class provides some basic support for browsing the solution space associated with a given Prolog query. Note that simply returning an array containing all the solutions is not an option, since, in general, execution of a Prolog query is not guaranteed to terminate (e.g. it is possible for the Prolog engine to return some solutions, and then to hang). In order to overcome this problem, the tuProlog engine allows the programmer to iteratively retrieve all the n solutions of a given query — this is accomplished by making n calls to the `Prolog` engine class, as shown in Figure 5.6c.

For each solution we must determine e.g. whether the solution is valid and whether other solutions are available; such boilerplate code has the rather unpleasant effect of hiding the user code effectively processing the solutions retrieved by the Prolog engine. Note also that solutions are Prolog terms; as such, an additional conversion might be required should a more suitable Java representation be needed by the user code. In this case, since it is known that permutations are indeed Prolog lists, the user code might be interested in converting those lists back to plain Java Collection objects — thus requiring further bridge code.

Requirement 3. The integration support must provide better support for performing common tasks such as iteratively browsing the solution space associated with a given Prolog query; if a prolog predicate is known to yields multiple results, a mapping should be defined so that such results can be easily accessed from Java code e.g. using a *for-each* loop.

2 Prolog from Java: Basic PatJ

PatJ is a framework that greatly enhances interoperability between Java and Prolog; the key idea of the PatJ framework is to provide a way so that Java methods can be implemented declaratively — that is, in terms of Prolog rules and facts. Among the various mechanisms we provide, a Java `abstract` method can be decorated with a custom Java annotation [JGSB05] that is used to define a Prolog-based implementation — we call such a method *Prolog method*. Thanks to reflection and Java Dynamic Proxy classes [Mica] (*proxies* in the following), PatJ is able to synthesise a concrete implementation of a Prolog method; consequently, from the user perspective, the computation of a given Prolog query is triggered by a simple method call — no boilerplate code is required for interfacing with the underlying tuProlog engine.

Java generics and wildcards plays a crucial rule in the PatJ framework; first, PatJ defines a hierarchy of *generic* Java classes modelling first-order logic terms that features automated marshaling/unmarshaling from Java to Prolog, and viceversa; the bidirectional semantics of Prolog terms is hence modeled directly at the level of the Java generic type-system — this API is a noticeably sophisticated application of Java generics and wildcards. Secondly, generics are used in order to define how Prolog method arguments should be rearranged in the corresponding Prolog query — this is accomplished by introducing a mapping between the type-variables of a generic Prolog method and the logic variables in the Prolog predicate modeled by that method.

Any declarative feature provided by PatJ (e.g. Prolog method call) is implemented in terms of requests to an underlying tuProlog engine — in fact, the core of the PatJ framework can be seen as a tiny wrapper around the tuProlog engine providing just basic capabilities such the ability of retrieving Prolog solutions using a Java iterator. On top of this layer lies the PatJ runtime, the most important part of the PatJ framework; the PatJ runtime handles the creation of dynamic proxy classes that allow the framework to intercept Prolog method calls and to dispatch them — after an appropriate transformation — to the underlying tuProlog engine.

```
abstract class Term<X extends Term<?>> { ... }
class Atom extends Term<Atom> { ... }
class Int extends Term<Int> { ... }
class Double extends Term<Double> { ... }
class List<X extends Term<?>> extends Term<List<X>> { ... }
class Var<X extends Term<?>> extends Term<X> { ... }
abstract class Comp<X extends Comp<?>> extends Term<Comp<X>> { ... }
```

Figure 5.7: The PatJ generic term hierarchy

2.1 Modelling Prolog Terms in PatJ

PatJ introduces a strongly-typed hierarchy of generic classes, which enhances the static type checking carried out by the Java compiler, and flexibly expresses the bidirectionality of Prolog terms: this allows to define a complete mapping from Java method signatures to Prolog predicates.

The root of the PatJ term hierarchy is the generic class `Term<X>` (see Figure 5.7); using a recursive pattern exploiting wildcard types [JGSB05, IV06], type-variable `X` is used to abstract over the type of the actual content of the term. Hence, a `Term<Int>` will be a term keeping an `Int`, `Term<Double>` a `Double`, `Term<List<Int>>` a `List<Int>`, and so on. On the other hand, variables are handled differently: the generic class `Var<X>` is a wrapper for a term with type `X` and is defined as a straight subtype of `Term<X>` (rather than `Term<Var<X>>`, as in the above cases). As a result of this careful design choice, the type `Term<Atom>` is a common supertype of both `Atom` and `Var<Atom>`; in other words, the type `Term<T>` can be used to abstract over the *role* of a Prolog term. For example, when an argument to a method needs to be either a logical input or output, it can be given the type `Term<T>`, so that one can pass either an actual term `T` (input), or a variable `Var<T>` (output) which will hold the result term (of type `T`) after computation is over.

The PatJ term hierarchy overcome the limitations of both translation schemes discussed in Section 1.3; a general solution for the signature of method `length()` is the following:

```
boolean length(Term<? extends List<?>> list, Term<Int> size)
```

```
abstract class Comp<X extends Comp<?>> extends Term<Comp<X>> { ... }
class CmpNil extends Comp<CmpNil> { ... }
class CmpCons<H extends Term<?>,R extends Comp<?>>
        extends Comp<CmpCons<H,R>> { ... }
class Comp1<X0 extends Term<?>>
        extends CmpCons<X0,CmpNil> { ... }
class Comp2<X0 extends Term<?>, X1 extends Term<?>>
        extends CmpCons<X0,CmpCons<X1,CmpNil>> { ... }
```

Figure 5.8: Compound terms in PatJ

This signatures expresses that `list` is actually a term containing any list, while `size` is an integer term; moreover both terms can can be either inputs or outputs — a wildcard of the kind '`? extends`' is used for it allows covariance of the argument type, so that e.g. a `Term<List<Int>>` could be passed [JGSB05, VR05, IV06].

The hierarchy of terms is completed by dealing with compound terms through classes `Comp<X>`, `CmpNil` and `CmpCons<H,R>` as shown in Figure 5.8; a compound term is basically a tuple of terms of any length, e.g., it can have arity two and orderly contain a `List<Int>` and an `Atom` as in a compound term of the kind `p([1,2],'a')`. Hence, PatJ provides a list-like construction mechanism for the type parameter `X`, through classes `CmpCons` and `CmpNil`, as in the following case:

```
CmpCons<List<Int>,CmpCons<Atom,CmpNil>> c= ... ;
List<Int> first = c.head; //OK!!
Number second = c.rest.head; //ASSIGNMENT ERROR!!
```

Variable `c` is declared to be a compound term with two arguments of type `List<Int>` and `Atom`, hence assignment to `second` can be statically rejected. Classes `Comp1`, `Comp2` (and so on) are introduced as a syntactic facility for expressing compound types with 1 and 2 arguments. For instance, type `Comp2<List<Int>,Atom>` is a subtype of `CmpCons<List<Int>,CmpCons<Atom,CmpNil>>`, and can therefore be used in place of it.

As the hierarchy of terms is defined, PatJ defines two methods — `fromJava()` and `toJava()`, respectively — for translating term represen-

```
<Z> Collection<Z> toJava() {
   ArrayList<Z> javaList = new ArrayList<Z>(items.size());
   for (Term<?> t : items) {
      javaList.add((Z)t.toJava());
   }
   return javaList;
}
```

(a) From PatJ list to Java collection: `List.toJava()`

```
static <X extends Term<?>, Z> List<Z> fromJava(Collection<Z> c) {
   ArrayList<X> items = new ArrayList<X>(c.size());
   for (Z elem : c) {
      items.add(Term.<X>fromJava(elem));
   }
   return new List<X>(items);
}
```

(b) From Java collection to PatJ list: `List.fromJava()`

Figure 5.9: From PatJ list to Java collections and back

tation (they roughly correspond to the abstract marshalling/unmarshalling functions m and u defined in Section 1.4). The former method is used to translate a plain Java object (see Figure 5.9b) — mainly Java collections and primitive Java types — into terms of the PatJ hierarchy, the latter converts a term back to a standard Java representation (see Figure 5.9a).

Both such methods make use of type inference, avoiding the redundant specification of type `Z` — an actual instantiation for type parameter `Z` might be avoided, as it is typically inferred from the enclosing assignment context [JGSB05] — that is, by looking to the type of the variable to which the returned object is assigned (see Section 2.1). For instance, PatJ terms are simply turned into a suitable Java representation as follows[2]:

```
ArrayList<Integer> a = Arrays.toList(new Integer[]{1,2,3});
List<Int> term = Term.fromJava(a);
...
Collection<Integer> c = term.toJava();
```

[2]Note that, since runtime generic types are not currently supported in Java (see Section 3.2), we cannot intercept the case where the variable storing the return value has an incompatible type: this error would possibly lead to a later `ClassCastException` due to the so-called "heap pollution" problem [JGSB05].

```
tuprolog.Struct marshal() {
   tuprolog.Term[] termArray = tuprolog.Term[items.size()];
   int i=0;
   for (Term<?> t : items) {
      termArray[i++] = t.marshal();
   }
   return new tuprolog.Struct(termArray);
}
```

(a) From PatJ list to tuProlog `Struct`: `List.marshal()`

```
static <Z extends Term<?>> List<Z> unmarshal(tuprolog.Struct s) {
   Iterator<tuprolog.Term> list_it = s.listIterator();
   ArrayList<Term<?>> items = new ArrayList<Term<?>>();
   while (list_it.hasNext()) {
      termList.add(Term.unmarshal(listIt.next()));
   }
   return new List<Z>(items);
}
```

(b) From tuProlog `Struct` to PatJ list: `List.unmarshal()`

Figure 5.10: From PatJ list to tuProlog `Struct` and back

Finally, PatJ term classes define two methods — `marshal()` and `unmarshal()`, respectively — which are the key mechanism for switching from PatJ terms to tuProlog terms and vice-versa. The former method is used to convert a PatJ term into a tuProlog term (see Figure 5.10a); dually, the latter method converts back a tuProlog term into a suitable PatJ representation (see Figure 5.10b). These methods play a crucial role in the PatJ framework: in fact, any declarative feature provided by PatJ — such as Prolog methods (see Section 2.2) and the PatJ library (see Section 3) — is implemented in term of requests to an underlying tuProlog engine.

2.2 Prolog Classes and Methods

PatJ defines some custom Java annotations [JGSB05, Mica] for explicitly marking classes and methods as being associated to some Prolog code. A *Prolog class* is an abstract class/interface annotated with the `@PrologClass` annotation. Prolog classes are never instantiated directly (e.g. using the Java `new` operator); rather they are instantiated using the PatJ factory method

```
@PrologClass (
   clauses = {"remove([X|Xs],X,Xs).",
              "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys)."})
public abstract class PermutationUtility {

  @PrologMethod (
     clauses={"permutation([],[]).",
              "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs), permutation(Zs,Ys)."})
  public abstract <$X extends List<Int>,
                   $Y extends List<Int>>
                   Iterable<$Y> permutations($X list);

  public static void main(String[] args) {
    PermutationUtility pu = PJ.newInstance(PermutationUtility.class);
    java.util.Collection<Integer> l =
          java.util.Arrays.<Integer>asList(new Integer[]{1,2,3});
    for (List<Int> p : pu.permutations(Term.fromJava(l))) {
      System.out.println(p.toJava());
} } }
```

Figure 5.11: Permutations in PatJ

`newInstance()`. More importantly, Prolog classes can define one or more *Prolog methods*. A Prolog method is an abstract Java method annotated with the `@PrologMethod` annotation, that can be used to specify some Prolog code for the method implementation.

A mapping between a Prolog method `m()` and a predicate of the kind $\texttt{p}(\overline{\texttt{t}})$ is fully specified when the following elements are identified: the name of the predicate `p` to be associated with the method call; the arity of `p` — namely, the number of terms in $\overline{\texttt{t}}$; the input/output role of each term in $\overline{\texttt{t}}$; how each term in $\overline{\texttt{t}}$ is mapped into the signature of `m()` (e.g. a term could be mapped into one of the arguments of `m()`); finally, the set of Java types $\overline{\texttt{T}}$ associated with each term in $\overline{\texttt{t}}$.

PatJ recovers all such information from the *signature* of a Prolog method. More formally the signature of a Prolog method can be described as follows:

$$\texttt{<}\overline{\texttt{X}}\texttt{ extends }\overline{\texttt{B}}\texttt{> Tr}^{\overline{\texttt{X}}}\texttt{ m(}\overline{\texttt{Ta}}^{\overline{\texttt{X}}}\texttt{)}$$

where overlines are used to express lists of elements as in [IPW99]. Hence $\overline{\texttt{X}}$

are the method type-variables, $\overline{\texttt{B}}$ their bounds (which should be PatJ term types), $\texttt{Tr}^{\overline{\texttt{X}}}$ is the return type (a PatJ term type possibly constructed from type-variables in $\overline{\texttt{X}}$), $\overline{\texttt{Ta}}^{\overline{\texttt{X}}}$ are the formal argument types — each type in $\overline{\texttt{Ta}}^{\overline{\texttt{X}}}$ is a PatJ term type possibly constructed from type-variables in $\overline{\texttt{X}}$. More precisely, each type-variable in $\overline{\texttt{X}}$ can occur either as one of the argument types of `m()` (input type-variable), as component of the return type of `m()` (output type-variable), or in both places (input/output type-variable).

For a method of this kind, the following Java-Prolog mapping is defined:

- The name of the method `m()` should coincide with the predicate name `p` to be used;
- Each type-variable in $\overline{\texttt{X}}$ whose name starts with the special character `$`, corresponds to a logic argument of the template predicate `p`; consequently, the arity of `p` is equal to the number of such type-variables;
- The role associated with a type-variable of the kind `$X` can be either input or output, depending on whether `$X` occurs in argument position or in the return type of `m()`. Moreover, such return type is a subtype of `Iterable`, then `m()` is implicitly assumed to yield multiple results;
- Each Prolog term in $\overline{\texttt{t}}$ is associated with a Java type — namely one of the bound types $\overline{\texttt{B}}$ in the generic method declaration.

In Figure 5.11 the code of `PermutationUtility` is shown; `PermutationUtility` is a utility class for retrieving all permutations of a given Java list that leverages some of the features of the PatJ framework. The `@PrologClass` annotation might provide additional Prolog clauses to be used in Prolog class body: in this case we included the rules for predicate `remove/3` as it might be seen as a library predicate. Optionally, one can specify an external document as containing the theory to be used.

In this class an abstract method, namely `permutation()`, is defined with the signature one would use in an object-oriented context to get all the permutations of a given list. Its `@PrologMethod` annotation is used to specify the intended behaviour in terms of Prolog code — its `clauses` attribute defines the Prolog implementation of the `permutation/2` predicate. Thanks

to the mapping discussed above, PatJ assumes that the name of the template predicate `p` is `permutation`, the arity of `p` is equal to 2 and two type-variables, namely `$X` and `$Y`, are used to represent the first and second arguments of `p`, respectively; `$X` is an input type-variable while `$Y` is an output type-variable (and all its results will be considered by iteration); finally, the type `List<Int>` is associated with both terms in `permutation/2`.

Hence, a method call of the kind `permutation(l)`, where `l` is a PatJ term representing a Prolog list, can be mapped to a goal term of the kind `permutation(`m`(l),Y)`.

Inside the method `main()`, an instance of the Prolog class is created exploiting the `newInstance()` factory method provided by the PatJ framework (see Figure 5.12b). Exploiting the proxy technique, this method dynamically wraps the Prolog class passed as argument and returns the proxy object `pu` to the user. The code for the proxy class — namely, `PermutationUtility$Proxy` — is shown in Figure 5.12a; the proxy class defines some synthetic fields for *(i)* the Prolog theory associated with the underlying Prolog class, *(ii)* the Prolog theory associated with the Prolog method `permutation()`, *(iii)* a reflective object (of type `java.lang.reflect.Method`) representing the corresponding Prolog method and *(iv)* a proxy handler class (of type `MethodHandler`) — the entry point of the PatJ framework. When the Prolog method `permutation()` is invoked, a suitable `Theory` object is first retrieved — in this case, the resulting `Theory` object is the theory obtained by merging the class and method theories, orderly. Assuming that the method parameter `list` is a Prolog list of the kind `[1,2,3]`, a goal term of the kind `permutation([1,2,3],Y)` is then constructed. The Prolog method call is then dispatched to the PatJ framework which, in turn, triggers the resolution of a Prolog query to an underlying tuProlog engine and yields an iterator over all instances of `Y` — which are valid permutations of `list`. The logic of the iterator object, shown in Figure 5.12b, is similar to the one discussed in Section 1.4.

In general, the Java programmer may appreciate the high-level of specification for the method behaviour, and the simplicity in coding the client code. Moreover thanks to type inference, method invocations are properly checked — the Java compiler can properly select the correct method implementation,

```
class PermutationUtility$Proxy extends PermutationUtility implements PrologObject {
   InvocationHandler _pj;
   tuprolog.Theory _theory$class =
                new tuprolog.Theory("remove([X|Xs],X,Xs).\n" +
                                    "remove([X|Xs],E,[X|Ys]):- remove(Xs,E,Ys).\n");

   tuprolog.Theory _theory$permutation =
                new tuprolog.Theory("permutation([],[]).\n" +
                                    "permutation(Xs,[X|Ys]):- any(Xs,X,Zs)," +
                                    "permutation(Zs,Ys).\n"});

   java.lang.reflect.Method _method$permutation = ...

   <$X extends List<Int>,$Y extends List<Int>> Iterable<$Y> permutation($X l) {
      Theory _theory = _theory$class.append(_theory$permutation);
      Comp2<List<Int>,Var<List<Int>>> _goal = new Comp2("permutation",
                                                        l
                                                        new Var<List<Int>>("Y"));
      return (Iterable<$Y>)_pj.invoke(this,
                                      _method$permutation,
                                      new Object[] {_theory, _goal});
} }
```

(a) The PatJ proxy class

```
class PJ implements InvocationHandler {
   ...
   PrologObject newInstance(Class<?> _class) {
      (PrologObject)Proxy.newProxyInstance(PJ.class.getClassLoader(),
                                           new Class[] { PrologObject.class, _class },
                                           this);
   }

   public Object invoke(Object proxy, Method method, Object[] args) {
      tuprolog.Theory _theory = (Theory)args[0];
      final Term<?> _goal = (Term<?>)args[1];
      final Prolog _engine = getEngine();
      SolveInfo _firstSolution = _engine.solve(_goal.marshal());
      if (!_engine.hasOpenAlternatives()) {
         return PJ.unmarshal(_firstSolution.getTerm());
      }
      else {
         return new Iterator<Term<?>>() {
            boolean _backtrack = false;
            public boolean hasNext() {
               _engine.hasOpenAlternatives();
            }
            public Term<?> next() {
               SolveInfo _next = null;
               if (!_backtrack) {
                  _backtrack = true;
                  _next = _firstSolution;
               }
               else {
                  _next = _engine.solveNext(_goal.marshal());
               }
               return PJ.unmarshal(_next.getTerm());
            }
            ...
         };
} } }
```

(b) The PJ class

Figure 5.12: Permutations in PatJ: dynamic proxy class and Prolog method handler

and then establish whether a Prolog method invocation is correct. Recalling the example above, `permutation()` should be supplied an argument that is a subtype of `List<Int>`, while the return type of permutation can be assigned to a variable whose type is compatible with `Iterable<List<Int>>`, or it can be used directly into a for-each loop to iteratively get lists of integers.

2.2.1 Benefits of Generics and Type Inference

As generics were introduced in J2SE 5.0, the gap between required skills of API developers and API users seriously increased. The design of the Java Collections Framework, for instance, heavily relies on generics, wildcards, and type inference; on the other hand, users of this API may know very little about such concepts — typically they just create generic collections and then call methods defined by the API: the Java compiler is in charge of checking for an incorrect usages of generic types. For instance, the declaration of method `Collections.sort()` might appear overly complex at first:

```
static <T extends Comparable<? super T>>
                          void sort(List<T> list)
```

On the other hand `sort()` can be simply used as follows:

```
ArrayList<Number> l=new ArrayList<Number>();
l.add(3); l.add(2); l.add(1);
Collections.sort(l);
```

Type inference in method calls takes care of finding a proper instantiation of the method type parameters, and accordingly checks the validity of the invocation — it infers e.g. `Number` for `T`, and it checks that `Number` is a subtype of `Comparable<? super Number>` (see Section 2.1).

By relying on the expressiveness of the generic type-system, PatJ follows a similar approach. Representing Prolog predicates in terms of Java generic methods provides two significant advantages: first, it makes it possible to define expressive constraints on the types of Prolog terms in a given predicate; secondly, it allows for concise syntax at the call-site, by leveraging Java support for generic method type inference.

Suppose that we want to define a variant of the above Prolog method `permutation()`, where the new method — namely `permutation2()` — should

accept a list of some *unknown* type `E` and returning lists of the *same* unknown type `E` that are also permutations of the input list. In other words we are seeking for a way to express a type-constraint — the type of the elements in the input list and the type of the elements of the lists returned by the Prolog method `permutation2` must match. In PatJ, such constraints are easily expressed through additional type-variables attached to the signature of a Prolog method; such type-variables can be used to abstract over a concrete PatJ term type, as shown below:

```
@PrologMethod (  clauses= ... )
<E extends Term<?>,
 $X extends List<E>,
 $Y extends List<E>> Iterable<$Y> permutation2($X l);
```

Here, the type-variable `E` is not treated as a logical argument of predicate `permutation/2` — its name does not contain the special '`$`' character. On the other hand, the compiler can use this variable for checking any further constraint by method type inference: in this case, the compiler checks that both `X` and `Y` are indeed Prolog list of the kind `List<E>` — where the `E` is some concrete PatJ term type, such as `Int`, `Atom`, etc. The combined exploitation of additional method type-variables and bounded polymorphism greatly enriches the expressiveness of the PatJ framework — the reader might appreciate how the conciseness of this approach is comparable e.g. to the one in [Esp06], where a true extension of the Java programming language enabling declarative features is exploited.

Moreover, it can be noticed that, although the above Prolog method declaration involves some tricky aspect of generics, its exploitation is instead rather simple:

```
List<Atom> la = ... ;
for (List<Atom> p : permutation2(la))  ...
for (List<Int> p : permutation2(la))  ...  //error!
```

In the first case, the compiler infers that the type-variable `E` of `permutation2()` should have type `Atom`. Consequently, the two remaining type-variables `X` and `Y` should both have type `[E/Atom]List<E>` = `List<Atom>`. Similarly, in the second case the compiler infers that `E` has

```
new_object(ClassName,Args,Id):-prolog_class(ClassName),
                               pj_proxy_object(ClassName,Args,Id).
new_object(ClassName,Args,Id):- !, java_object(ClassName,Args,Id).

Obj <- What :- java_call1(Obj,What,Res),
               Res \== false.
Obj <- What returns Res :- java_call1(Obj,What,Res).

java_call1(Obj,What,Res):-unmarshal_method(What, M2),
                          lookup_method(Obj, M2, Meth),
                          prolog_method(Meth),!,
                          pj_call_rest(Obj, Meth, M2, Res).
java_call1(Obj,What,Res):-lookup_method(Obj,What,Meth),
                          java_method_call(Obj,Meth,What,Res).

pj_call_rest(Obj,Meth,What,Res):-is_iterable(Meth), !,unmarshal
                                 java_method_call(Obj,Meth,What,R2),
                                 R2 <- iterator returns I,
                                 next(I, E), marshal(E, Res).
pj_call_rest(Obj,Meth,What,Res):-!,java_method_call(Obj,Meth,What,R2),
                                 marshal(R2, Res).

unmarshal(L1, L2):-'Term' <- marshal(L1) returns L2.

marshal(L1, L2):-L1 <- marshal() returns L2.
```

Figure 5.13: The PatJ library

type `Atom` — this is done by looking at the type of the actual argument supplied to `permutation2()`; after type-variable substitution we have that the method return type is thus inferred as `List<Atom>`, which is incompatible with the type of the variable `p` in the for loop. Hence, in PatJ, library developers can either partially or fully rely on the power of method type inference: a careful design fully adopting type-inference enhances type-safety in clients of a PatJ library and frees them from most of the burden associated with static typing.

3 Java from Prolog: the PatJ Library

PatJ supports another form of interoperability that has not been discussed so far: the ability of calling Java code from Prolog methods; suppose that a Prolog method needs to perform a complex operation requiring one ore more Java libraries, such as interacting with a Java GUI, or accessing a

```
public boolean pj_proxy_object_3(Term className, Term args, Term id) {
   if (!className.isAtom() && !args.isEmptyList())
      return false;
   Class<?> clazz = Class.forName(((Struct)className.getTerm()).getName());
   PrologObject po = PJ.newInstance(clazz);
   return JavaLibrary.getObjectReference(id, po);
}
```

Figure 5.14: Implementation `pj_proxy_object`

remote object via RMI. In tuProlog, any Java component can be directly accessed and used from Prolog, in a simple and effective way, by means of the *JavaLibrary* library [DOR05, tuP02]: this delivers all the power of existing Java components and packages to Prolog sources. The PatJ framework extends the basic functionalities provided by the JavaLibrary, by defining additional Prolog predicates that can be used to e.g. create an instance of a Prolog class, or to invoke a Prolog method, directly from Prolog.

3.1 Creating Objects

The PatJ library predicate `java_object/3` is used to create a new Java object of the specified class, according to the syntax (see Figure 5.13):

```
java_object(ClassName, Arguments, ObjectRef)
```

where `ClassName` is a Prolog atom bound to the name of the proper Java class (e.g. '`java.util.Vector`') — the class denoted by `ClassName` could be either a standard Java class (`is_Prolog_class(ClassName)` yields `false`) or a class annotated with the `@PrologClass` annotation (`is_Prolog_class(ClassName)` yields `true`); the parameter `Arguments` is a Prolog list used to supply the required arguments to the class constructor — the empty list denotes the default constructor; finally, the reference to the newly-created object is bound to `ObjectRef`: if the term associated with `ObjectRef` is a logic variable of the kind `X`, a new object is allocated and `X` is bound to the corresponding *object reference* — a unique identifier that is automatically generated by the JavaLibrary. For instance, the following Prolog code is used to create a new object of type `java.util.Vector`:

```
create_vector(X):-java_object('java.util.Vector', [10], X).
```

In the case above, since the name of the class (`'java.util.Vector'`) denotes a standard Java class, no special treatment is required and the semantics of the `java_object` predicate falls back to the basic semantics provided by the tuProlog's JavaLibrary (`java_object` predicate in Figure 5.13). Conversely, Prolog classes must be handled differently; Prolog classes are usually abstract Java classes (or interfaces) that lack a proper constructor; consequently, the instantiation of Prolog classes should take place by invoking the static factory method provided by the PatJ framework (see implementation of the `pj_proxy_object` predicate in Figure 5.14), rather than exploiting standard reflective features — e.g `Class.newInstance()`. More specifically, given a goal of the kind `java_object(ClassName,Args,Obj)`, where `ClassName` denotes a class annotated with the `@PrologClass` annotation, the PatJ framework dynamically creates a new proxy object; such object is then associated with an object reference and bound to `X` the usual way. The following code is used to create a new instance of the Prolog class `PermutationUtility`:

```
permutation_utility(P):-new_object('PermutationUtility', [], P).
```

Since `PermutationUtility` is a class annotated with the `@PrologClass` annotation - `new_object` will bind a new dynamic proxy (obtained calling `PJ.newInstance`) to the Prolog variable `P`.

3.2 Calling Methods

An object reference can be used as a receiver in a method call expression; the PatJ predicate '`<-`' is used to invoke a method on a Java object using the following syntax (see Figure 5.13):

```
ObjectRef <- MethodName(Arguments)
```

where `ObjectRef` is the receiver object — an atom interpreted as a Java object reference as explained above; `MethodName` is the name of the Java method to be invoked; finally, the parameter `Arguments` denotes the actual arguments to be supplied in a given method call. Note that the method name and the runtime types of the supplied arguments must be used to perform a dynamic overload resolution process (see predicate `lookup_method(Obj, M2, Meth)` in Figure 5.13), as the receiver class could define more than one matching

method; this resolution process bounds the logic variable `Meth` to an object reference of the kind `java.lang.reflect.Method` — the reflective object associated with the method to be invoked; again, the method denoted by `Meth` could be either a standard Java method (`is_Prolog_method(MethodName)` yields `false`) or a method annotated with the `@PrologMethod` annotation (`is_Prolog_method(MethodName)` yields `true`). The following example adds three elements to an object of type `java.util.Vector`, by repeatedly calling its `add()` method:

```
init_vector(V):-create_vector(V),
                add_el(V,1), add_el(V,2), add_el(V,3).
add_el(V, E):-V <- add(E).
```

In order to model method calls with a return value, the following syntax is used instead:

```
ObjectRef <- MethodName (Arguments ) returns Term
```

Here, the `returns` keyword is used to retrieve the value returned from non-void Java methods and to bind it to a Prolog term; there are two possible cases: if the type of the return value can be mapped onto a primitive Prolog data-type (e.g. a number or a string), `Term` is unified with the corresponding Prolog term; conversely, if the return value is a Java object other than the ones above, `Term` is bound to a new object reference. The following code is used to increment the $\texttt{I}_{\texttt{th}}$ element stored in a collection denoted by `V` — the `returns` predicate provides a way to denote the value returned by `get()`:

```
inc_vector(V, I):- V <- get(I) returns E1,
                   E2 is E1 + 1,
                   V <- set(E2, I).
```

In the case above, since the method to be invoked (e.g. `Vector.add()`) is a standard Java method, no special treatment is required and the semantics of the '`<-`' predicate falls back to the standard behaviour provided by the tuProlog's JavaLibrary (`java_method_call` predicate in Figure 5.13).

Conversely, Prolog method calls must be handled differently, as they typically involves two symmetric conversions: first the method arguments

must be converted into a suitable PatJ representation, so that the resolution process can effectively take place; secondly, the Prolog method return value must be converted back into a suitable tuProlog representation — such conversions are performed by the special `unmarshal` and `marshal` predicates given in Figure 5.13. The `unmarshal` predicate converts a tuProlog term into a PatJ term by calling the static method `unmarshal()` defined by the PatJ class `Term`; dually, a PatJ term is converted back into a tuProlog term by calling the method `marshal()` — each PatJ term class overrides this method so that the most suitable tuProlog representation is returned (see Section 2.1).

The PatJ library must also model the behaviour of an `Iterable` return type in terms of standard Prolog backtracking; this is accomplished by leveraging the PatJ predicate `pj_call_rest`, which iteratively binds a logic variable to all the elements associated with a Java iterator. In other words, all the complexity of interfacing with a Prolog method is hidden by the PatJ library — for instance the following code is used to invoke the Prolog method `permutation()`:

```
call_permutation(L, R):-permutation_utility(P),
                        P <- permutation(L) returns R.
```

Note that there is no need to perform explicit marshalling/unmarshalling of method parameters and to handle the `Iterable` result returned by the Prolog method `permutation()`; all these tasks are performed automatically by the PatJ library by overriding the semantics of the tuProlog '`<-`' operator.

4 Advanced Features

In this section we discuss some more sophisticated aspects of the PatJ framework such as static type-checking of PatJ annotations, stateful Prolog objects embedding an instance theory and Prolog fields, and support for custom data-types.

First, the contents of PatJ annotations can be checked statically by means of a *custom annotation processor*; this way, the set of static checks carried out by the Java compiler can be smoothly extended so that the compiler will e.g.

```
@Target(ElementType.METHOD)
public @interface PrologMethod {
   String[] clauses() default {};
   String predicate() default "";
   String signature() default "";
   String[] types() default {};
}
```

Figure 5.15: The `@PrologMethod` annotation

issue error messages if the declaration of a Prolog method does not match its abstract specification (given in its `@PrologMethod` annotation).

Secondly, as it is useful to define the implementation of a Java method in terms of declarative facts and rules, it is also important to be able to represent the state of an object declaratively; the PatJ framework provides support for *instance theories* — Prolog theories attached to Prolog objects that can be dynamically accessed and/or updated; moreover, PatJ provides special support for Java fields whose type is a PatJ term — we call such fields *Prolog fields*.

Finally, the PatJ framework support ad-hoc marshalling/unmarshalling of custom data-types; that is, a Prolog method can be supplied a Java object for which no default mapping exists: in that case the PatJ framework handles the call either by wrapping the Java object into an object reference (as discussed in Section 3) or by *transforming* it into a Prolog compound representation, so that the contents of the object can be more naturally accessed from Prolog code.

4.1 Checking Prolog Methods

Recalling from section 2.2, the mapping between a Prolog method `m()` and a predicate of the form `p(`$\overline{\texttt{t}}$`)` is fully specified when the following elements are provided: *(i)* the *name* and the *arity* of the predicate, *(ii)* the "logical" role of each term in $\overline{\texttt{t}}$, *(iii)* how each term in $\overline{\texttt{t}}$ is mapped into the signature of `m()` and *(iv)* the Java types that can be associated to each term in $\overline{\texttt{t}}$.

The `@PrologMethod` annotation defines some additional attributes that are used to explicitly define each of the properties above. Altough the use of

these attributes is optional — the simplest way to define a mapping between a Prolog method and its corresponding predicate is to define a generic method, as described in Section 2.2 — such attributes can be fruitfully exploited in order to make the mapping between a Prolog method and a Prolog predicate more explicit; an important consequence (other than improving readability) is that this additional information can be made available to a type-checker that can thus verify the well-formedness of a given Prolog method signature.

Here is a complete list of attributes that can be attached to a `@PrologMethod` annotation (see Figure 5.15):

predicate denotes the Prolog predicate that should be used for building the goal term — a term-like notation that is used to keep track of the name and the arity of the predicate, as well as naming each argument: a possible value for it is e.g. `foo(X,Y,Z)`. Moreover, each argument can be attached an ISO Prolog notation, providing constraints on the roles of the terms to be passed. It can be either '`+`' (input term, i.e., not a variable), '`-`' (output term, i.e., a variable), '`?`' (either input or output), and '`@`' (a *ground* input term, i.e., a term with no variables inside); annotations '`-@`' and '`?@`' are added to model ground output and ground input/output — they are not part of the ISO standard, but they can be usefully expressed in PatJ. For instance, the template predicate could be '`foo(@X,+Y,-Z)`'.

signature specifies the mapping between the arguments in the `predicate` attribute and the position in the signature of the Prolog method, that is, how they correspond to the Prolog method's argument or return type. A possible value for `signature` is e.g. '`(X,Y)->(Z)`', stating that the the first argument of the Prolog method should map to the term `X`, the second to `Y`, and that the return type maps to `Z`, respectively. A signature of the kind '`(X,Y)->{Z}`' is used when the method should actually return an iterator over all possible results for `Z`, while a signature of the kind '`(X,Y)->(Z,X)`' is used when the return must be a 2-ary compound term including the result for `Z` and `X`, orderly. Figure 5.1a shows how a Prolog method `permutation()`, whose `types` attributes

`signature`	*Prolog method signature*
`(X)->(X,Y)`	`Comp2<List<Int>,List<Int>> permutation(List<Int> l)`
`(X)->(Y)`	`List<Int> permutation(List<Int> l)`
`(X,Y)->()`	`boolean permutation(List<Int> l, Var<List<Int>> x)`
`(X)->{Y}`	`Iterable<List<Int>> permutation(List<Int> l)`

(a) The `signature` attribute

i_{th} term	*Type*
`@X`	`List<Int>`
`+X`	`List<? extends Term<Int>>`
`-X`	`Var<? extends List<? extends Term<Int>>>`
`?X`	`Term<? extends List<? extends Term<Int>>>`
`-@X`	`Var<List<Int>>`
`?@X`	`Term<? extends List<Int>>`

(b) The `types` attribute: '`-`' = input, '`+`' = output, '`?`' = input/output, '`@`' ground

Table 5.1: Overview of the attributes of the `@PrologMethod` annotation

is {`List<Int>,List<Int>`}, can induce different signatures depending on the value of the `signature` attribute.

types specifies a mapping between each of the arguments listed in the `predicate` attribute and a Java type in the PatJ term hierarchy. Such an attribute could e.g. be {`Atom,Int,List<Comp2<Atom,Atom>>`}, stating that `X`, `Y` and `Z` are associated with the PatJ term types `Atom`, `Int`, and `List<Comp2<Atom,Atom>>`, respectively. Note that the actual type reported in the Prolog method signature is a refined version of the one specified in the `types` attribute. Hence, a type listed in the `types` attribute can be viewed as an abstract specification of the Java type associated with a given predicate variable; such type must be instantiated accordingly, depending on the value of the ISO annotation `+-?@` specified in the `predicate` attribute. Figure 5.1b shows how an argument of type `List<Int>` is turned into a Java type depending on the Prolog annotation expressed in the `predicate` attribute. For instance, a Prolog term of the kind `?X` is associated with the Java type `Term<? extends List<? extends Term<Int>>>`. In fact, `X` could be

```
@PrologMethod (
     predicate="permutation(@X,-@Y)",
     signature="(X)->{Y}",
     types={"List<Int>","List<Int>"},
     clauses={"remove([X|Xs],X,Xs).",
              "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
              "permutation([],[]).",
              "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs),
                                       permutation(Zs,Ys)."})
  public abstract Iterable<List<Int>> permutation(List<Int> list);
```

Figure 5.16: An alternate declaration of `permutation()`

either an input or output term, so that either a ground term (e.g. of type `List<Int>`) or a variable (e.g. of type `Var<List<Int>>`) could be supplied; consequently, the type used in the mapping should be of the kind `Term<...>` as discussed in Section 2.1. Moreover, the input list can contain variables as in a Prolog term of the kind `[_,_,_]`; hence any subtype of `Term<Int>` might be used as parameter of `List<...>`. Finally, the outermost wildcard `Term<? extends ...>` is used to make the whole type covariant (see the covariance propagation rule in [IV06]), e.g., to make instances of `Term<List<Int>>` be compatible with it[3].

An alternate declaration for the Prolog method `permutation()` discussed in Section 2.2 is reported in Figure 5.16. The `predicate` attribute is set to `permutation(@X,-@Y)`: `X` is the input list and it should be ground, while `Y` is an output term — a logic variable eventually bound to a ground term. The `signature` attribute is set to `(X)->{Y}`: `X` is the only argument of the method, while `Y` is the return value of the method, accessed through an iterator. Finally, the `types` attribute is set to {`List<Int>`,`List<Int>`}: both `X` and `Y` are terms of the kind `List<Int>`.

PatJ is equipped with a custom annotation processor that can be used to check the above annotation attributes without the need of building an actual

[3]Should the programmer be concerned about the complexity of such generic types, we observe that by our approach the compiler will enforce their correctness and suggest the correct type in case of mistakes, as discussed in the following.

compiler extension — causing obvious deployment issues. This feature is built on top of the JSR 269 support that has been introduced in JSE 6 [Mic05], allowing subclasses of the `javax.annotation.processing.Processor` class [Mica] to define custom annotation processors that can be passed to the Java compiler[4]. Assuming that the PatJ jarfile is in the classpath, the PatJ annotation processor is automatically detected by the Java compiler and used whenever a source file containing custom PatJ annotations is found.

The PatJ annotation processor verifies that the code of a Prolog class/method is compliant with the PatJ framework; most importantly, it checks the well-formedness of a Prolog method signature against the ISO notations specified by its `predicate` attribute; the compiler is actually able to infer (and report to the user) the correct signature to be used in a Prolog method declaration — this is accomplished by inspecting the attributes of the `@PrologMethod` annotation associated with a given Prolog method declaration. As the PatJ term hierarchy and annotation library are rather rich, the possibility of checking for the well-formedness of a Prolog class definition turns out to be crucial in making PatJ a usable tool.

4.2 Coding State: Prolog Fields and Instance Theories

Sometimes it is useful to model the state of a Java object in a declarative way, as a logic theory that can be dynamically be accessed and updated if needed. The PatJ framework provides two different techniques for encoding the state of a Prolog class in terms of Prolog clauses; first, a Prolog class can define one or more *Prolog fields* that can be accessed and updated either from Prolog or Java; secondly an instance-specific theory can be passed on to a Prolog object during its initialisation — all Prolog methods declared in that class can dynamically update the contents of the instance theory through standard Prolog assertion/retraction features.

4.2.1 Prolog Fields

A Java field whose type is a PatJ term type can be annotated with the `@PrologField` annotation — we call such field a Prolog field. The

[4]In the `javac` compiler this is accomplished using the `-proc` option.

```
@PrologClass
public abstract class Maze {

    @PrologField(init="node(start)", predicate="current_site")
    public Comp1<Atom> currentSite;

    @PrologField(init="node(exit)")
    public Comp1<Atom> exit;

    @PrologMethod (
            clauses = {"path(X,Y):-door(X,Y).",
                       "path(X,Y):-door(Y,X).",
                       "reachable_sites(X):-current_site(C),
                                            path(C, X)."}
    )
    public abstract <$X extends Comp1<Atom>>
                    Iterable<$X> reachable_sites();
}
```

Figure 5.17: Maze using Prolog fields

`@PrologField` annotation may optionally define a Prolog initializer (`init` attribute), specifying the initial value for that Prolog field; this annotation is also used to specify how the Prolog field should be mapped into a Prolog predicate (`predicate` attribute). For instance, the following code is used to declare a Prolog field storing a list of integer values:

```
@PrologField(init="[1, 2, 3]", predicate="p_list")
public List<Int> p_list;
```

As it can be seen, the value of the `init` attribute is a Prolog term of the kind `[1,2,3]`; as usual, the `@PrologField` annotation is checked by the PatJ annotation processor so that the compiler can check the well-formedness of the initialisation term against the declared type of the Prolog field.

There are two ways to access or update the contents of a Prolog field. From Prolog code, a Prolog field `f` can be accessed by exploiting the clause implicitly defined by `f`'s `predicate` attribute, while a new value is assigned to `f` using the ':=' operator provided by the PatJ library:

```
p_list := [42] //updates p_list
p_list(Y). //Y unifies with [42]
```

Each Prolog field implicitly defines a pair of getter/setters that can be used whenever the Prolog field must be accessed from a Prolog method defined in a different class — this is accomplished with a special usage of the PatJ '`->`' operator (see Section 3.2):

```
PO.p_list <- set(42) //updates p_list
PO.p_list <- get(Y). //Y unifies with [42]
```

Finally, since a Prolog field is also a Java field, it can be accessed and updated as usual from Java code:

```
Collection<Integer> coll = Arrays.asList(new int[] 42);
p_list = new List(coll);
System.out.println(p_list); //prints '[42]'
```

A special read-only Prolog field `this` is implicitly added to each Prolog class. This field holds the reference to the Prolog instance being the receiver of a given Prolog method call.

In Figure 5.17 is shown a simple PatJ class representing a maze. This class exploits two Prolog fields — `currentSite` and `exit` of type `Comp1<Atom>` — used to to store the current and the exit node, respectively — compound terms of the kind '`node(...)`'. The Prolog method `reachable_sites()` is used to retrieve the set of nodes that are reachable from the current node in `currentSite`. Note that, apart from the start/exit node — Prolog terms of the kind `node(start)` and `node(exit)`, respectively — the topology of the maze is here left unspecified; in the next section we discuss how an instance of a Prolog class can be parameterised with its own instance theory.

4.2.2 Instance Theories

The state of a Prolog class can be expressed in a declarative fashion, in terms of facts and rules; this is accomplished by associating a Prolog object with a so called *instance theory*, that must be supplied to the PatJ factory method. The clauses of this Prolog theory are available in all the Prolog methods declared in the Prolog object's class; hence, the instance theory can be dynamically accessed and/or updated from Prolog code using the PatJ variants of the standard Prolog `assert`/`retract` predicates.

```
@PrologClass
public abstract class PJBot {

    public Maze maze;

    @PrologMethod (
           clauses = {"explore:-this(Z), Z.maze <- get(M),
                                M.currentSite <- get(N),
                                explore_1(N, X).",
                      "explore_1(N, X):-this(Z), Z.maze <- get(M),
                                M.exit <- get(N), !.
                      "explore_1(N, X):-!, this(Z), Z.maze <- get(M),
                                M <- reachable_sites returns D,
                                not visited(D), M.currentSite <- set(D),
                                add_rule(visited(D)), explore_1(D,X)."}
    )
    public abstract void explore();

    @PrologMethod (
       clauses = {"visited_nodes(X):-visited(X)."}
    )
    public abstract <$X extends Comp1<Atom>> Iterable<$X> visited_nodes();

    public static void main(String[] args) throws Exception {
       String topology = "door(node(start),node(a)).\n" +
                         "door(node(a),node(g)).\n" +
                         "door(node(b),node(a)).\n"+
                         "door(node(a),node(d)).\n"+
                         "door(node(e),node(b)).\n"+
                         "door(node(g),node(h)).\n"+
                         "door(node(e),node(f)).\n"+
                         "door(node(f),node(i)).\n"+
                         "door(node(i),node(exit)).\n";
       Maze m = PJ.newInstance(Maze.class,new Theory(topology));
       PJBot b = PJ.newInstance(PJBot.class);
       b.maze = m;
       b.explore();
       for (Comp1<Atom> a : b.visited_nodes()) {
           System.out.println("[visited node = " + a.get0().toJava() + " ]");
       }
    }
}
```

Figure 5.18: Bot using private instance theory

There are three PatJ meta predicates that allows a programmer to manipulate the content of a given instance theory: `add_clause`, `remove_clause`, `remove_clauses`, whose semantics closely follows the Prolog standard meta-predicates `assert`, `retract` and `retractAll`, respectively. The predicate `add_clause` is used to add the clause passed as argument to the receiver's instance theory. Dually, the `remove_clause` predicate is used to remove the clause passed as argument from the receiver's instance theory. Finally the `remove_clauses` predicate is used to remove *all* clauses matching the clause

passed as argument from the receiver's instance theory. When an instance theory is updated using the above PatJ meta predicates, the changes will survive across multiple PatJ method calls — that is, the PatJ framework must intercept the execution of such meta predicates so that the instance theory associated with a given Prolog object can be updated accordingly.

A PatJ class named `PJBot` is shown in Figure 5.18; this Prolog class defines a Prolog method, namely `explore()`, that is used to traverse a maze in order to find the exit node (the maze implementation has been shown in Figure 5.17). The topology of the maze is stored in an instance theory, which is structured as a set of facts of the kind `door(X,Y)` where both `X` and `Y` are compound terms of the kind `node(...)` — meaning that the node `X` can be reached from `Y`, and vice-versa. In order to build an instance of the `Maze` class, the user must specify the topology of the maze — this is accomplished by passing a Prolog theory (a `Theory` object) to the PatJ factory method `newInstance()`, as shown in method `main()`.

The bot is equipped with its own instance theory; this theory (initially empty) is used to keep track of the previously explored nodes — this is needed as the maze topology might contain loops. For each new visited node of the kind `node(n)`, a new Prolog fact of the kind `visited(node(n))` is added to the bot instance theory — this is accomplished by leveraging the `add_clause` meta predicate. As the bot explores new nodes, it updates the current position in the maze; this is done by setting the value of the `Maze.currentSite` Prolog field. The exploration routine ends as soon as the bot finds the exit node — this is done simply by comparing the current node with the maze exit node `node(exit)`; at this stage the bot instance theory will contain several facts of the kind `visited(node(n))`, one for each node that has been visited during the exploration process.

4.3 Support for Custom Data-types

The PatJ framework supports two different translation schemes that allow an object of a custom class to be passed to a Prolog method: in the *call-by-reference* scheme, an object is wrapped in a PatJ object reference — an atom, as discussed in Section 3 — which is then passed to the Prolog method;

```
tuprolog.Struct marshal() {
   return JavaLibrary.getObjectReference(_object);
}
```

(a) From `JavaRef` to tuProlog `Struct`

```
static <Z> JavaObject<Z> Object(tuprolog.Struct s) {
   Z obj = (Z)JavaLibrary.dereference(s);
   return new JavaObject<Z>(obj);
}
```

(b) From tuProlog `Struct` to `JavaRef`

Figure 5.19: Marshalling/unmarshalling of `JavaRef` in PatJ

in the *call-by-value* scheme a Java object is turned into a full-fledged Prolog representation — typically a compound term — which can thus be accessed declaratively from Prolog code.

4.3.1 Call-by-reference

In the call-by-value scheme a Java object is turned into an object reference (see Section 3) that can be accessed from Prolog code by exploiting the PatJ library. The call by reference scheme is supported by means of a special PatJ term class — namely `JavaRef`:

```
public class JavaRef<O> extends Term<JavaRef<O>> {
   O _object
   ...
}
```

The type-variable `O` is used to abstract from the type of the Java object wrapped by a `JavaRef` instance — namely `_object`. The `JavaRef` class defines specialised marshalling/unmarshalling operations that leverage the PatJ library: each time a `JavaRef` term must be passed to a Prolog method, such term is converted into an object reference — this is accomplished by registering the object stored wrapped by a `JavaRef` in the PatJ library (see Figure 5.19a). Conversely, when a method returns a `JavaRef` term, the object associated with a given object reference is retrieved and then wrapped in a new `JavaRef` instance (see Figure 5.19b).

Java reference terms are simply built from the Java object that needs to be passed to a Prolog method; for instance, the following code creates a new `JavaRef` term storing a Java object of type `BigInteger`:

```
BigInteger bi = BigInteger.valueOf(100000000000);
JavaRef<BigInteger> rbi = new JavaRef<BigInteger>(bi);
```

Calling methods on a Java reference terms from the Prolog code is accomplished the usual way by means of the PatJ library — this is possible because a `JavaRef` term is internally converted into an ordinary object reference before a Prolog method is executed; for instance, the following Prolog method computes the amount of bits required in order to encode a `BigInteger` object:

```
@PrologMethod(
   clauses="bit_length(BI, L) :- BI -> bitLength() returns L.")
public abstract <$X extends JavaRef<BigInteger>,
                 $Y extends Int> $Y bit_length($X);
```

4.3.2 Call-by-value

In the call-by-value scheme a Java object is turned into a Prolog representation that can directly be leveraged from Prolog code, that is without the need of exploiting the PatJ library; hence, the call-by-value strategy is very useful when we want to pass a custom Java objects to a Prolog method, without loosing the ability to access the object contents declaratively. The call-by-value scheme is supported by means of a special PatJ term class — namely `JavaVal`:

```
public class JavaVal<O> extends Comp<Term<?>> {
   O _object
   ...
}
```

The type-variable `O` is used to abstract from the type of the Java object wrapped by a `JavaVal` instance — namely `_object`. The `JavaRef` class defines specialised marshalling/unmarshalling operations that leverage the `@Termifiable` annotation: each time a `JavaRef` term must be passed to a Prolog method, such term is converted into a Prolog compound term, as shown in Figure 5.20a — the details of this conversion are reported below.

```
tuprolog.Struct marshal() {
   Vector<Term<?>> termArr = new java.util.Vector<Term<?>>();
   BeanInfo binfo = Introspector.getBeanInfo(_object.getClass());
   int count = 0;
   for (PropertyDescriptor pdesc : binfo.getPropertyDescriptors()) {
      //only read-write properties are translated into a compound
      if (pdesc.getReadMethod() != null && pdesc.getWriteMethod() != null) {
         Object o = pdesc.getReadMethod().invoke(_object);
         Term<?> t = o != null ?
                     Term.fromJava(o) :
                     new Var("X" + count);
         termArr.add(t);
         count++;
      }
   }
   String functorName = _object.getClass.getAnnotation(Termifiable.class).predicate();
   PJ.termifiableRegistry.put(functorName, _object.getClass());
   return new tuprolog.Struct(functorName, termArr);
}
```

(a) From `JavaVal` to tuProlog `Struct`

```
static <Z> JavaTerm<Z> unmarshal(tuprolog.Struct s) {
   Class<?> _class = PJ.termifiableRegistry.get(s.getName());
   Z obj = (Z)_class.newInstance();
   BeanInfo binfo = Introspector.getBeanInfo(_class);
   int count = 0;
   for (PropertyDescriptor pdesc : binfo.getPropertyDescriptors()) {
      if (pdesc.getReadMethod() != null && pdesc.getWriteMethod() != null) {
         pdesc.getWriteMethod().invoke(po, Term.unmarshal(s.getTerm(count++)).toJava());
      }
   }
   return new JavaVal<Z>(obj);
}
```

(b) From tuProlog `Struct` to `JavaVal`

Figure 5.20: Marshalling/unmarshalling of `JavaVal` in PatJ

Conversely, when a method returns a `JavaVal` term, such compound term is converted back into a suitable Java representation that is then wrapped in a new `JavaVal`, as shown in Figure 5.20b.

When the PatJ framework needs to convert a Java object into a Prolog compound term, it does so by inspecting the contents of the `@Termifiable` annotation attached to the object's class; in fact, every object wrapped in a `JavaVal` term must be annotated with the `@Termifiable` annotation. This annotation specifies how the class should be mapped into a Prolog compound term; more specifically, its `predicate` attribute defines the name of the Prolog compound term that should be used to encode an instance of a given termifiable class. The PatJ framework keeps track of the functor name

associated with a given termifiable class by exploiting a shared *registry* (see Figure 5.20a) — each time a new termifiable class is marshalled, the registry is updated with a new entry, which is then used (during unmarshalling) to lookup the Java Class object associated with a given functor name. Note also that the arity of the compound term associated with a termifiable class `C` is given by the number of the *public* fields declared in `C`; moreover,the fields that must be included in the corresponding Prolog representation should have getter/setter methods — this is required, since the marshalling/unmarshalling routines defined by `JavaVal` rely upon *JavaBeans* introspection features [Mica].

In the following code, the `@Termifiable` annotation is used to associate instances of the `Pair` class with compound terms of the kind `pair(X,Y)`, where `X` and `Y` are Prolog terms corresponding to the values of `x` and `y`, respectively — the public fields of `Pair`:

```
@Termifiable(predicate="pair")
public class Pair  {
   public Term x;
   public Term y;
   ...
}
```

A `JavaVal` term is simply built from the temifiable Java object that needs to be passed to a Prolog method; for instance, the following code creates a new `JavaVal` term storing an instance of the termifiable class `Pair`:

```
Pair p = new Pair(new Int(4), new Atom('Hello!'));
JavaVal<Pair> vp = new JavaVal<Pair>(p);
```

The above `Pair` instance is converted by the PatJ framework into a Prolog compound term of the kind `pair(4, 'Hello!')`. Hence, the contents of a termifiable instance are more naturally accessed from Prolog code — without leveraging the PatJ library. For instance, the following Prolog method is used to swap the elements of a `Pair` object:

```
@PrologMethod(clauses="swap(pair(X,Y), pair(Y,X)).")
public abstract <$X extends JavaVal<Pair>,
                 $Y extends JavaVal<Pair>> $Y swap($X x);
```

The reader might appreciate the elegance and the compactness of the resulting code.

```
<exp>   :=  <term> | <term> (‘+’|‘-’) <exp>
<term>  :=  <fatt> | <fatt> (‘*’|‘/’) <term>
<fatt>  :=  <num> | ‘(’ <exp> ‘)’
<num>   :=  ‘0’ | ‘1’ | ‘2’ | ...
```

Figure 5.21: Syntax of arithmetic expressions

5 An Example: Parsing and Interpretation

An interesting area where declarative specifications are fruitfully exploited is in building parsers and interpreters. As an example, in this section we discuss how the features of the PatJ framework can be useful to rapidly prototype new parsers. Consider the definition of a (context-free) grammar for simple mathematical expressions reported in Figure 5.21. The code of the class `ExprParserEval` is reported in Figure 5.22 — an implementation of a mathematical expression parser built on top of PatJ. `ExprParser` defines a Prolog method, namely `parse_expr()`, used to build a declarative representation of a mathematical expression — a compound term — from a list of input tokens — Prolog atoms. Each non terminal symbols is mapped onto a different Prolog compound term: for instance, the Prolog predicate `plus(X,Y)` is used to encode non-terminal symbols of the kind `<plus>`.

Internally, the parser is defined in terms of *Definite Clause Grammar* rules (DCG henceforth). A DCG rule is defined using the '`-->`' operator, which replaces the '`:-`' operator used to define standard Prolog clauses. There is a DCG rule for each non-terminal symbol of the grammar `<exp>`, `<term>`, etc. — for instance, a DCG rule of the kind '`term(T) --> ...`' is associated with the a non-terminal symbol of the kind `<term>`. The predicates on the right-hand-side of the '`-->`' operator corresponds to the conditions that must be matched in order to parse a given non-terminal symbol — for instance, if both `fact(F)` and `term2(F,T)` yields true, then `T` is bound to a compound term corresponding to a parsed sub-expression of the kind`<term>`. A DCG rule might optionally specify a list of terminal symbols, enclosed in square brackets,

```
@PrologClass
public abstract class ExprParserVal {
  @TRACE
  @PrologMethod (clauses={"parse_expr(E,L):-phrase(expr(E),L).",
                          "expr(E) --> term(T), expr2(T,E).",
                          "expr2(T,E) --> ['+'],term(T),
                                          expr2(plus(T,T2),E).",
                          "expr2(T,E) --> ['-'],term(T),
                                          expr2(minus(T,T2),E).",
                          "expr2(T,T) --> [].",
                          "term(T) --> fact(F), term2(F,T).",
                          "term2(F,T) --> ['*'],fact(F2),
                                          term2(times(F,F2),T).",
                          "term2(F,T) --> ['/'],fact(F2),
                                          term2(div(F,F2),T).",
                          "term2(F,F) --> [].",
                          "fact(E) --> ['('],expr(E),[')'].",
                          "fact(X) --> [X],{number(X)}."})
  public abstract <$L extends Term<?>, $E extends List<?>> $L parse_expr($E expr);

  @TRACE
  @PrologMethod (clauses={"eval_expr(plus(L,R),X):-eval_expr(L, X1),
                                                   val_expr(R, X2), X is X1 + X2.",
                          "eval_expr(minus(L,R),X):-eval_expr(L, X1),
                                                    eval_expr(R, X2), X is X1 - X2.",
                          "eval_expr(times(L,R),X):-eval_expr(L, X1),
                                                    eval_expr(R, X2), X is X1 * X2.",
                          "eval_expr(div(L,R),X):-eval_expr(L, X1),
                                                  eval_expr(R, X2), X is X1 / X2.",
                          "eval_expr(X,X):-number(X)."})
  public abstract <$E extends Term<?>, $X extends Int> $X eval_expr($E expr);

  public static void main(String[] args) throws Exception {
    ExprParserVal ep = PJ.newInstance(ExprParserVal.class);
    java.util.List<Object> s1 = java.util.Arrays.asList(
            new Object[] {1,"+",2, "-", 3, "*", 5, "+", "(", 5, "/", 2, ")"});
    Term<?> expr = ep.parse_expr(new List(s1));
    System.out.println(ep.eval_expr(expr).toJava());
  }
}
```

Figure 5.22: A basic arithmetic expression parser/evaluator in PatJ

associated with a given non-terminal symbol, as in the definition of `expr2`. Finally, DCG rules can trigger standard Prolog goals, enclosed in curly braces, as in the definition of `fact`. The reader may appreciate how the DCG rules in the Prolog method `parse()` strictly adhere to the abstract specification of the grammar given in Figure 5.21. In method `main()`, `parse_expr()` is invoked with a tokenised expression of the kind '`['12','+','3','*','4']`'; the resulting parse tree — a compound term of the kind `plus(12, times(3, 4))` — is then passed to another Prolog method, namely `eval_expr()`, that is used to *evaluate* a compound term associated with a parsed mathematical

```
@PrologClass
public abstract class ExprParserVisitor {
  @PrologMethod (clauses={"parse_expr(E,L):-phrase(expr(E),L).",
                          "expr(E) --> term(T), expr2(T,E).",
                          "expr2(T,E) --> ['+'],term(T2),expr2(plus(T,T2),E).",
                          "expr2(T,E) --> ['-'],term(T2),expr2(minus(T,T2),E).",
                          "expr2(T,T) --> [].",
                          "term(T) --> fact(F), term2(F,T).",
                          "term2(F,T) --> ['*'],fact(F2),term2(times(F,F2),T).",
                          "term2(F,T) --> ['/'],fact(F2),term2(div(F,F2),T).",
                          "term2(F,F) --> [].",
                          "fact(E) --> ['('],expr(E),[')'].",
                          "fact(X) --> [X],{number(X)}."})
  public abstract <$L extends Term<?>, $E extends List<?>> $L parse_expr($E expr);

  @PrologMethod (clauses={"eval_expr(E,X, V):-this(Z), V <- visit(E) returns X."})
  public abstract <$E extends Term<?>,
                   $X extends Term<?>,
                   $V extends JavaRef<? extends EvalVisitor>>
                   $X eval_expr($E expr, $V visitor);

  public static void main(String[] args) throws Exception {
    ExprParserVisitor ep = PJ.newInstance(ExprParserVisitor.class);
    EvalVisitor v = PJ.newInstance(EvalVisitor.class);
    java.util.List<Object> s1 = java.util.Arrays.asList(
            new Object[] {1,"+",2, "-", 3, "*", 5, "+", "(", 5, "/", 2, ")"});
    Term<?> expr = ep.parse_expr(new List(s1));
    System.out.println(ep.eval_expr(expr, new JavaObject<EvalVisitor>(v)));
  }
}
```

Figure 5.23: A mathematical expression evaluator exploiting the visitor pattern

expression — in this case `eval_expr()` yields the numeric value `24`. Again, the reader might appreciate how the implementation of the Prolog method `eval_expr()` can concisely be expressed in terms of declarative rules.

5.1 Visitor Pattern Revisited

Figure 5.23 shows a slight variant of the example in Section 5, where the Java parse tree is evaluated by means of a visitor class whose methods are Prolog methods. The visitor pattern [GHJV95] is implemented through a PatJ interface, namely `PrologVisitor` (see Figure 5.24), which defines a set of abstract Prolog methods — these methods are overridden in specialised vistor classes. The Prolog method `visit()` is the entry point of the visitor pattern: it accepts a node representing an arithmetic expression — a compound term similar to the one discussed in Section 5 — and it recursively calls the most suitable visitor method. The dispatching logic is entirely written in Prolog: for

```
@PrologClass
interface PrologVisitor {
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_plus($E expr);
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_minus($E expr);
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_times($E expr);
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_div($E expr);
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_number($E expr);

    @PrologMethod (
        clauses={"visit(plus(L,R),X):-this(Z), Z <- visit_plus(plus(L,R)) returns X.",
                 "visit(minus(L,R),X):-this(Z), Z <- visit_minus(minus(L,R)) returns X.",
                 "visit(times(L,R),X):-this(Z), Z <- visit_times(times(L,R)) returns X.",
                 "visit(div(L,R),X):-this(Z), Z <- visit_div(div(L,R)) returns X.",
                 "visit(N,X):-number(N), this(Z), Z <- visit_number(N) returns X."})
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit($E expr);
}


@PrologClass
public abstract class EvalVisitor implements PrologVisitor {

    @Override
    @PrologMethod (clauses={"visit_plus(plus(L,R),X):-this(V),
                                                      V <- visit(L) returns X1,
                                                      V <- visit(R) returns X2,
                                                      X is X1 + X2."})
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_plus($E expr);

    @Override
    @PrologMethod (clauses={"visit_minus(minus(L,R),X):-this(V),
                                                        V <- visit(L) returns X1,
                                                        V <- visit(R) returns X2,
                                                        X is X1 - X2."})
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_minus($E expr);

    @Override
    @PrologMethod (clauses={"visit_times(times(L,R),X):-this(V),
                                                        V <- visit(L) returns X1,
                                                        V <- visit(R) returns X2,
                                                        X is X1 * X2."})
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_times($E expr);

    @Override
    @PrologMethod (clauses={"visit_div(div(L,R),X):-this(V),
                                                    V <- visit(L) returns X1,
                                                    V <- visit(R) returns X2,
                                                    X is X1 / X2."})
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_div($E expr);

    @Override
    @PrologMethod (clauses={"visit_number(N, N)."})
    public abstract <$E extends Term<?>, $X extends Term<?>> $X visit_number($E expr);
}
```

Figure 5.24: Visitor pattern in PatJ

instance, when a term of the kind `plus(X,Y)` is processed, the PatJ library is leveraged to recursively call the (abstract) Prolog method `visit_plus()`.

The class `EvalVisitor` defines a Prolog implementation for all the visitor methods in `PrologVisitor`; the implementation is, again, given in terms of Prolog code — the PatJ library is used to perform recursive call to visitor methods (in order to evaluate the subtrees associated with a given binary expression). Finally, the Prolog method `eval_expr()` in Figure 5.23 now accepts an object of type `EvalVisitor` — this is accomplished by leveraging the call-by-reference scheme. Note how the double-dispatching technique of the visitor pattern is easily reproduced in terms of Prolog programming.

5.2 A Java Parse Tree

Another possibility is to define a Prolog method generating a Java parse tree that can be directly evaluated in terms of Java code. Figure 5.25 shows a hierarchy of Java classes that can be used to encode parse tree nodes in simple arithmetic expressions. The root of this hierarchy is the `IExpr` interface, which defines an utility method, namely `eval()`, used to compute the value associated with a given arithmetic expression. Another abstract class, namely `BinaryExpr`, is used to factor over all binary expression classes, such as `Plus`, `Minus`, etc. — this class defines the fields associated with the two operands of a given binary expression tree. All classes leverage the `@Termifiable` annotation — this is discussed in greater details in the next section.

Figure 5.26 shows a revised example of the PatJ parser, where the DCG rules have been slightly adjusted in order to generate Java parse tree nodes instead of Prolog compound terms — this is accomplished by leveraging the PatJ predicate `java_object` discussed in Section 3. The Prolog method `parse()` adopts a call-by-reference scheme — its return type is `JavaRef<? extends IExpr>`; consequently the user must dereference the `JavaRef` instance returned by the `parse()` method in order to access the underlying Java parse tree — this is accomplished by calling `toJava()` on the `JavaRef` object returned by the Prolog method `parse()`. The reader may appreciate the degree of interoperability between Java and Prolog: creating a Java parse tree from Prolog and the evaluating it in Java just takes a few method calls —

```
public interface IExpr {
    public double eval();
}

public abstract class BinaryExpr implements IExpr {
    IExpr left; IExpr right;

    public BinaryExpr(Object left, Object right) {
        this.left = (left instanceof Integer) ?
                          new Num((Integer)left) :
                          (IExpr) left;
        this.right = (left instanceof Integer) ?
                          new Num((Integer)right) :
                          (IExpr) right;
    }

    public IExpr getLeft() {return left;}
    public IExpr getRight() {return right;}
    public void setLeft(IExpr _left) {left = _left;}
    public void setRight(IExpr _right) {right = _right;}
}

@Termifiable(predicate="plus")
public class Plus extends BinaryExpr {
    public Plus(Object left, Object right) { super(left, right); }
    public double eval() { return left.eval() + right.eval(); }
}

@Termifiable(predicate="minus")
public class Minus extends BinaryExpr {
    public Minus(Object left, Object right) { super(left, right); }
    public double eval() { return left.eval() - right.eval(); }
}

@Termifiable(predicate="multiply")
public class Multiply extends BinaryExpr {
    public Multiply(Object left, Object right) { super(left, right); }
    public double eval() { return left.eval() * right.eval(); }
}

@Termifiable(predicate="div")
public class Div extends BinaryExpr {
    public Div(Object left, Object right) { super(left, right); }
    public double eval() { return left.eval() / right.eval(); }
}

@Termifiable(predicate="num")
public class Num implements IExpr {
    int num;

    Num(int i) { this.num = i; }
    public double eval() { return num; }
    public int getNum() {return num;}
    public void setNum(int num) {this.num = num;}
}
```

Figure 5.25: Hierarchy of Java classes for representing mathematical expressions

```
@PrologClass
public abstract class MathExprParser {

  @PrologMethod (
     clauses={"parse_expr(E,L):-phrase(expr(E),L).",
              "expr(E) --> term(T), expr2(T,E).",
              "expr2(T,E) --> ['+'], term(T2),
                              {java_object('Plus', [T, T2], SUM)},
                              expr2(SUM, E).",
              "expr2(T,E) --> ['-'],term(T2),
                              {java_object('Minus', [T, T2], DIFF)},
                              expr2(DIFF, E).",
              "expr2(T,T) --> [].",
              "term(T) --> fact(F), term2(F,T).",
              "term2(F,T) --> ['*'],fact(F2),
                              {java_object('Multiply', [F, F2], MUL)},
                              term2(MUL,T).",
              "term2(F,T) --> ['/'],fact(F2),
                              {java_object('Div', [F, F2], DIV)},
                              term2(DIV,T).",
              "term2(F,F) --> [].",
              "fact(E) --> ['('],expr(E),[')'].",
              "fact(X) --> [X],{number(X)}."})
  public abstract <$L extends JavaRef<? extends IExpr>,
                   $E extends List<?>> $L parse_expr($E expr);

  public static void main(String[] args) throws Exception {
    MathExprParser ep = PJ.newInstance(MathExprParser.class);
    java.util.List<Object> s1 = java.util.Arrays.asList(
       new Object[] {1,"+",2, "-", 3, "*", 5, "+", "(", 5, "/", 2, ")"});
    IExpr expr = ep.parse_expr(new List(s1)).toJava();
    System.out.println(expr);
    System.out.println(expr.eval());
  }
}
```

Figure 5.26: A mathematical expression parser generating a Java AST

all the complexity is hidden by the PatJ framework.

5.3 A Prolog Evaluator

The code in Figure 5.27 shows another variant of the example given in section 5.2 where evaluation is performed in Prolog rather than in Java. This is possible thanks to the `@Termifiable` annotation that allows straightforward mapping of custom Java objects into Prolog compound terms — an instance of a termifiable class (e.g. `Plus`) is turned into a suitable Prolog representation (the compound term `plus(X,Y)`). Note that a Prolog class exploiting one or more termifiable classes must be annotated with a `@WithTermifiable` annotation listing the qualified names of such classes — this is required in order to updated the shared termifiable registry (see Section 4.3.2) and,

```
@PrologClass
@WithTermifiable({"alice.tuprologx.pj.test.expr.Plus",
                  "alice.tuprologx.pj.test.expr.Minus",
                  "alice.tuprologx.pj.test.expr.Multiply",
                  "alice.tuprologx.pj.test.expr.Div",
                  "alice.tuprologx.pj.test.expr.Num"})
public abstract class ExprParserVal {
   @PrologMethod (clauses={"parse_expr(E,L):-phrase(expr(E),L).",
                           "expr(E) --> term(T), expr2(T,E).",
                           "expr2(T,E) --> ['+'],term(T2),expr2(plus(T,T2),E).",
                           "expr2(T,E) --> ['-'],term(T2),expr2(minus(T,T2),E).",
                           "expr2(T,T) --> [].",
                           "term(T) --> fact(F), term2(F,T).",
                           "term2(F,T) --> ['*'],fact(F2),term2(times(F,F2),T).",
                           "term2(F,T) --> ['/'],fact(F2),term2(div(F,F2),T).",
                           "term2(F,F) --> [].",
                           "fact(E) --> ['('],expr(E),[')'].",
                           "fact(X) --> [X],{number(X)}."})
  public abstract <$L extends JavaVal<?>, $E extends List<?>> $L parse_expr($E expr);

  @PrologMethod (clauses={"eval_expr(plus(L,R),X):-eval_expr(L, X1),
                                                   eval_expr(R, X2), X is X1 + X2.",
                          "eval_expr(minus(L,R),X):-eval_expr(L, X1),
                                                    eval_expr(R, X2), X is X1 - X2.",
                          "eval_expr(times(L,R),X):-eval_expr(L, X1),
                                                    eval_expr(R, X2), X is X1 * X2.",
                          "eval_expr(div(L,R),X):-eval_expr(L, X1),
                                                  eval_expr(R, X2), X is X1 / X2.",
                          "eval_expr(X,X):-number(X)."})
  public abstract <$E extends JavaVal<?>, $X extends Int> $X eval_expr($E expr);

  public static void main(String[] args) throws Exception {
    ExprParserVal ep = PJ.newInstance(ExprParserVal.class);
    java.util.List<Object> s1 = java.util.Arrays.asList(
            new Object[] {1,"+",2, "-", 3, "*", 5, "+", "(", 5, "/", 2, ")"});
    JavaVal<?> expr = ep.parse_expr(new List(s1));
    System.out.println(ep.eval_expr(expr).toJava());
  }
}
```

Figure 5.27: A mathematical expression parser and evaluator exploiting termifiable classes

consequently, to disambiguate marshalling/unmarshalling of compound terms associated with termifiable classes.

The `ExprParserVal` class defines a Prolog method, namely `parse_expr()`, which returns a Java parse tree; the Java objects corresponding to the parse tree nodes are here built *implicitly*, by leveraging the call-by-value scheme: each compound term of the kind `plus`, `minus`, `times`, `div` is converted into an instance of the corresponding termifiable class `Plus`, `Minus`, `Multiply` and `Div`, respectively — there is no need to explicitly create Java objects through the PatJ library predicates. The parse tree returned by `parse_expr` can thus

be passed to another Prolog method, namely `eval_expr`; again, this is done leveraging the call-by-value scheme — this time, the framework will convert a Java object (of type `IExpr`) into a Prolog compound term. Hence, the Prolog code associated with the Prolog method `eval_expr()` can access the contents of a binary expression node in a declarative fashion, as for any other Prolog compound term. This leads to a compact and elegant implementation.

Chapter 6

Conclusions

The contributions of this thesis are twofold: first, we proposed a revised implementation for Java generics that greatly enhances the expressiveness of the Java platform by adding reification support for generic types; secondly, we discussed how Java genericity can be leveraged in a real world case-study in the context of the multi-paradigm language integration.

In this thesis we discussed all the issues that must be tackled to implement a full-fledged reification support for generics/wildcards types; this resulted in a reification scheme — namely, the GCVM — that is both complete, efficient and fully backward compatible. The effectiveness of our solution has been validated by real world benchmarks such as the GJ compiler [Mic01] (see Section 5). To the best of our knowledge, the GCVM is also the only proposal effectively addressing all the features included in the Java Programming Language [JGSB05]. On the one hand, existing runtime approaches, as in [MBL97] support reified generics through a true language extension so that backward compatibility is typically compromised. On the other hand, certain subtleties of the Java type-system are not easily mimicked by standard (non-generic) Java code; consequently, whether smooth extensions of the legacy Java compiler, such as NextGen [SC06], would ever be able to provide a complete reification scheme is still an open issue. Concerning subtyping for instance, types of the kind `List<? super T>` are contravariant, hence, the set of their supertypes is not closed: for any newly defined class `C` such that `C<:T`, type `List<? super C>` should be a supertype of `List<? super T>`. Since

NextGen is conceived around the idea of reusing concrete Java classes to simulate each different instantiation of a generic type used in an application — class `List$String` for type `List<String>` and so on — supporting open subtyping hierarchies can lead to serious implementation issues.

We then presented PatJ, a framework that significantly improves the seamless integration of Prolog code into Java applications, exploiting the tuProlog technology. PatJ is structured in a compositional way — in fact, the core of the PatJ framework can be seen as a tiny wrapper around the tuProlog engine providing just basic capabilities; on top of this layer PatJ defines a hierarchy of *generic* Java classes modelling first-order logic terms that feature automated marshaling/unmarshaling from Java to Prolog, and vice-versa — this API is arguably one of the most remarkable applications of Java generics and wildcards so far. This hierarchy is leveraged in order to fill the gap between method invocation and Prolog goal satisfaction, exploiting Java type inference in method calls — we believe this plays a crucial role in enabling those programmers who are familiar with Java mainstream programming to easily incorporate declarative features into their programs. Moreover, the possibility of expressing rich (generic) types for Prolog terms, along with the exploitation of Java type inference for checking consistency and reconstructing bridging information, allows for seamless integration of Prolog code into Java classes and methods. As such, PatJ is a concrete attempt to address the problems affecting existing solutions for integrating Java and Prolog. On the one hand, when integration is accomplished by merging the two paradigms into a single hybrid language supporting both Object-Oriented and logic features, as in [Esp06], the resulting language is typically too complex, thus making mainstream application development an harder task. On the other hand, library-based integration approaches [JPL, Kin05, PLB, DOR05] typically fail to provide true language integration, and some "boilerplate code" has to be implemented each time to fix the paradigm mismatch.

Acknowledgments

I would like to thank Gilad Bracha, for he gave me the opportunity to work in Sun Labs for four months in 2003. During these months, I had the chance to meet members of the compiler team, most noticeably Neal Gafter, who provided me many precious insight on the fundamentals of the `javac` compiler, and Mikhail Dmitriev, my tutor — working with them has been an amazing experience. I would also like to thank Alex Buckley, Peter Ahé and Paul Hohensee, as they took the time to do a short trip to Cesena in January 2007 in order evaluate the outcome of the GCVM project. Simone Pellegrini has been a valuable collaborator, who contributed to some of the technical details and the performance measurments discussed in this thesis. Finally, I would like to thank professors Antonio Natali and Andrea Omicini, for their many thoughtful suggestions that contributed to improve the PatJ framework in many ways, and Giulio Piancastelli, who helped me with many tuProlog related issues. Special thanks to Mirko Viroli, for the patience and support he relentlessly showed during the countless technical discussions that shaped many parts of the work described in this thesis.

Bibliography

[ABC03] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. *SIGPLAN Not.*, 38(11):96–114, 2003.

[ACC93] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 157–170, New York, NY, USA, 1993. ACM.

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 215–230, Atlanta, Georgia, 5–9 October, 1997. ACM, New York.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of OOPSLA '06*, pages 57–74, New York, NY, USA, 2006. ACM Press.

[AR08] Suad Alagic and Mark Royer. Genericity in Java: persistent and database systems implications. *VLDB J.*, 17(4):847–878, 2008.

[BCD+99] Jean-Daniel Boissonnat, Frédéric Cazals, Frank Da, Olivier Devillers, Sylvain Pion, François Rebufat, Monique Teillaud, and Mariette Yvinec. Programming with CGAL: the example of

triangulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 421–422, New York, NY, USA, 1999. ACM.

[BGL02] *The boost graph library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[BKG+06] Johan Brichau, Andy Kellens, Kris Gybels, Kim Mens, Robert Hirschfeld, and Theo D'Hondt. Application-Specific Models and Pointcuts Using a Logic Meta Language. In *ISC*, pages 1–22, 2006.

[Bon04] Jonas Bon. Annotation-driven AOP for Java. In *Annual European Conference on Java and Object-Oriented Software Engineering*, 2004. http://aspectwerkz.codehaus.org.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Not.*, 33(10):183–200, 1998.

[CAF04] Brian Cabana, Suad Alagić, and Jeff Faulkner. Parametric polymorphism for Java: is there any hope in sight? *SIGPLAN Not.*, 39(12):22–31, 2004.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and*

computer architecture, pages 273–280, New York, NY, USA, 1989. ACM.

[CD09] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *FTfJP '09: Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, pages 1–7, New York, NY, USA, 2009. ACM.

[CDE08] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java Wildcards. In *ECOOP 08*, Lecture Notes in Computer Science, June 2008.

[Cha92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[CV07] Maurizio Cimadamore and Mirko Viroli. A Prolog-oriented extension of Java programming based on generics and annotations. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 197–202, New York, NY, USA, 2007. ACM.

[CV08a] Maurizio Cimadamore and Mirko Viroli. Integrating Java and Prolog through generic methods and type inference. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 198–205, New York, NY, USA, 2008. ACM.

[CV08b] Maurizio Cimadamore and Mirko Viroli. On the reification of Java wildcards. *Sci. Comput. Program.*, 73(2-3):59–75, 2008.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[DJ05] Gabriel Dos Reis and Jaakko Järvi. What is generic programming? In *Proceedings of the First International Workshop of Library-Centric Software Design (LCSD '05). An OOPSLA '05 workshop*, October 2005.

[DKTE04] Alan Donovan, Adam Kiežun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 15–34, New York, NY, USA, 2004. ACM.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[DRS06] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM.

[Esp06] M. Espák. Japlo: Rule-based programming on java. *j-jucs*, 12(9):1177–1189, 2006.

[Gaf04] Neal Gafter. Puzzling Through Erasure: answer section. http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GHS05] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent Java programs. *Sci. Comput. Program.*, 58(3):384–411, 2005.

[HLS09] William Harrison, David Lievens, and Fabio Simeoni. Safer typing of complex API usage through Java generics. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 67–75, New York, NY, USA, 2009. ACM.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherwieght Java: a minimal core calculus for Java and GJ. *SIGPLAN Not.*, 34(10):132–146, 1999.

[IV06] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006.

[JGSB05] Bill Joy, James Gosling, Guy Steele, and Gilad Bracha. *The Java Language Specification (Third Edition)*. Addison-Wesley, New York, 2005.

[JLo02] JLog team. JLog – Prolog in Java. http://jlogic.sourceforge.net/, 2002.

[JPL] JPL: A bidirectional Prolog/Java interface. http://www.swi-prolog.org/.

[JWL03] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 228–244, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[k-p] K-Prolog Official Website. http://www.kprolog.com/.

[KETF07] Adam Kieżun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE'07, Proceedings of the 29th International Conference on Software*

Engineering, pages 437–446, Minneapolis, MN, USA, May 23–25, 2007.

[Kin05] Nobukuni Kino. Jipl: Java interface to prolog. http://www.kprolog.com/jipl/, 2005.

[Kow74] Robert A. Kowalski. Predicate Logic as Programming Language. In *IFIP Congress*, pages 569–574, 1974.

[KP06] Andrew J. Kennedy and Benjamin C. Pierce. On Decidability of Nominal Subtyping with Variance, September 2006. FOOL-WOOD '07.

[KR08] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 189–197, New York, NY, USA, 2008. ACM.

[KREY06] Andrew Kennedy, Claudio Russo, Burak Emir, and Dachuan Yu. Variance and Generalized Constraints for C# Generics. In *European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[Lov07] Howard Lovatt. Comparing Inner Class/Closure Proposals. http://www.artima.com/weblogs/viewpost.jsp?thread=202004, 2007.

[LSL99] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. *SIGPLAN Not.*, 34(10):399–414, 1999.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MBL97] Andrew C. Meyers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 15-17 January, 1997. ACM, New York.

[Mey86] Bertrand Meyer. Genericity versus inheritance. *SIGPLAN Not.*, 21(11):391–405, 1986.

[Mey89] B. Meyer. Reusability: the case for object-oriented design. *Software reusability: vol. 2, applications and experience*, pages 1–33, 1989.

[Mica] Sun Microsystems. Java Platform, Standard Edition 6 API Specification. http://java.sun.com/javase/6/docs/api/.

[Micb] Sun Microsystems. Proposed extension to the classfile format (Chapter 4 of the Java Virtual Machine Specification). http://java.sun.com/docs/books/jvms/second_edition/ClassFileFormat-Java5.pdf.

[Mic01] Sun Microsystems. Adding generics to the JavaTM programming language: Public review. JSR- 000014-PR, Sun Microsystems, Palo Alto, CA, 2001.

[Mic05] Sun Microsystems. Jsr 269: Pluggable annotation processing api. JSR- 000269-PR, Sun Microsystems, 2005.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Min] Minerva Official Website. http://www.ifcomputer.co.jp/MINERVA/.

[MS88] David Musser and Alexander A. Stepanov. Generic Programming. In *Symbolic and algebraic computation: ISSAC '88*, pages 13–25. Springer, 1988.

[MZ06] Karl Mazurak and Steve Zdancewic. Type inference for Java 5: Wildcards, f-bounds, and undecidability. http://www.cis.upenn.edu/ stevez/note.html, 2006.

[N'g06] Olayinka N'guessan. Generic programming in Scala. Master's thesis, Texas A&M University, 2006.

[Nin07] Jaime Nino. The cost of erasure in Java generics type system. *J. Comput. Small Coll.*, 22(5):2–11, 2007.

[OG08] Bruno C.d.S. Oliveira and Jeremy Gibbons. Scala for generic programmers. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming*, pages 25–36, New York, NY, USA, 2008. ACM.

[ON94] Andrea Omicini and Antonio Natali. Object-Oriented Computations in Logic Programming. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 194–212, London, UK, 1994. Springer-Verlag.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15-17 January, 1997. ACM, New York.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, New York, NY, USA, 2008. ACM.

[PLB] Documentation for PrologBeans. http://www.sics.se/isl/sicstuswww/site/index.html.

[Plu07] Martin Plumicke. Typeless programming in Java 5.0 with wildcards. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 73–82, New York, NY, USA, 2007. ACM.

[PNCB06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 311–324, New York, NY, USA, 2006. ACM.

[PT98] Benjamin P. Pierce and David N. Turner. Local Type Inference. In *Symposium on Principles of Programming Languages*, pages 252–265. ACM, New York, 1998.

[RS06] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *POPL*, pages 295–308, 2006.

[SA98] Johe H. Solorzano and Suad Alagic. Parametric polymorphism for Java: A reflective solution. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 216–

225, Vancouver, British Columbia, Canada, 18–22 October, 1998. ACM, New York.

[SC06] James Sasitorn and Robert Cartwright. Efficient first-class generics on stock Java virtual machines. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1621–1628. ACM, 2006.

[SC08] Daniel Smith and Robert Cartwright. Java type inference is broken: can we fix it? In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 505–524, New York, NY, USA, 2008. ACM.

[SDNB02] Nathanael Sharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behavior. Technical report, 2002.

[SK01] Don Syme and A. Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Languages Design and Implementation*, Snowbird, Utah, 20–22 June, June 2001. ACM, New York.

[SL94] A. A. Stepanov and M. Lee. The Standard Template Library. Technical report, 1994.

[SL05] Jeremy G. Siek and Andrew Lumsdaine. Essential language support for generic programming. *SIGPLAN Not.*, 40(6):73–84, 2005.

[Str] Bjarne Stroustrup. C++ Style and Technique FAQ. http://www2.research.att.com/ bs/bs_faq2.html#constraints.

[Sun] Sun Microsystems. Bug 4929881. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4993221.

[Sun09] Sun Microsystems. The Java EE 6 Tutorial. http://java.sun.com/javaee/6/docs/tutorial/doc/, 2009.

[swi] SWI-Prolog Official Website. http://www.swi-prolog.org/.

[TEPH05] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In Phil Wadler, editor, *Proceedings of FOOL 12*, Long Beach, California, USA, January 2005. ACM, School of Informatics, University of Edinburgh. Electronic publication.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM.

[tuP02] tuProlog Team. tuProlog at SourceForge. http://sourceforge.net/projects/tuprolog/, 2002.

[vDD04] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. *SIGPLAN Not.*, 39(10):1–14, 2004.

[VEK76] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J. ACM*, 23(4):733–742, 1976.

[Vir03a] Mirko Viroli. A Type-Passing Approach for the Implementation of Parametric Methods in Java. *Comput. J.*, 46(3):263–294, 2003.

[Vir03b] Mirko Viroli. A type-passing approach for the implementation of parametric methods in Java. *The Computer Journal*, 46(3), 2003.

[Vir05] Mirko Viroli. Effective and Efficient Compilation of Run-Time Generics in Java. In Viviana Bono, Michele Bugliesi, and Sophia

Drossopoulou, editors, *2nd Workshop on Object-Oriented Developments (WOOD 2004)*, volume 138(2) of *Electronic Notes in Theoretical Computer Science*, pages 95–116. Elsevier Science B.V., CONCUR 2004, London, UK, 30 August 2005.

[VN00] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN*, 35(10):146–165, October 2000. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000), Minneapolis, MA, USA, 15-19.

[VR05] Mirko Viroli and Giovanni Rimassa. On Access Restriction with Java Wildcards. *Journal of Object Technology*, 4(10), 2005. Special Issue: OOPS Track at ACM SAC 2005.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[WLT07] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized Interfaces for Java. In *ECOOP 2007, Proceedings*, LNCS. Springer-Verlag, July 2007. 25 pages; To appear.

[WT09] Stefan Wehr and Peter Thiemann. On the Decidability of Subtyping with Bounded Existential Types. In *Proceedings of the Seventh Asian Symposium on Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, Seoul, South Korea, 2009. SPRINGER.

[ZPA+07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.