TypePlug

Pluggable Type Systems for Smalltalk

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Niklaus Haldimann

April 2007

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Further information about this work, the tools used and an online version of this document can be found at the following places.

Niklaus Haldimann
nhaldimann@gmx.ch
http://www.squeaksource.com/TypePlug.html

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://www.iam.unibe.ch/~scg/

Contents

Abstract v

Acknowledgements vii

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Thesis Outline . . . 3

2 Programming with Optional Types 5
- 2.1 A Coding Session . . . 5
- 2.2 Annotating Source Code . . . 8
- 2.3 The Type Browser . . . 10

3 Defining a Type System 13
- 3.1 Types and Their Representation . . . 13
- 3.2 Typing Elements of Smalltalk Syntax . . . 16
- 3.3 Subtyping and Unification . . . 17
- 3.4 Other Elements of a Type System Implementation . . . 18
- 3.5 The Static Type System . . . 18

4 Type Checking 21
- 4.1 The Type Checker in a Nutshell . . . 21
- 4.2 Ensuring Type Safety . . . 22
- 4.3 Type Inference . . . 24
- 4.4 Customizing Type Checking . . . 29
- 4.5 Problematic Idioms . . . 32

5 Case Studies 35
- 5.1 A Class-Based Type System . . . 35
- 5.2 A Confinement Type System . . . 40

6 Related Work 47

7 Conclusion 53

7.1 Future Work 54

A Theoretical Background **57**

Abstract

Statically and dynamically typed programming languages have complementary strengths. While static typing provides early error detection, optimized execution and machine-checkable documentation, dynamic typing makes a language more expressive, better suited for rapid prototyping and more adaptive to changing requirements.

Pluggable type systems strive to combine these strengths by declaring types and type systems to be optional. Supporting multiple coexisting type systems, pluggable type systems open up a language to various kinds of static analyses other than those provided by traditional type systems.

We present TypePlug, a framework for pluggable type systems for Smalltalk. TypePlug provides infrastructure to optionally annotate source code with types and to define in a simple way semantics for type systems. It contains a generic type checking algorithm, dealing with issues arising when statically checking a a dynamically typed language. To improve type checking results and the user experience, TypePlug integrates optional type inference. We introduce type systems comparable to traditional class-based type systems and a type system for confinement, proving the validity of our approach.

Acknowledgements

First of all, I would like to thank Prof. Dr. Oscar, head of the Software Composition Group, for giving me the opportunity to work in his group, for continuous guidance during my adventures in types and for his detailed comments on my thesis.

I would like to thank Prof. Dr. Stéphane Ducasse, now at the University of Savoie in France, for pointing me to the field of pluggable type systems and suggesting it as a thesis topic.

Also I would like to thank the members of the Software Composition Group for their valuable comments and suggestions, especially Marcus Denker for reading and commenting on drafts of this thesis. Philippe Marschall made a crucial contribution by developing the Persephone framework during the course of his own master thesis and supporting me in its use. His excellent framework enabled me to focus on conceptual rather than implementation issues.

Last but not least, I would like to thank the people around me in my private life for putting up with an unusually stressed out me during the last few months of my studies—my family, my housemates and my friends, especially Dan who was keeping my spirits high like no one else.

Niklaus Haldimann
April 2007

Chapter 1

Introduction

Statically typed languages usually sport a *mandatory* type system, meaning that everything must be typed and type checking is enforced at compile-time. The advantages of static, mandatory typing are well established: it provides early error detection, optimized execution and machine-checkable documentation. In contrast, dynamically typed languages defer type checking to a runtime stage and do not mandate or do not even support any form of explicit type annotation in source code. There are compelling reasons to choose a dynamically typed language over a statically typed one: they are often more expressive, they are better suited for rapid prototyping and adapt better to changing requirements.

The strengths of statically and dynamically typed languages complement each other, so bridging the gap between the two approaches is a desirable goal. Bracha proposed *pluggable type systems* [Bra04], language environments that incorporate several coexisting *optional* type systems. He defines an optional type system to be one that:

1. has no effect on the runtime semantics of the programming language, and

2. does not mandate type annotations in the syntax.

If a language offers a way to implement optional type systems as "plug-ins" we arrive at pluggable type systems. Bracha argues that "pluggable types can provide most of the advantages of mandatory type systems without most of the drawbacks", citing the constrained expressiveness of a language as a drawback.

What is more, pluggable type systems open up a language to more than just the traditional class- and interface-based type systems known from mainstream (object-oriented) statically typed languages. In general, a type sys-

tem abstracts and expresses certain properties of a program, which can be more than just constraints about which classes of values can be stored in a variable. Type systems can govern aliasing, ownership, information flow and more.

A framework for pluggable type systems in a dynamically typed language faces two main challenges, arising from both the concepts of pluggable types and of dynamic typing:

- The key aspect that types should be *optional* means that a pluggable type system must be able to handle partially typed or untyped programs. A programmer cannot be expected to exhaustively add types to his code and employed libraries until he can benefit from a pluggable type system. The same goes for implementors of a pluggable type system. The cost of creating a new type system would be too high if it required providing types for, *e.g.,* core libraries of a language.
- There are fundamental obstacles when retrofitting static type checking onto a dynamically typed language. The lack of an existing static type system as a basis complicates static analysis of code, for example because due to dynamic binding we cannot statically determine which method will be looked up at runtime. Furthermore, these languages support features such as reflection that do not lend themselves to static type checking at all.

In this thesis we present TypePlug, a framework for pluggable type systems for Smalltalk, for the Squeak dialect to be precise. We chose Squeak Smalltalk as an example of a dynamically typed language for its clean object-oriented model, its simple syntax, its support for the modification of core libraries and the availability of a powerful framework for source code annotations.

We address the two challenges outlined above, mainly by combining type checking with certain forms of type inference (reducing the need for exhaustively typing code) and by pragmatically type checking method calls.

1.1 Contributions

This thesis makes the following contributions:

- A model for optional type systems (Chapter 3)
- A model for the static type system inherently present in Smalltalk source code and its integration with optional type systems (Section 3.5)

- A description of a generic algorithm to statically type check Smalltalk while dealing with optional type annotations, enhanced with type inference (Chapter 4)
- Implementations of pluggable type systems for catching unwanted `nil` values and for confining instance variables (Section 5.2) as well as a traditional type system based on classes (Section 5.1)
- An implementation of a programming environment supporting type annotations and type checking in Squeak Smalltalk

1.2 Thesis Outline

Chapter 2 demonstrates the usefulness of optional types and type checking. We first walk through a typical programming session using optional types. Then we describe our implementation from the point of view of a programmer, detailing how annotations are made and introducing a customized code browser.

Chapter 3 discusses our approach to modeling pluggable type systems. We explain how a type system is implemented in our model, defining types, typing for elements of Smalltalk syntax and finally a subtyping relation and unification operation. The implementation of the non-nil type system is discussed as an example.

Chapter 4 details our algorithm for type checking within the constraints of a pluggable type system. We describe the basic approach of assigning types to AST nodes and inserting subtyping checks at appropriate points. We then discuss type inference for local variables and return types as well as some special cases such as control flow messages and block evaluations. The chapter ends with a discussion of coding idioms that are problematic with this approach to type checking.

Chapter 5 gives two examples of type systems we implemented on top of our framework, a class-based type system and a confinement type system. The class-based type systems features many traits of modern static type systems such as generic types, polymorphic messages and union types. The confinement system optionally restricts instance variable to use within a given class, preventing its references to leak to other classes.

Chapter 6 summarizes and compares related work in the areas of pluggable type systems, optional typing and type systems for Smalltalk.

Chapter 7 concludes with a recapitulation of our results and findings, summarizing how we faced the two main challenges outlined in this introduction. We finish with an outlook on future work.

Appendix A gives a brief theoretical overview of key terms and concepts used in this thesis.

Chapter 2

Programming with Optional Types

TypePlug provides to programmers a framework to optionally annotate code with types and have the code checked for type safety based on the annotations made. This chapter will demonstrate how optional types and type checking might be used in a typical coding session and discusses other aspects of TypePlug from the point of view of a programmer.

2.1 A Coding Session

While our framework is open to support a wide range of different kinds of type systems we will in the following discussion use our implementation of a non-nil type system. This type system has exactly one type, the nonNil type. Its semantics are as simple as it gets: if a variable has the nonNil type it cannot hold the value nil. Anything not typed nonNil is considered to potentially evaluate to nil (there is no explicit nil type to keep this type system as simple as possible).

To declare a variable to be of type nonNil we add the annotation <**:nonNil :**> to it. In the following class definition of a two-dimensional line we declare the instance variable endPoint to be nonNil:

```
Object subclass: #Line
    uses: TReflectiveMethods
    typedInstanceVariables: 'startPoint endPoint <:nonNil :>'
    typedClassVariables: ''
```

Notice the trait TReflectiveMethods that we use in this class [1]. It pulls in the annotation framework that we need to annotate its methods. We assume that we have not added any other annotations yet to the methods of this Line class. If we now browse the source of one of the methods we can experience the type checking in action. Here is a method of a line that moves it horizontally by an amount of units:

```
moveHorizontally: anInteger
    startPoint := self movePoint: startPoint horizontally: anInteger.
    endPoint := self movePoint: endPoint horizontally: anInteger <– type 'TopType'
        of expression is not compatible with type 'nonNil' of variable 'endPoint'.
```

When displaying the source of a method it is type checked and type errors are shown inline. The type error found in the above method is emphasized. What it says is that the expression self movePoint: endPoint horizontally : anInteger was found to have the top type which basically means that its type is *unknown*. But the instance variable endPoint expects the type nonNil, hence the type error.

To fix this type error we have to look into the method movePoint:horizontally: and declare its return type to be nonNil. This is how it looks like:

```
movePoint: aPoint <:nonNil :> horizontally: anInteger
    ↑ (aPoint addX: anInteger y: 0) <:nonNil :>
```

A return type annotation must be added to the return statement expression. Notice how parentheses are needed here to apply the annotation to the whole expression. Adding a return type to a method has two repercussions: in call sites of the method the message send expression will be typed accordingly and within the method its return statements will be type checked. Here this means that the expression aPoint addX: anInteger y: 0 must be of type nonNil (we assume we annotated the addX:y: method with a nonNil return type already, so no type error occurs here).

As an example of an argument type we annotated the first argument of movePoint:horizontally: with a nonNil type as well. Adding an argument type declares the argument to be of that type within the method, and it is also a requirement to arguments passed when the method is used. Because of this requirement we now get a different type error when we look at moveHorizontally: again:

```
moveHorizontally: anInteger
    startPoint := self movePoint: startPoint horizontally: anInteger <– in message '
        movePoint:horizontally:' of class 'Line' is argument 'TopType' not
```

[1] *Traits* are basic building blocks of classes and primitive units of code reuse, see [SDNB03] for details.

```
        compatible with expected type 'nonNil'.
    endPoint := self movePoint: endPoint horizontally: anInteger.
```

The startPoint instance variables does not have an annotation so its type is the default, the top type, which is not what is required for the first argument to movePoint:horizontally:. The obvious fix to this type error is to declare startPoint to be nonNil as well. A different strategy that TypePlug offers is to *cast* an expression to an arbitrary type, in this example by adding the annotation <**:castNonNil :**>.

```
moveHorizontally: anInteger
    startPoint := self movePoint: startPoint <:castNonNil :> horizontally:
        anInteger.
    endPoint := self movePoint: endPoint horizontally: anInteger.
```

With a cast a programmer asserts to the type checker that he knows the type of a given expression. Casts are useful for one-off dissolutions of type errors where more type annotations would be overkill or impossible, for example because the cast expression uses untyped third-party code.

Let's annotate one more method of Line with a return type:

```
hasStartPoint
    ↑ (startPoint notNil) <:nonNil :>
```

We are still operating under the assumption that no other type annotations have been made than those previously discussed. We surely have not annotated any notNil method with types, so how come we don't get a type error here because the expression startPoint notNil can't be proven to be nonNil ? The reason is that the type checker does try to infer return types of methods if they are not explicitly annotated. The notNil method has two implementations, one in the base class Object—simply returning false—and one in UndefinedObject—simply returning true. So obviously invocations of notNil will always return a nonNil boolean value, which is what the type checker figured out in this case, sparing us from explicitly annotating the notNil method with a return type.

While this preceding coding session serves as a light introduction to TypePlug, it also illustrates an important issue when programming with optional types. Once one adds just one type to previously untyped code the need for more type annotations usually comes up to have other parts of the code type checked. Most likely, these additional types prompt for even more type annotations. This means that quite a bit of work is usually required from a programmer if he introduces types in a section of his code. A framework for optional typing should be aware of this problem and minimize the work required by the programmer. As demonstrated in the coding session, two of

our strategies in this area are type inference and casts, reducing the need for explicit type annotations.

2.2 Annotating Source Code

TypePlug makes use of the Persephone framework [Mar06] for the type annotations to Squeak source. Persephone supports annotations to any variable and expression within a method. As the code examples in the previous section have shown, type annotations can be added in some places in a method. The following is a list of these places and including definitions of what annotations in those places mean from the point of view of a developer.

- *Instance and class variables* are typed by just appending type annotations to their names in a class definition. An annotation on an instance or class variable means that all assignments to the variable should be type checked.
- A *method argument* with a type annotation signals that the type of this argument should be checked in all the call sites of the method. It also sets the type of the variable within the method.
- A type annotation on a *return statement* defines the type of the returned object and it states that all return statements of the method should be type checked. With several return statements in a method, only one annotation on one statement is required to define a return type. A method with several conflicting annotated return statements will not compile. The return type of a method without return statements is always the type that the respective type system assigns to the pseudo-variable self.

  Note that the return type annotation must be added to the *whole* expression in a return statement. If this expression is, *e.g.,* a message send parentheses around the whole expression must be used to place the annotation correctly. This requirement is a slight burden on the programmer, but with future developments of the annotation framework a better way to specify return types should emerge. As an example, the return statement

  ```
  ↑ self add: 2 <:nonNil :>
  ```

  is not annotated—the annotation is applied to the literal 2. With parentheses the annotation is correctly recognized as a return type annotation:

  ```
  ↑ (self add: 2) <:nonNil :>
  ```

- A *local variable* with a type annotation specifies the type of the variable within the method and states that assignments to the variable should be type checked.
- Similar to method arguments, *block variables* can be annotated with a type. These type annotations define the type for a variable within the block. When the block type for the block is constructed its argument types are based on these annotations (a block type consists of a block's argument types and its return type). Also, if a block is evaluated by sending it, for example, the value:value: message, the argument types are checked using the block variable annotations.

Since TypePlug supports an arbitrary number of coexisting pluggable type systems, any of these annotation places can have more than one type annotation. In the source, annotations are simply added one after another. For example, to give the instance variable list both the nonNil type from the non-nil type system as well as confine it to a class using the confinement type system we write:

```
list <:nonNil :> <:confine: toClass :>
```

In addition to type annotations TypePlug support *cast annotations*. These are used to cast any expression to some type. Casts are unchecked—they always succeed, regardless of whether they can statically be proven type safe. As with return type annotations, care has to be taken to apply a cast to the correct expression. An annotation binds to the inner most expression of the code to the left of the annotation. Parentheses must be used to annotate composite expressions. Take this message send as an example:

```
(self printOnStream: someStream <:castNonNil :>) <:castNonNil :>
```

The first annotation applies to the variable someStream. With the parentheses, the second annotation applies to the whole message send.

One more peculiarity of argument and return type annotations should be mentioned: if they are not explicitly specified on a method and the method has implementations in superclasses, the types are inherited from the closest superclass with annotations. We introduced this inheritance rule mostly for convenience and pragmatic reasons. In most cases the types of a method are consistent in a class hierarchy, so a single annotation at the root can serve to type all implementations of a method. Exceptions can always be made by explicitly typing some implementations.

An example to illustrate this point: one might want to annotate the hash method of the Object class with the nonNil type. Since many classes implement their own hashing semantics the hash method is overriden dozens of

times throughout an image. Thanks to the inheritance rule a type annotation on Object's method is sufficient to type all implementations.

2.3 The Type Browser

In Smalltalk IDEs, browsers are traditionally used to navigate and modify the source code in an image. Part of TypePlug is a type browser for Squeak, a browser that enhances a standard browser with some type-specific behavior (the implementation is based on the OmniBrowser [Put]). The type browser has three main features: it integrates type checking, it provides an alternative way of type annotations without changing source code (external type annotations) and it exports types to a distributable form.

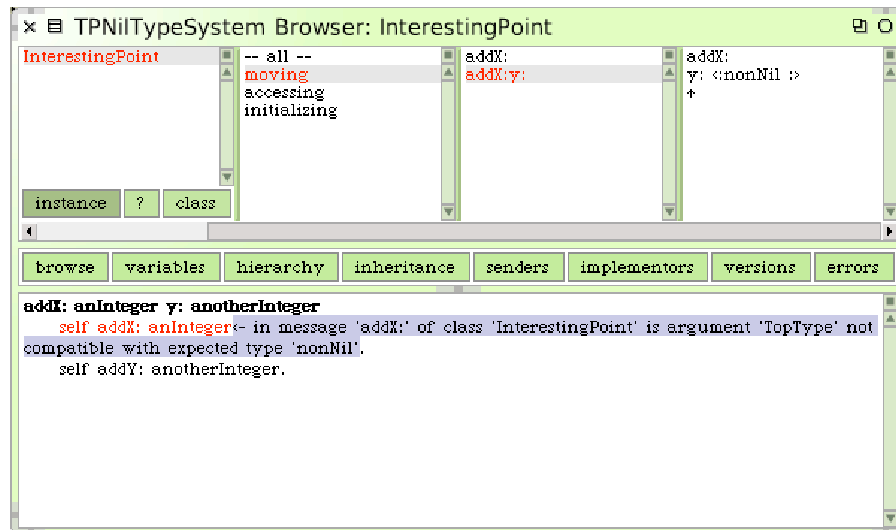

Figure 2.1: The Type Browser

Figure 2.1 shows the type browser in action. A browser is always opened on a specific type system. In this example it is the non-nil type system. The current method shown in a browser is type checked using the type system and type errors are displayed inline, with the offending expression highlighted in red.

Compared to a standard browser, the type browser has an extra panel to the right of the list of methods. This panel shows the method type of the currently selected method, *i.e.*, its argument and an up arrow (↑) as a stand in for the return statement, including types if there are type annotations. Selecting one of these types brings up the type in the bottom (source code) area of the browser.

External Type Annotations

For a type system to support useful type checking it is often necessary to annotate at least a minimal set of methods of the standard Smalltalk classes with types. For example, you may want to annotate the return type of the hash method of Object with a nonNil type. But normally it is not a good idea to add such an annotation by modifying the source of a method. Redefining methods in standard classes such as Object works well locally in an image, but problems arise when the code or the types should be packaged up for distribution, a version control system or any other form of migration between images.

Addressing these issues, the type browsers offers a way to add *external type annotations*, which are separate from the source code of a method. Types can simply be added and modified by selecting one from the types panel and editing the type in the bottom area of the browser. This registers the type with an external cache, it does not modify the source code of a method.

From the context menu of a class, the type browser offers the option to export all types of a class' methods, both external and in-source types. Before the export the user is prompted for a method category name. The types will be exported to a class-side method named exportedTypes of the class, categorized in that method category. All the type information is right there in the source of the exportedTypes method, so it can be easily included in, *e.g.,* a version control system. Importing the exported types of a class is as easy as sending it the exportedTypes method which is supported from the context menu of a class in the browser. Additionally, there is some machinery to create an installer for types over several classes by subclassing the class TPInitializer.

The external type annotation support of the type browser is an important tool for the development and use of pluggable type systems. It allows developers of a type system to distribute a set of types for a Squeak base image, but it also allows other parties to create and distribute type packages for arbitrary type systems.

Chapter 3

Defining a Type System

A type system in TypePlug is defined by first defining types and then the properties of the type system, that is, for the most part, mappings from Smalltalk elements to types as well as operations on types. Since one goal of TypePlug is to enable the creation of new type systems without much effort, a lot of thought was put into keeping the number of properties that have to be defined to a minimum while retaining great flexibility. This section discusses in detail our approach to defining a type system.

Concretely, a new type system is created by subclassing the TPTypeSystem class and overriding some of its methods. As a quick reference and overview, Table 3.1 lists all of TPTypeSystem's methods that can be overridden when implementing a new type system. We will touch on the details of each method during the discussion.

3.1 Types and Their Representation

TypePlug gathers type information from annotations made in Squeak source. Since several type systems can coexist we need a way to distinguish annotations for differing type systems. To achieve this every type system defines a unique *system key* that is used to identify its annotations. The system key is a symbol returned from the class method systemKey.

For the non-nil type system, the key is nonNil, thus an annotation for the single type in the system is <**:nonNil :**>. This is a special case—usually a type system will have more than one single type, and keys will usually have a colon at the end to signal that. For example, the confinement type system discussed in Section 5.2 has the key confine:. In an annotation for that system, the actual type appears as the *annotation value* after the key, *e.g.,* <**:confine: toClass :**>.

The type system obviously must define what its valid annotation values and thus valid types are. It does so by defining a method annotationValueToType :inContext: which returns instances of its types. The grammar of the Persephone annotation framework requires all annotation values to be valid Smalltalk expressions. The annotationValueToType:inContext: method takes as the first argument an abstract syntax tree (utilizing the Refactoring Browser AST nodes) of such an expression. This is a great help for type systems which might have complex annotation values, so they do not need to do their own parsing. The second argument is an instance of TPContext which describes the context of the annotation, *i.e.,* in which method the annotation at hand is located. In a type system, the interpretation of an annotation

systemKey	unique key for this system. *Defined as class method.*
annotationValueToType:inContext:	converts annotations to types. *Abstract, must be overridden.*
is:subtypeOf:	defines the subtyping relation. *Abstract, must be overridden.*
unifyType:with:	defines the type unification operation. *Abstract, must be overridden.*
typeForArray:	maps arrays to types.
typeForBlock:	maps blocks to types.
typeForGlobal:value:	maps global variables to types.
typeForLiteral:	maps literals to types.
typeForPrimitive:	maps primitives to types.
typeForPrimitiveNamed:module:	maps named primitives to types.
typeForSelfInClass:	maps self pseudo-variables to types.
typeForSuperInClass:	maps super pseudo-variables to types.
methodsForMessage:...	customizes the set of methods to be considered when type checking message sends.
transformMethodType:...	transforms methods type before they are used in the type checker.
transformType:...	transforms types before they are used in the type checker.
assignmentTo:...	customizes type checking for assignments
displayClass:	creates a custom string version of classes.
displayMethod:in:	creates a custom string version of methods.

Table 3.1: Methods of TPTypeSystem to override in subclasses

might depend on where it is found.

Types in TypePlug are instances of subclasses of the class TPType. Essentially, implementors of a type system are completely free in making up their type semantics. There are only two technical requirements: the type classes should implement the systemKey and the asAnnotationValue methods. The former should return the key for the type system the type is associated with, the latter should return a string representation of a type as it appears in an annotation.

Note the special case of type systems with just a single type, like the non-nil type system. Those do not need to explicitly define their one type. In code, a single type per type system can be acquired by sending the singleType message to a type system instance.

Regardless of the type system all instances of types carry some extra information, the so called static type information discussed later in Section 3.5. It is important that all the methods of a type system that return types create *new instances* to be returned on every invocation. This is a consequence of the way access to static types is implemented (static types are provided as instance variables of types). For example, the singleType method mentioned above always returns a new instance of the single type of a type system. It can't be a singleton.

The Top Type

One type is predefined and shared by all type systems: the top type. It is assumed to be a supertype of each other type in a type system. Within the TypePlug framework, the top type can appear in many places where a type is expected, and if it does it means one of two things:

1. This can be any type.
2. Nothing is known about this type.

An example for the first meaning is the default argument types for methods; by default, if a method argument is not explicitly annotated with a type, it is assumed to have the top type. The second meaning can be observed in the typing methods discussed in the next section: all of these by default return the top type to state that, *e.g.,* the type of literals (defined by typeForLiteral:) is unknown—unless, of course, defined differently by the type system.

Generally speaking, every typed thing has the top type if it is unannotated or has an unknown type. This may seem like an insignificant implementation detail but the top type is an important device to usefully type check only partially annotated source. With its property of being a supertype of

every other type it guarantees that, *e.g.,* unannotated source type checks safely.

3.2 Typing Elements of Smalltalk Syntax

Type systems can define the types of a range of basic constructs of Smalltalk syntax: types for literals, global variables, primitives, arrays, blocks and pseudo-variables such as self and super. This is achieved by implementing any of the typeFor* family of methods defined on type systems. Below the semantics of these methods are listed. By default all of these methods return the top type which means that the element being typed does not have a type or that its type is unknown. As a general rule when a type is expected, type systems should return the top type from the typing methods if they do not want to or cannot provide the corresponding type.

typeForLiteral: aValue provides a type for literals, *i.e.,* integers, floats, strings, symbols, as well as the constants true, false and nil. aValue holds the literal value being typed.

typeForArray: anArray provides a type for a literal array, appearing in source code either in the form #(1 2 3) or in the form {1. 2. 3}. anArray holds an array of the types of the elements of the array. Note that elements of anArray can be the top type if nothing is known about the respective types.

typeForGlobal: aSymbol value: aValue provides a type for the global variable named aSymbol that at the time of typing has the value of aValue in the system. By far the most common form of globals are classes referenced by name in source code.

typeForSelfInClass: aClass provides a type for the pseudo-variable self, referring to the receiver of the current message. aClass holds the class the current method is defined in.

typeForSuperInClass: aClass is analogous to typeForSelfInClass:, except this is for the pseudo-variable super.

typeForBlock: aBlockType provides a type for a literal block. aBlockType holds an instance of TPBlockType which is composed of the argument types and the return type of the block. Those can be accessed with aBlockType argTypes and aBlockType returnType respectively.

typeForPrimitive: anInteger provides the return type of a numbered primitive with number anInteger.

typeForPrimitiveNamed: aSymbol module: anotherSymbol provides the return

type of a named primitive with name aSymbol in the module anotherSymbol
.

In the non-nil type system, the implementation of the typing methods is very simple. Here are two of the more interesting ones:

```
typeForLiteral: aValue
    ↑ aValue ifNotNil: [self singleType] ifNil: [self topType]

typeForSelfInClass: aClass
    ↑ (UndefinedObject includesBehavior: aClass)
            ifTrue: [self topType]
            ifFalse: [self singleType]
```

Obviously, the type for a literal is always nonNil except when the value is nil. To assess the type of self we need to be slightly cautious since within nil's class, UndefinedObject, we cannot say that self is nonNil. But the same is true for all superclasses of UndefinedObject since their methods could be called from the nil instance (the includesBehavior: method returns true if the argument is the receiver or a superclass of the receiver).

3.3 Subtyping and Unification

The two most important operations a type system must define in our model are a subtyping relation and a type unification operation. Both of these are heavily put to work at the core of the type checking algorithm as described in Chapter 4. In this section we concentrate on their properties rather than their usage.

The subtyping relation is defined by implementing the is:subtypeOf: method which takes two types as arguments. It should return true if the first type is a subtype of the second in the context of this type system, false otherwise. While type systems are free to define whatever strange subtyping relation they please, it should usually be reflexive and transitive to be of practical use. The second argument may be the top type, representing an unknown type. The top type is considered a supertype of every other type of any type system by definition, so is:subtypeOf: is never called with the first argument the top type.

The unification operation creates a type that represents the union of two types. Again, type systems are completely free to define this in any way they require. Generally, the union of two types should be the most specific common supertype, but a type system can also work with an explicit union type that it defines.

In code, the unification operation is defined by implementing the unifyType: with: method which should return the result of unifying the two types passed as arguments. One of these two arguments might be the top type. Furthermore, if the unification of two types is not defined in a type system it can return the top type.

Since the non-nil type system only knows a single type, subtyping and unification are defined uninterestingly simple in it. The nonNil type is a subtype of itself and the nonNil type unified with itself gives the nonNil type again:

```
is: aType subtypeOf: anotherType
    ↑ (self isNilType: aType) and: [self isNilType: anotherType]

unifyType: aType with: anotherType
    ↑ ((self isNilType: aType) and: [self isNilType: anotherType])
        ifTrue: [self singleType]
        ifFalse: [self topType]
```

3.4 Other Elements of a Type System Implementation

To give a type system more control over the type checking process, there are hooks into some central parts of the type checker. Since they are easier to understand when talking about the type checking algorithm, they are discussed in the context of the type checker in Section 4.4.

Finally, type systems get the chance to customize the way classes and methods are displayed in the type browser. The methods displayClass: and displayMethod:in: can be overridden for this. displayClass: takes a class as an argument, displayMethod:in: a selector and a class. By default, a class will be represented by its name and a method by its selector, just as in a standard Smalltalk browser.

3.5 The Static Type System

While Smalltalk is dynamically typed its source code still contains some inherent static type information. For example, the class of an object is statically known if it appears as a literal in the source. Static type information such as this can be very valuable to the type checking algorithm and to any pluggable type system. It is therefore important to capture this information and make it available in a convenient form. TypePlug achieves this with a

static type system which is itself implemented as an ordinary pluggable type system.

Every expression is assigned a *static type*, one of the types defined in the static type system. The static type system defines these types:

- The static self type, the type for the pseudo-variable self
- The static super type, the type for the pseudo-variable super
- Static class types, the types for objects whose class is known
- Static object types, the types for objects whose exact value is known, *e.g.,* literals and globals. Notice that the static object types of a given class are all subtypes of the corresponding static class type, *e.g.,* the static object type for the integer literal 42 is a subtype of the static class type for SmallInteger.
- Static block types, the type for literal blocks

Of course the top type is also part of the static type system, as with any other pluggable type system. If nothing is known about the static type of an expression then its static type is the top type.

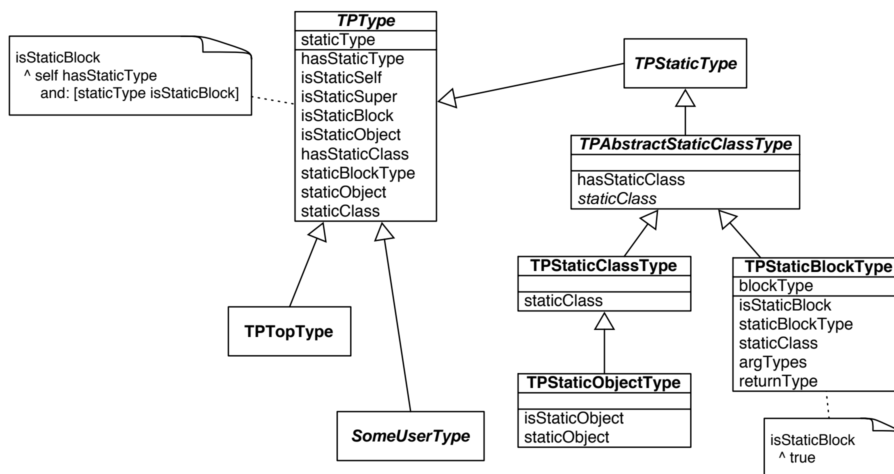

Figure 3.1: UML class diagram of static types (static self and super types not shown)

Differing from a user-defined pluggable type system, the static type system is always present underneath every other type system. In our implementation we make static types available through other type instances, *i.e.,* the static type of an expression is stored on the type of that expression with regards to the current pluggable type system. This can be slightly confusing, but allows a type system implementor to easily access the corresponding static type

wherever he has hold of a type instance of his own type system. Figure 3.1 details the relationships between types and the static type system in the implementation.

Notice that the static block type is a static class type, since the class of a literal block is obviously known. The same goes for the static super type (not shown on the diagram). The static self type, however, is not a static class type—in class hierarchies the pseudo-variable self can refer to any of the subclasses of the class it appears in.

In code, static types are accessed by sending the staticType message to a type instance. A few more convenience methods on types such as isStaticSelf and hasStaticClass are available. The following example shows how one would enforce the argument of a given message (here #collect:) to be a literal block, making use of static types:

```
transformMethodType: aMethodType of: aMethodReference withArgs: anArray
    withReceiver: aType inContext: aContext
    ((aMethodReference methodSymbol = #collect:)
        and: [anArray first isStaticBlock not])
            ifTrue: [self typeSafetyError: 'illegal']
```

The transformMethodType:... hook is called on every message send, so general properties about message sends can be enforced here. We check that the argument's static type is a block type by sending the isStaticBlock message to the argument type (anArray is an array of the argument types passed in the current message send, where types come from the current pluggable type system).

Chapter 4

Type Checking

The heart of TypePlug is its type checking algorithm. The design of the algorithm was largely guided by the goal of making it useful for programmers. With that we mainly mean that it should not be hard work for a programmer to plug in a new type system. Type annotations in pluggable type systems are *optional*, so the programmer should not have to exhaustively annotate code. The algorithm must deliver useful results in the face of an only partly annotated codebase.

From these considerations we derived two main guiding principles for the design of type checking. The type checker must

1. in the absence of explicit type annotations, make as much use as possible of the static type system and infer as many types as possible,

2. and only type check code that has conclusive type information.

The combination of these two principles should make it possible to deal well with a mix of typed and untyped code.

4.1 The Type Checker in a Nutshell

Type checking in TypePlug is applied per method and per type system. The type checker takes a type system and a method as input, checks the method for type safety and returns detailed results about a type error if there is one. The type checking is based solely on static analysis of the source. TypePlug does not support runtime type checks.

Source code is analyzed by traversing an abstract syntax tree (AST) representation of it. A type is assigned to every expression node in the AST. How

the type of an expression is determined depends on the kind of expression it is.

For basic elements of Smalltalk syntax, the checker uses the typing definitions of the underlying type system, *e.g.,* when it encounters a literal such as a string or symbol it will use the typeForLiteral: method of the type system to determine its type. The other elements that will prompt the checker to consult the type system are literal arrays, global variables, primitives, the pseudo-variables self and super as well as literal blocks. The respective methods as described in Section 3.2 will be used.

The type of a message send expression is the return type of the method invoked. This may be either explicitly annotated return types or inferred return types. How the return type of a message send is determined is discussed in detail in Section 4.3.

A programmer can add a cast annotation to any expression in the source of a method. A cast always takes precedence over the type that the type checker assigned to an expression. This means that casts are never type checked and should be used with care.

In addition to the type system at hand the type checker always calculates types for the static type system as well. Recall that every value has a static type which represents the static type information inherently present in the source. Since the static type system is implemented simply as another type system, the static types of expressions are determined completely analogous to the types of any other type system. For example, for an integer literal the result from calling typeForLiteral: on the static type system becomes the static type of the literal just as the current type system's typeForLiteral: is called to determine the type with regards to the current type system.

4.2 Ensuring Type Safety

When the type checker traverses the AST, at certain points type safety is ensured by inserting subtyping checks (for static type checking, the runtime code is not modified). The three points are assignments, return statements and message sends. The behavior of the checker in these cases is described below.

Checking Assignments

When checking an assignment, the type of the expression assigned is checked to be a subtype of the annotated type of the variable—if the variable has

been annotated with a type. If the variable is not annotated it will have the top type, and the assignment will always turn out type safe.

Checking Return Statements

On return statements, the checker simply makes sure that the type of the expression returned is a subtype of the annotated return type of the current method. If the method was not annotated with a return type, no check happens.

Checking Message Sends

Ensuring the type safety of message sends is the most difficult problem in type checking Smalltalk. In general, the class of a receiver of a message is not statically known, so there is usually no way to statically determine which method will be invoked at runtime or if a matching method even exists. Clearly, any statical type checking algorithm relying on partial type annotations must be pragmatic here and make some compromises.

One possible solution to improve the situation is to extensively use type inference to determine the class of message receivers. There are existing approaches to type inference for dynamically typed languages such as [Age96], and with RoelTyper [Wuy] there is even a current effort specific to Squeak Smalltalk (which only infers types of instance variables, however). Implementing a full type inference system to support TypePlug is outside the scope of our project. We do utilize some simple forms of type inference (see Section 4.3) but in the context of a specific pluggable type system, not to determine the class of expressions.

Hence we choose a simple scheme that nevertheless yields useful results. Our approach tries to look up the *set of methods* that could be invoked for a specific message send and involve the whole set during type checking. The static type system is used to make this set as small as possible. Three cases are distinguished based on the static type of the receiver:

1. If the receiver class is statically known, *i.e.,* the receiver has a static class type, the method invoked at runtime can be looked up precisely and the set consists of that one method. Note that static object types, block types and super types all fall under this case since they are static class types.

2. If the receiver has a static self type we need to consider the class the method being type checked belongs to. The set consists of all implementors of the message in this class and its subclasses. It is

necessary to include subclasses because the self pseudo-variable can refer to an instance of a subclass if the method is called from a subclass.

3. If the receiver does not have a static type, *i.e.,* its static type is the top type, we have to resort to a very broad strategy: the set consists of all implementors of a message, *i.e.,* all methods of the message's name implemented in the whole system. This case is the most common, unfortunately.

   A pluggable type system might carry information that can be used to further reduce the set of methods in this case. That is why type systems get the chance to implement their own strategy for this third case. This is done via the hook methodsForMessage discussed in Section 4.4.

Once the set of methods to be considered has been fixed, the actual type check of the message send consists of asserting that the types of the arguments are subtypes of the respective types of the methods parameters. If a method parameter does not have a type it is assumed to have the top type which means that untyped parameters effectively are not type checked. If the set of methods is empty a type error is raised.

This approach as a whole has the desirable property of catching most type errors. But it has the undesirable property of possible raising too many type errors. When unrelated classes have methods with the same name but with different parameter types the all implementors strategy might label a message send as a type error even though at runtime the error would never occur. This drawback becomes less problematic when one considers what actually happens when type checking arguments in a message sends. If a method does not have any type annotations argument type checking is always successful. Only methods that have types on their arguments can provoke type errors. Since we do not expect to operate in a fully annotated code base these questionable type errors should not occur often.

One problem remains: This approach cannot guarantee that a receiver actually responds to a message at runtime, *i.e.,* that a receiver actually implements a method of that name. We only guarantee type safety for the case that a receiver actually responds to a message. It is a fundamental property of the approach that it does not detect failures to understand a message.

4.3 Type Inference

We treat type inference as a crucial tool to enhance the user experience of TypePlug. It spares a programmer from exhaustively annotating source

code with types. However, Bracha states in his position paper on pluggable type systems [Bra04]:

> Type inference should be optional, just like type checking. By mandating type inference as a requirement on a type system, designers of soft type systems fall into the very trap they seek to escape.
>
> A mandatory type inference scheme restricts the expressiveness of the type system in much the same way as a mandatory type checker restricts the expressiveness of the executable language.

This is a very strong statement but we believe that it does not apply to our approach. Our type checker has two specific limited forms of type inference built in, for local variable types and for method return types. Both of those are optional, insofar as the type checker just *tries* to infer types but does not depend on a conclusive result. What is more, the pluggable type systems themselves do not have to care about type inference—it emerges from our model of type systems and does not restrict expressiveness within that model.

Local Variables

While it is possible to annotate local variables with a type it is often unnecessary. In many cases the type of local variables can be inferred by looking at the assignments to it. Consider the following example:

```
frazzleTheFooble: anObject
    | fooble |
    fooble := self defaultFooble.
    anObject hasFooble ifTrue: [fooble := anObject element].
    self frazzle: fooble.
```

When type checking this method using the non-nil type system the type of the local variable fooble will be inferred. Assuming that both the defaultFooble and element messages have a nonNil return type, fooble will clearly be a non-nil value as well. If frazzle: expects a non-nil value the inference has just saved us from a type error here.

Thus the inferred type of a local variable at a given point in a method is: the unification of all the expression types assigned to it up to that point. Notice that this means the type of a local variable may change while the type checking moves forward through a method.

The type checker will only infer the types for local variables without an explicit type annotation. Note that for these unannotated local variables

type safety is not checked. In other words, if the inferred type of a local variable turns out to be the top type this does not violate type safety. This is in line with the philosophy of only type checking annotated things.

Return Types

When the returned result of a message send is used, the type checker needs to make a useful assumption about the return type of the message. Again, Smalltalk's dynamic properties makes it impossible to know which method will actually be invoked at runtime. So we use the same rules as with argument type checking described above to get a set of methods to be considered. The return type of the message is then the union of all return types of those methods.

If a method was not annotated with a return type, its return type is by default considered to be the top type. But to improve the quality of type checking and comply with the requirement to infer as many types as possible, the type checker will try to infer the return type of an unannotated method. Inference basically works the same way as type checking (implementation-wise it is in fact identical): we walk the AST of the method and keep track of the types of AST nodes. The inferred return type becomes the union of the types of all return statements in the method. If the method's last statement is not a return statement, the type defined by the type system for `self` is also part of this union since by default Smalltalk methods return `self`.

When inferring the return type of a method the result can obviously be improved by inferring the return types of message sends within that method as well. In fact the inferencer could drill even deeper and walk the whole graph of method calls to make the result as precise as possible. This is not realistic for performance reasons, so we limit the inferencer to an *inference depth*. It defines how deep the inferencer looks into the call graph.

Increasing the inference depth is very costly in terms of performance, since the total number of methods considered grows very fast. For the type browser, we use an inference depth of just 1 because the user tolerance for lag times is low when just browsing code. While using TypePlug with real world code we discovered that 3 is about the highest tolerable inference depth. We use various caching strategies to improve performance, but in general inferred return types can't be kept in a cache very long because the conditions that lead to a cache invalidation are very costly to detect, especially with a high inference depth.

The Value of Return Type Inference

To evaluate the usefulness and practicality of our approach to return type inference, we made a little experiment. In a stock Squeak image without any explicit type annotations we tried to infer the return type of all methods of classes in the category Files−Directories. This category contains Squeak's abstraction of file system directories, with a total of 8 classes with 210 methods. This serves as an example of a typical small package of user code that one would want to type check.

In the second stage of the experiment, we added as many return type annotations to methods of the Object class as possible. Object is the base class for almost all other classes in Smalltalk; it implements some heavily used methods such as = (equality), class and copy, so annotating those should improve return type inference (recall that overridden methods inherit types from superclasses). With this small set of type annotations we again tried to infer all return types in the Files−Directories class category. The interesting metric of this experiment is the number of return types inferred, *i.e.,* return types other than the top type.

Some more notes on the methodology: every method was examined individually and independently from all the others—no type information gained from inferring previous methods was retained. We used an inference depth of 1, meaning that the type inferencer is allowed to delve one level deep into messages used in an examined method, but not further. This is the default value that is used during type checking in the type browser.

We ran this experiment for both the non-nil and the class-based type system (discussed in Section 5.1). It does not make sense to apply the same metrics to the confinement type system (discussed in Section 5.2) since it can't be used to type arbitrary methods. The results are summarized in Table 4.1.

	Non-nil	*Class-based*
Total methods	210	210
Inferred, without annotations	67	66
in percent	31.9%	31.4%
Inferred, with annotations	73	77
in percent	34.8%	36.7%

Table 4.1: Return type inference for class category Files−Directories in a stock Squeak image, version 3.9 final

In both type systems, for about 31% of all methods the return type could be inferred even without any type annotations present in the image. This shows that return type inference does add significant value to the system as

a whole, since in this package for more than 30% of methods explicit return type annotations are not even needed. We should mention that 39 methods (19% of all methods) do not contain a return statement, so their return type is trivially inferred to be the type of self.

With type annotations for Object, inferred return types are about 3% more for the non-nil type system and 5% more for the class-based type system. These numbers may not seem very impressive, but considering that we added just a few annotations to Object this is a decent improvement. These results indicate that the cumulative effect of additional annotations to other commonly used classes such as numbers, strings and collections should be good.

In summary, the experiment exhibits two findings: return type inference is effective and just a few annotations to common methods improve quality of return type inference, which in turn should improve quality of type checking.

Control Flow

The default algorithm to infer return types is not satisfactory for a very common case: control flow messages such as ifTrue:ifFalse:. We define control flow messages to be those that accept several blocks as arguments and return the result of the evaluation of one of those blocks. Since the return type for control flow messages depends on the arguments it does not make sense for most type systems to annotate them with types and return type inference alone will never give a useful result.

To enable useful type checking of control flow messages they are treated as a special case: the return type of a control flow message is the union of the return types of all the argument blocks. The special case is only applied to control flow messages whose arguments are all literal blocks, so the usual typing rules apply when at least one of the arguments is not a literal block.

What counts as a control flow message is defined by the method controlFlowSelectors of the type system which returns a collection of selectors. By default it consists of #ifTrue:ifFalse:, #ifNil:ifNotNil:, #ifEmpty:ifNotEmpty: and the versions of all of these where the branches are reversed (*e.g.,* #ifFalse: ifTrue). Type systems are free to customize the definition of a control flow message by overriding controlFlowSelectors. But note that the decision about treating a message as a special case is made based only on the name of a message and not on the receiver type, so it is important that the selectors are really uniquely used for control flow. This is a reasonable assumption to make—because control flow messages are such important building blocks

of the Smalltalk language some of them are special cased by the compiler anyway, *i.e.,* you can't implement a method named ifTrue:ifFalse: anywhere else but on the Boolean class.

Block Evaluation

Another kind of message that loosely falls under the category of control flow are block evaluations. Literal blocks in Smalltalk are closures, not interpreted until they are explicitly asked to do so. This is done by sending a block one of the "value" messages, *i.e.,* value, value:, value:value: etc. The arguments in these messages are passed on to the block. The return value of such a block evaluation is the result of the last statement in the block, its type is the return type of the block.

```
[:arg <:nonNil :> | self isSafe ifTrue: [arg] ifFalse: [1]] value: 2
```

In the above example, the return type of the block evaluation is obviously and statically nonNil. But because the default return type inference will not deliver any result for these block evaluations, they are special cased: any "value" message sent to a receiver with a static block type has the return type of this static block type. Also, the arguments to the block are checked for type safety, based on the block argument annotations. In the example above, the argument passed to value: is checked to be of type nonNil.

There are a number of more dynamic evaluation methods on blocks that accept the arguments as an array, for example valueWithArguments: and valueWithPossibleArgs:. These can hardly ever be type checked statically and are thus not treated as a special case of a message send.

4.4 Customizing Type Checking

A number of methods implemented on type systems fall into the category of hooks into the type checker. They influence some central aspects of the type checking algorithm. These three hooks are discussed below.

Method Lookup

The methodsForMessage:... hook can be used to customize the set of methods that should be considered when type checking a message send. As discussed, the default is to consider all implementors of a given message or—in case the class of the receiver is statically known—the actual method that will be invoked. This hook is mostly useful if the types of a type system are

somehow tied to classes, such as with the class-based type system discussed in Section 5.1. In that type system method lookup can be much more precise than the default since we might know the class of a receiver or, at least, its type interface.

The full method signature of the hook is

```
methodsForMessage: aMessageNode receiverType: aType inContext: aContext
```

where aMessageNode is an instance of RBMessageNode representing the message as an AST node, aType is the receiver type (can be the top type if the receiver type is unknown) and aContext is the current context. The method should return a collection of method references (instances of MethodReference) to the methods to be considered for this message. Returning nil, the default, means that no customization should be used for this case. Returning an empty collection is valid, too—this means that no matching method exists and will cause this message send to be considered unsafe.

Transforming Types

There are two hooks used to arbitrarily transform types within the type checker: the methods transformType:... and transformMethodType:.... They exist because type systems may want to postprocess type instances in some manner, after they were created from annotations but before they are used within the type checker.

transformType:... transforms single types, for example the types local variables are annotated with. Its full signature is

```
transformType: aType inContext: aContext
```

where aType is the type to be transformed and aContext is the context in which the transformation happens, *i.e.,* the current class and method. Implementations should return some new type or, if no transformation is desired, return aType unchanged. The default implementation applies no transformation.

An example where this transformation hook is used are the placeholder types in the class-based type system (discussed in Section 5.1). One placeholder type is Self, standing in for the type of the current class. Before the type is used, for example in a subtyping check, it must be transformed into a concrete class-based type, *e.g.,* into the type Boolean if we are in a method of the class Boolean.

transformMethodType:... transforms a whole method type, that is, the argument types and the return type of a method. The transformation is applied

anytime when argument types or return types are looked up in the course of type checking a message send. The full signature is

```
transformMethodType: aMethodType of: aMethodReference withArgs: aCollection
    withReceiver: aType inContext: aContext
```

where aMethodType is the method type to be transformed and aMethodReference is a reference to the actual method. But the transformation can also make use of type information about the message send being checked: aCollection is an ordered collection of the types of the arguments passed in and aType holds the type of the message's receiver. As usual, aContext holds the current context, *i.e.,* information about the method the type checker is being run on. The method type transformation should either return a new method type (an instance of TPMethodType) or return aMethodType unchanged. The latter is the default.

A prime example of an interesting method type transformation is again found in the class-based type system discussed as a case study later in Section 5.1. The type system supports polymorphic messages where the return type depends on the argument types, such as with this simple id: method:

```
id: anObject <:type: MethodParam A :>
    ↑ anObject <:type: MethodParam A :>
```

The untransformed method type here consists of the argument type MethodParam A and the return type MethodParam A. With these method type parameters, the type checker will not be able to type check message sends involving id:. But with a suitable implementation of transformMethodType:..., the class-based type system replaces these type parameters in method types with concrete types gleaned from the concrete argument types.

Assignments

A last hook, assignmentTo:..., is used to customize the type checking of assignments. Its full signature is

```
assignmentTo: aVariableNode ofType: aType expressionType: anotherType inContext:
    aContext
```

where aVariableNode is the AST node representing the variable assigned to, aType and anotherType are the types of the variable and the expression in the assignment respectively, aContext is the current context. The method must return the type of the expression or any transformation of it that might be appropriate in this context. By default it just returns anotherType

unchanged.

This hook is called before the check that the expression type is a subtype of the variable type, so it lets the implementor control precisely whether the subtyping check should be successful or not under the given circumstances. It is especially useful to differently type check assignments to different kinds of variables. aVariableNode is an instance of RBVariableNode (part of the Refactoring Browser AST) and features some useful methods: isTemp, isInstance and isGlobal test whether the variable is respectively a local variable, an instance variable or a global variable (unfortunately, global and class variables are not distinguished). The confinement type system for example uses this to restrict assignments to instance variables and global variables.

Custom Type Errors

Through the discussed hooks type systems also get the chance to raise custom type errors. From within methodsForMessage:..., transformMethodType:... and assignmentTo:... they can signal an exception to label the corresponding message send as type unsafe (transformType:... must not raise an error because it is too basic an operation). The exception is signaled by sending the typeSafetyError: message to the type system, with a string describing the error as the argument. For example:

```
self typeSafetyError: 'message glorp cannot be used in method bla'
```

The applications for custom type errors are manifold. Since during type checkin methodsForMessage:... and transformMethodType:... are invoked on every message sends and assignmentTo:... on every assignment, custom type errors allow one to control how the type safety of message sends and assignments is determined. For example, specific message sends could be disallowed in certain contexts in a type system. The confinement type system discussed in Section 5.2 utilizes this concept to enforce some of its constraints. For example if any method type is encountered that returns a confined type, a type error is raised.

4.5 Problematic Idioms

In a dynamically typed language like Smalltalk there are programming idioms that our type checker simply is not capable of handling—in some cases because of the inherent reflective or dynamic nature of an idiom, in some cases because of properties of our specific approach. These idioms are discussed below.

perform: family of methods

The perform: method is used to send a message to an object with the message name determined at runtime. It takes a message name as its argument and sends that message to the receiver (the variants perform:with:, perform:with: with: and so on also take arguments to be passed along with the message). The return type of a perform: message depends on the value of the argument which in general is not known statically, so there is no way to reasonably type check a perform: message, its arguments and its return value.

To get around this problem the return type of a perform: message send can be "declared" using a type cast, in case the programmer is reasonably sure about the return type. In some cases perform: can also be replaced by an equivalent block idiom if type checking is really important. Instead of passing selectors around for a dynamic message, as in

```
self perform: #dynamicMessage: with: argument
```

use blocks equivalently:

```
[:receiver :arg | receiver dynamicMessage: arg] value: self value: argument
```

Since block evaluation is a special case in the type checker, the latter can be type checked.

doesNotUnderstand:

The doesNotUnderstand: idiom—often abbreviated with *DNU*—is an even bigger problem for our type checking algorithm. The doesNotUnderstand: method allows classes to intercept runtime errors occurring when a message was sent that is not understood by the receiver. This means any object possibly accepts a message sent to it even though its class does not explicitly implement a method of the same name. This is a problem because type checking by default assumes that it can find all implementors of a given message name. To avoid false type checking results, the only recommendation we can make is to avoid the DNU idiom in areas of code that should be type safe.

Instance variable initialization

At the instantiation of a new object its instance variables hold the value nil. It is up to the programmer to initialize instance variables to a valid value, using any of several common idioms, *e.g.,* after object creation calling a setter per instance variable or one setter for all of them. This creates an issue with typed instance variables when the uninitialized value nil is

not of the expected type. The type checker checks assignments to typed instance variables, but it does not check whether an instance variable has been initialized when it is used. Given the many ways to initialize instance variables this would be hard to do.

This is obviously a problem for the non nil type system, for example. If an instance variable is typed nonNil we really do not want it to be used uninitialized. The recommendation here is to always use the initialize idiom when working with typed instance variables. The method initialize is guaranteed to be called on any object right after its instantiation, therefore it is a good place to initialize instance variables to a type safe value. The type checker does not treat initialize specially, but type errors in assignments to instance variables in initialize will of course be caught. This discussion is also valid for typed class variables; initialize can also be implemented as a class side method.

Exceptions

TypePlug currently does not model exceptions. In general, every message send in Smalltalk could result in an exception being signaled. The concept of exceptions is fortunately orthogonal to our notion of type safety. For our purposes, if an exception is signaled it simply means that the execution of a method stops at that point, it does not in any way affect the types of variables or other expressions. The type checker can also easily deal with Smalltalk's exception catching mechanism. In code, exceptions are not caught by using a special syntax but by sending the on:do: message to a block, so the usual typing rules for blocks and messages apply. TypePlug simply does not deal with exceptions at the moment, not diminishing the usefulness of type checking.

But there is one kind of exception that is problematic. Smalltalk's runtime type checking consists of checking whether a object understands the messages that it receives. If a message is not understood at runtime, an exception is raised. TypePlug's type checking cannot guarantee that messages are understood by their receivers. As explained before, this is an inherent problem with statically type checking dynamically typed languages.

Chapter 5

Case Studies

5.1 A Class-Based Type System

On top of the TypePlug framework we were able to implement an expressive class-based type system, sporting many features of modern statically typed languages. It supports generic types, polymorphic methods, type unions and typed blocks. The syntax for type annotations in this system is summarized in Table 5.1. This is powerful enough to meaningfully annotate most Smalltalk code with class-based types.

Class ::=		
	ClassName	
	Self	
	Instance	
Type ::=		
	Class	(Simple type)
	Class class	(Class-side type)
	Type | *Type*	(Union)
	Block args: {*Type**} return: *Type*	(Block type)
	Class (*ParamName*: *Type*)+	(Generic type)
	Param *ParamName*	(Type parameter)
	MethodParam *ParamName*	(Type parameter)

Table 5.1: Grammar for class-based types

Simple Types

The type of instances of a class is simply the name of that class, *e.g.,* booleans and integers have the types Boolean and Integer respectively. Now,

classes themselves (as opposed to their instances) have a type, too. Such a class type is formed by appending class to the name of the class, *e.g.,* Boolean class for the type of the class Boolean.

Other simple types are the placeholder types whose concrete value depends on their context. The Self type is the type of an instance of the class where it is used. When the Self type is used in an *instance side* method of the class Boolean, it refers to the type Boolean. When it is used in a *class side* method, it refers to the type Boolean class. To refer to the type of instances of a class from a class side method, the type Instance is used. So the Instance type used in a class side method of the class Boolean stands for the type Boolean.

Subtyping

The usefulness of a class-based type system such as this one depends largely on the definition of subtyping. Smalltalk code in general is not well suited to be typed using classes. For example, classes might override methods while breaking assumptions about types made in the superclass, and other classes might "delete" methods defined in superclasses. All of these things function well considering Smalltalk's dynamic properties, but are not easily compatible with the notion of subtyping.

Furthermore, it is important keep the requirement in mind that the programmer should not have to make a whole lot of annotations to get useful type checking results. In other words, a programmer should not have to annotate all the methods of the class Integer just to be able to use the type Integer meaningfully.

On the basis of these considerations we define a subtyping relation that is based on the *type interface* of a class. We define the type interface of a type to be the set of its class' method types that do have some type annotation. Table 5.2 shows some example code to demonstrate this. The type interface of the type Fruit is the method type mixWith: Fruit ↑ Array E: Fruit plus of course any method types inherited from superclasses (a method type is a method's name with its argument and return types). isVegetable is not part of the type interface since it does not have any type annotations.

The definition of our subtyping relation consists of two clauses. The first clause forms the basis for the relation with a standard definition for structural subtyping:

1. Type A is a subtype of type B if the type interface of B is a subset of the type interface of A and for all method types mB of the interface B and the corresponding method type mA of the interface A it is true that: the argument types of mB are subtypes of the respective argument

types of mA and the return type of mA is a subtype of the return type of mB.

Isolated, this clause is a standard subtyping relation from the literature (*e.g.*, [Pie02, page 182]) based on structural interfaces, using the usual contravariant subtyping for arguments. Note that method types of an interface might be incomplete in the sense that some arguments or the return statement of the method might not be annotated with a type. The top type will be substituted for these missing types, and since the top type is considered to be a subtype of itself, this definition works well with partial method types.

According to this first clause, in the fruit example from Table 5.2 it is clear that both types Apple and Orange are subtypes of the type Fruit; their type interface contains the (inherited) mixWith: method with types. Also, more surprisingly, the Apple and Orange types are both subtypes of each other—their type interfaces are identical since they both implement an annotated color method.

There is one obvious problem with subtyping based only on the above first clause: if this were the complete subtyping rule then every simple type would be a subtype of every other simple type in a codebase without any type annotations. This would not be useful, which motivates the second clause of our definition:

2. If the type interface of a type A is identical with the type interface of the type based on its superclass (the class A does not add or change any type annotations), then the type A is its only subtype and no other type is a subtype of it.

This means that for types based on classes that do not define any typed

```
class Fruit subclassing: Object
    mixWith: aFruit <:type: Fruit :>
        ↑ (Array with: self with: aFruit) <:type: Array E: Fruit :>
    isVegetable
        ↑ false

class Apple subclassing: Fruit
    color
        ↑ (Color red) <:type: Color :>

class Orange subclassing: Fruit
    color
        ↑ (Color orange) <:type: Color :>
```

Table 5.2: Fruit bowl code

methods, subtyping is based on type identity only.

This rather unusual subtyping definition has some interesting implications. First, thanks to the second clause, it works well with untyped classes. Second, it puts a lot of responsibility into the hands of the programmer who makes type annotations. To get subtyping relations in a class hierarchy, type annotations must be added to all the classes in the hierarchy that should participate in the subtyping relations. Also, subtyping relations can exist outside of class hierarchies because of matching type interfaces. In a way, type interfaces formalize the Smalltalk practice of having objects support an informal "protocol".

Generic Types

Generic types are types parameterized by at least one named type parameter. A typical example of generic types are collections, *e.g.,* the type of an OrderedCollection. This type has one parameter referring to the type of the elements of the collection. This parameter is named E, an OrderedCollection of integers then has the type OrderedCollection E: Integer. In a generic type parameter names are followed by a colon and then the type it should be bound to.

In the context of a class with a generic type, type parameters can be used just like any other type: the type Param E refers to the type that the parameter E takes in a generic type.

We have not yet explained how a type is declared to be generic. Since we did not want to change the way classes are declared there is no explicit declaration of type parameters, instead types implicitly become generic under certain circumstances. Namely, a type has all the type parameters that appear in the type interface of the class its based on. For example, the type of an OrderedCollection is generic because the parameter Param E appears in the methods of the class OrderedCollection, *e.g.,* as the argument type of the add: method.

More than one parameter can of course appear in the type interface of a class—in that case a typed based on such a class is a generic type with several parameters. An example is the type Dictionary which has a parameter K for the type of its keys and V for its values. A dictionary mapping symbols to strings then has the type Dictionary K: Symbol V: String.

To distinguish generic types from simple ones and for better usability, the class-based type system customizes the way classes are displayed in the type browser (by overriding the displayClass: method). Type parameters found in the interface of a class are listed after the class name, such that *e.g.,* the

class Dictionary appears as Dictionary <K, V> in the browser.

Polymorphic Methods

Apart from type parameters with class scope as used in generic types, our class-based type system also supports type parameters with method scope. Methods making use of such type parameters are *polymorphic methods*. The quintessential and most simple example of a polymorphic method is the id: method which takes an argument and does nothing but return that argument again. To type id: we can give its argument the type MethodParam A, a type parameter named A with method scope. As the return type of id: we use the same type parameter again, MethodParam A. This expresses that we expect the return type of id: to be the type of its argument.

A more interesting example is the method ifTrue:ifFalse: of the class Boolean. Table 5.3 shows the method with type annotations. This also shows the usefulness of polymorphic method support and that it is needed to type some crucial innards of Smalltalk such as this control flow method.

```
ifTrue: aBlock <:type: Block args: {} return: MethodParam R :>
ifFalse: anotherBlock <:type: Block args: {} return: MethodParam R :>
    ...
    ↑ (...) <:type: MethodParam R :>
```

Table 5.3: Method ifTrue:ifFalse: of class Boolean with type annotations

During type checking of code invoking a polymorphic message, the type checker infers the value of all type parameters from the type of the concrete arguments passed (in the type system implementation, this is achieved via the hook transformMethodType:...). As an example consider the following method:

```
bitFromBoolean: aBoolean <:type: Boolean :>
    ↑ (aBoolean ifTrue: [1] ifFalse: [0]) <:type: Integer :>
```

The method bitFromBoolean: will type check safely in our class-based type system. The type checker will discover that both blocks have the return type Integer. When checking the message ifTrue:ifFalse: sent to a variable of type Boolean, it will thus infer that MethodParam R in this particular message is of type Integer. Accordingly, the return type of the whole message will be Integer which is the expected return type of the bitFromBoolean: method.

Block Types

It is common in Smalltalk to pass around blocks as for example demonstrated by the ifTrue:ifFalse: method in Table 5.3. Therefore we must be able to express the type of a block. A block type is simply the sequence of types of the arguments it accepts plus its return type, *i.e.,* the type of a block's result when it is evaluated. The syntax used for a block type is Block args: { Type1. Type2. ...} return: ReturnType.

Accordingly, the block [true] has the type Block args: {} return: Boolean since it accepts no arguments and returns a boolean. A slightly more complex example is the block

[:a1 <**:type: Integer :**> :a2 <**:type: Integer :**> | a1 + a2]

which has the type Block args: {Integer. Integer} return: Integer.

Union Types

A last special kind of type are union types. The situation often arises that a certain variable could hold both an instance of some class and nil, Smalltalk's null reference. These situations can be handled by a union type. A union type is formed by stringing together other types with a |, *e.g.,* a type that can be both a string or nil is String | UndefinedObject (where UndefinedObject is the type of nil). It is possible to form a union type of more than two types.

With respects to subtyping it is interesting to consider the type interface of a union type: it is the set union of the type interface of all participating types.

The type system's union type is mostly a proof of concept implementation to date, it does not address some of the issues that come with union types. It does not intelligently resolve conflicts in method types between the participating types. Also, there is no way to dissect a union type again after its entered the system by, *e.g.,* branching on the actual type of a variable.

5.2 A Confinement Type System

As an example very different from the non-nil and class-based type systems we created a confinement type system. This type system implements a very specific kind of confinement: confined instance variables. References to mutable objects such as a collection are often considered to be private to a class when they are stored in an instance variable; these references should not

be shared with other classes. The confinement type system can guarantee that such confined instance variables do not leak from their class and thus are not modified outside their class. In Smalltalk, all instance variables are always private, encapsulated by instance. In a way, this type system expresses an extended form of privateness for instance variables.

The confinement type system demonstrates how TypePlug enables type systems with fairly complex semantics to be implemented succinctly. The implementation consist of only 4 classes with a total of about 110 lines of code.

Working with Confined Instance Variables

In this section we present a usage scenario for the confinement type system. As an example we consider a class Directory that defines a file system directory containing a number of files stored in the instance variable files.

```
Object subclass: #Directory
    typedInstanceVariables: 'files <:confine: toClass :>'
    ...
```

Since the instance variable files holds a mutable list of files, it better not be modified outside the class. This constraint is expressed by annotating the variable with <**:confine: toClass :**>.

We assume the class has an accessor method files that simply returns the instance variable files. A user of this accessor is the class Archive representing a compressed archive containing several files (*e.g.,* a ZIP file). The class has a method addDirectory: to add all files of a directory to an archive.

```
Directory>>files
    ↑ files

Archive>>addDirectory: aDirectory
    members
        ifNil: [members := aDirectory files]
        ifNotNil: [members addAll: aDirectory files]
```

When the type checker is run on the addDirectory: method it complains on the first files message send with the error message "method is returning a confined value of Directory". Through return type inference it discovered that the result of the message send might have a confined type, which is not allowed in a class other than Directory. The potentially damaging side effects of such a leak of a confined instance variable can be observed in this example: adding a directory to a new archive sets its members instance variable to the same collection that contains the directory's files, and later

calls to addDirectory: will thus modify both the archive and the first directory added to it. One could argue that the code in addDirectory: is bad style and that members should be initialized to an empty collection anyway—but the type system is meant to detect exactly these kinds of oversights.

Confinement violations such as the one detected here can usually be fixed by operating on copies of the confined values. Here, one would have the files accessor return a copy of the instance variable:

```
Directory>>files
    ↑ files copy
```

This eliminates the type error since files now has a neutral return type. The reason why this works is a bit more subtle than it may appear at first sight and needs an explanation about how the return type is determined in this case. By default, the return type of a message sent to a receiver with a confined type is considered to be that same confined type. The only exception are methods where the static type system can prove that the return type is not a static self type. The rationale behind these rules is that any method could be returning self, a reference to the receiver which of course keeps the receiver's confinement restrictions.

But some methods such as copy inherently always return a new and thus unconfined reference, even though this might not be statically provable. To allow such methods to be tagged accordingly by the user the confinement type system knows about the type unconfined. In our directory example, the copy method's return type was annotated as **<:confine: unconfined :>**. The unconfined return type will always override any confinement restrictions that might have been inferred for a method.

There is another important use case for the unconfined type, in casting. Consider the following method from the Directory class:

```
Directory>>copyAllTo: aDirectory
    aDirectory addAll: files <:castConfine: unconfined :>
```

The confinement type system forbids the usage of confined references as arguments in a message send. This is necessary because the type checker has no way to tell whether anything unwanted happens to a confined reference once it was passed to a method. But since there are legitimate and safe uses of confined references as message arguments, they can be explicitly cast to the unconfined type. This is a strategy that many type systems implemented with TypePlug might adopt: better generate too many type errors to be safe, but let the user control these type errors through casts.

Confined and Unconfined Types

This type system has two kinds of types, confined types and unconfined types, instantiated respectively by the <**:confine: toClass** :> and <**:confine: unconfined** :> annotations. A confined type is associated with at least one class, which is the only class within which the type is allowed to appear (a confined type can be associated with several classes as a result of unification).

Confinement can be applied to instance variables and class variables. A confined instance variable means that it may only be accessed from instances of the respective class. Analogously, a confined class variable may only be accessed from the respective class (*i.e.,* class-side methods). Our notion of confinement honors inheritance as well, so confined variables may also be accessed from subclasses of the class that defines them.

Different from the non-nil and class-based type systems, the types in this type system are not applicable to all annotation places. <**:confine: toClass** :> should only be added to instance variables and class variables. This restriction is actually enforced by raising an error when a confined type is created anywhere else. This is done in the annotationValueToType:inContext : method of the type system which has access to information about the annotation place through the context.

Unconfined type annotations are in theory allowed everywhere, but they are only useful for two purposes: as return types and in casts. An unconfined return type signals that the method always returns something other than a reference to the receiver. The quintessential example is the copy method defined on the Object baseclass—it should be annotated with an unconfined return type to make sure copies of confined references are not considered confined themselves.

Casting an expression to an unconfined type negates any confined type the expression might otherwise have, leaving the expression with a neutral type. Since passing around confined references is by default forbidden but still necessary and safely possible at times, such casts serve to cancel out type errors when passing around confined references.

Subtyping and Unification

For the implementation of this confinement type system, the usually very important subtyping relation and unification operation are quite uninteresting. We are not interested in subtyping checks, so we consider any two types to be mutual subtypes (simply returning true from is:subtypeOf:). Unification for confined types is defined as follows:

- The unification of a confined type and the top type is the confined type.
- The unification of two confined types is a new confined type associated with all the classes the two types to be unified are associated with.

Through unification, confined types associated with more than one class can emerge. This can happen during the return type evaluation for a message send and most of the time immediately results in a type error. The current class will be compatible with at most one of the several associated classes, *e.g.,* a confined type associated with both the classes Directory and Archive is not allowed anywhere because within Directory it would be a leak from Archive and vice versa. Exceptions are rare but occur when subclass relationships between the associated classes exist, *e.g.,* a confined type associated with Directory and Archive would be allowed within Archive if that were a subclass of Directory.

Unconfined types are ignored in unification, *i.e.,* the result of a unification involving an unconfined type is always the other type (or the top type if both types being unified are unconfined). The purpose of unconfined type is to counter confined types right at the point where an unconfined annotation was placed, so there is no point in caring about or propagating them after that.

Keeping References Confined

What is left to explain is how exactly we make sure that confined references do not leak from a class. In general, references only leave the bounds of a class through message sends, so most violations of "type safety" in this type system happen there. There are two conditions that make a message send unsafe:

1. The return type of the message send is a confined type associated with a class A not compatible with the current context class B (meaning A and B are not identical or B is not a subclass of A).
2. One of the arguments has a confined type.

Examples of both of these type errors were given during the introductory examples of this section.

Another kind of message send needs special attention: those where the receiver has a confined type. To be on the safe side, we must assume that a message send to a confined receiver returns a reference to self, meaning that the return result is confined as well (except if the return type is explicitly unconfined). For some cases, return type inference can statically determine

(using the static type system) that the return type of a message send is definitely not self. But unfortunately the framework and hooks into the type checker do not currently offer a way to make decisions based on the inferred return type of a method, so we cannot single out this special case.

There is another potential source of type errors: assignments. When an confined reference is assigned to a variable, the variable would have to become confined if it is not already. But our type checking algorithm does not offer the option to change the type of, *e.g.,* an instance variable during type checking. So we explicitly forbid assignments where a confined reference is assigned to an untyped instance variable, class variable or global variable. These restrictions on message sends and assignments are implemented within the transformMethodType:... and assignmentTo:... hooks, raising custom type errors. In the case of assignments, the error message will suggest to declare the left hand side variable to be confined as well.

Drawbacks of the Approach

The type system works remarkably well in confining instance variables, partly by being conservative (*e.g.,* considering the result of messages sent to a confined reference to be confined). But there is one caveat: this confinement type system relies on the type inference capabilities of TypePlug. Whether a method returns a confined reference or not is determined solely by inference of return types, not by explicit type annotations as in, *e.g.,* the non-nil type system. But return type inference is mostly just a tool to enhance the quality of type checking; it is not supposed to (and, in fact, cannot) work for all methods.

In this type system, problems arise if a confined reference enters a method through multiple levels of indirection. Return type inference only walks the tree of message sends to a fixed inference depth—if some method past that depth returns a confined reference that reference is not considered during return type inference. Here is an example to illustrate this point.

```
Directory>>files
    ↑ files

Directory>>firstFiles: anInteger
    ↑ (self files size > anInteger)
        ifTrue: [self files first: anInteger]
        ifFalse: [self files]

Device>>directorySneakPeeks
    ↑ directories collect: [:dir | dir firstFiles: 3]
```

Provided the instance variable files of Directory is confined, it is leaking from Directory>>firstFiles: and thus violates confinement in Device>>directorySneakPeeks. But if we are type checking directorySneakPeeks with an inference depth of just 1 no type error is detected, since the type checker will not try to infer the return types of methods within firstFiles: and not find out that the files method returns a confined reference.

The way to minimize missed confinement violations due to this problem is obviously to use a large inference depth. Increasing the inference depth does however slow down type checking considerably. With the current state of TypePlug a depth of about 3 is anecdotally the highest tolerable value. We recommend to do development using a low value and occasionally let the type checker run on all methods in relevant packages with a higher inference depth (some support for this is provided with the TPImageChecker class). This occasional exhaustive type checking could for example be part of a build process.

There is one other caveat: the confinement in this type system is based on classes not on instances. This means that it can't prevent two separate instances of a class from sharing confined references. There is not much that can be done about this, since bounds between instances cannot generally be determined statically. Runtime checks, complementing TypePlug's static type checking, would be necessary to implement instance-based confinement.

Chapter 6

Related Work

Pluggable Type Systems

JavaCOP [ANMM06] is a program constraint system for implementing practical pluggable type systems for Java. The authors present a framework that allows additional types to be added to Java source code, based on Java's annotation facilities. They then define a declarative, rule-based language that can express rules as constraints on AST nodes, making use of the information from annotations as well as from Java's static type system. These rules form the semantics of pluggable type systems and can be enforced by a type checker that hooks into the compile process.

Annotations in JavaCOP are similar to TypePlug's, but are restricted to class and variable declarations by the Java language, so something like our casts on arbitrary expressions cannot be supported. The way type systems are implemented with rules in a domain-specific language is very different from TypePlug's type system model. The rule language enables very fine grained control over type checking. The authors prove the validity and versatility of this approach by implementing an impressive number of pluggable type systems, ranging from confined types to reference immutability and checks of the kind usually done by coding style analyzers. On the other hand, our approach with a rather fixed type checking algorithm and customization is arguably a bit simpler and results in simpler implementations at least for some applications. For example, our non-nil type system is extremely simple with about 30 (mostly very trivial) lines of code while JavaCOP's equivalent presumably needs several non-trivial rules.

On a closer look, there are some conceptual commonalities between the two approaches to modeling type systems. TypePlug's hooks to customize type checking of message sends and assignments serve basically the same purpose as JavaCOP's rules on assignment and method calls. One could

argue that the simplicity of Smalltalk's syntax works in our favor here, since it replicates a significant part of Java's syntax (such as `if` statements or `for` loops) with message sends. Another commonality is that JavaCOP lets the user define custom subtyping relationships pretty much the same way TypePlug does.

With Java's static type system, JavaCOP does not face some of the challenges that TypePlug does. Pluggable type systems implemented in JavaCOP have complete information about Java types at their disposal, instead of just the limited information from our static type system. What is more, JavaCOP can precisely look up the method types at method call sites, which is one of the main areas where our approach must make pragmatic compromises.

But the most important difference is that TypePlug uses type inference to improve type checking, which among other things reduces the need for exhaustive annotating. As an example, in JavaCOP methods always have to be explicitly annotated with a non-nil type, even when that return type could be statically proven to be non-nil anyway.

Non-null types for C# and Java were researched by Fähndrich and Leino [FL03]. They describe a model that allows any type declaration in a statically typed object-oriented language to be optionally extended with a non-null type. A custom type checker guarantees type safety based on these non-null annotations. An implementation for C# is presented, but the model is easily transferable to Java since the two languages have many conceptual parallels.

The authors put a special focus on non-null instance variable initialization, a problem that TypePlug does not solve satisfactory (see Section 4.5). The problem is to make sure that instance variables declared non-null are initialized to a non-null value at some point during the construction of an instance. In this work, a special "raw" family of types is proposed for partially initialized instances within constructors. Accessing instance variables of a raw type may yield null, regardless of whether they have been declared non-null or not. The type checker takes raw types into account and can guarantee that every path through each constructor returns with fully initialized non-null instance variables, with just a few restrictions on expressiveness within constructors.

A similar model might work in the context of TypePlug, but in Smalltalk additional means would have to be devised to overcome the difficulty that constructors are not syntactically different from methods.

Fleece [All06] also explores the notion of pluggable type systems, but explicitly for dynamically typed languages. As an example of a dynamically typed languages the report introduces and defines $\mathcal{R}_{sub}$, a subset of the

Ruby programming language, which is then used throughout the report. It develops the notion of annotators that automatically add and propagate annotations (types) on nodes of the AST of a program. A special case of an annotator is the programmer who can manually add annotations to a program. Fleece's handling of annotations correspond in many aspects to TypePlug's. Automatic annotators play the role of the inference part of TypePlug's type checking.

Due to the lack of a framework to add annotations to arbitrary AST nodes in Ruby source code, part of Fleece is a special editor that represents an AST visually in a 3D view and lets the programmer add annotations. This special editor must be used to take advantage of Fleece; it is not integrated into the standard environment of a language like TypePlug with its type browser. When the user modifies code in this editor, Fleece only reevaluates the annotations of AST nodes that were affected by the change—that is, the result of running annotators (type inference, essentially) are retained, resulting in good performance characteristics, something that TypePlug could benefit from as well.

The report then defines a generic type checking algorithm that verifies a program based on annotations. At the point of type checking, type inference in the form of annotators has happened already, so in contrast to TypePlug the phases of type inference and checking are strictly separate. Annotators and type checkers must be provided by the user to form type systems, but not many details are given about implementation aspects of these. Presumably, this is similar to TypePlug's model of a type system, but with a cleaner separation of annotation/inference (typing syntax elements, unification) and type checking (subtyping, type checker hooks).

Compared to TypePlug, Fleece is both more general and more restricted. It is more general, because it is independent of the language $\mathcal{R}_{sub}$; it copes with any other language that can be expressed in the grammar form that it expects. However, it is more restricted because it has not been shown to handle a real dynamically typed language. $\mathcal{R}_{sub}$ is a very restricted form of Ruby with, *e.g.,* no inheritance or support of the standard library. The feature that most sets TypePlug apart is the ability to pragmatically work with an actual full implementation of Smalltalk.

Optional Typing for Dynamically Typed Languages

The most interesting recent work on optional typing is on *Gradual Typing* [ST06]. It is a theory that combines static and dynamic typing, based on optional type annotations similarly to our work. It introduces a simply-typed lambda calculus enhanced with a dynamic type ?, comparable to our

top type. For this calculus it defines static as well as runtime type checking semantics. The static type checking mainly differs from the type systems we describe in the interpretation of ?: it is regarded not as a top type (*i.e.,* a supertype of every other type) but as an *unknown* type. ? is allowed in places where a concrete type is requested by an annotation. The safety of these implicit coercions from ? to some concrete type is checked at runtime. TypePlug does not do runtime type checking, so its interpretation of the top type is a good way to increase the amount of potential type errors found statically.

From a programmer's point of view, the difference between ? and the top type affects the meaning of a type annotation. In the class-based type system, a type annotation on a variable means that this type must be statically guaranteed. Gradual typing only statically guarantees an annotated type for code paths where the types to be checked against annotations are known. For example, this means that a method argument annotated with some type in gradual typing statically accepts ?, while in our class-based type system the same with a top type is a (static) type error.

However, nothing stops a type system implemented in TypePlug from adopting the semantics of ? for the top type. It is as simple as defining the top type to be a "subtype" of any other type, with the consequence that less type errors are detected statically. If TypePlug supported runtime type checking, it could serve as a vehicle to implement the full semantics of Gradual Typing.

Gradual Typing does not currently integrate type inference in any way.

Typing Smalltalk

The most recent effort at conceiving and implementing a type system for Smalltalk was *Strongtalk* [BG93]. Earlier, *TS* (Typed Smalltalk) [Gra89] created a type system for Smalltalk. Both of these support optional typing, but they provide just a single type system, not addressing pluggability.

Strongtalk is a fully functional Smalltalk implementation, recently released as open source software [Str]. Similar to TypePlug's class-based type system, Strongtalk's supports generic types, polymorphic messages and union types. However, subtyping is not based purely on structural matching but on declared interfaces, improving localization and understandability of type errors. Unlike TypePlug, Strongtalk comes with type annotations for the whole base image, so its type system was proven to be applicable to a large body of Smalltalk code and to be of practical use.

The main motivation of TS in creating a type system for Smalltalk was op-

timization. Its main goal is to provide type information to an optimizing compiler, *e.g.,* to support static method lookups. Like Strongtalk and our class-based type system, TS supports generic types, polymorphic messages and union types, which shows that these features are required to meaningfully type Smalltalk code. TS also employs a form of type inference to infer method types of untyped methods.

Chapter 7

Conclusion

In this thesis we presented our framework for pluggable type systems for Smalltalk. We described a model for type systems that allows Smalltalk methods to be checked with a generic type checking algorithm. We introduced concrete type system implementations adding real value to a programmer's toolbox.

In Chapter 1, the introduction, we mentioned two main challenges for pluggable type systems in a dynamically typed language: coping with optional types and static type checking in the absence of a native static type system. Let's revisit how we dealt with these challenges.

Optional type annotations are problematic because they tend to "spread" once just one type annotation was added to code. We countered this problem by employing type inference, rendering some explicit type annotations superfluous. Optional types also mean that a type checker must be able to handle partially typed code. Our use of type inference as well as the top type make our type checking algorithm useful still when facing missing explicit type information.

Dynamically typed languages such as Smalltalk do not lend themselves well to static type checking. We devised a pragmatic way to handle dynamic method lookup statically that gives reasonable type checking results for arbitrary type systems and can be refined by a type system implementation. We treat as special cases some important idioms such as control flow messages and offer casts as a device to get typing for inherently dynamic language features such as reflection.

Having dealt with these challenges, TypePlug offers researchers a simple way to implement and experiment with new type systems and it provides programmers with a tool to check properties of their code.

7.1 Future Work

Type Checking

- While TypePlug tries to push hard, static type checking can only go so far with dynamically typed languages. *Runtime type checking* could complement the static type checking to make type systems safe. The Persephone framework we use for annotations offers ways to enhance methods with type checks, so further work in this area will be bothered more with conceptual rather than implementation issues.
- An important aspect of type checking that can be further refined is the *method lookup* for receivers with statically unknown types. Currently, the algorithm by default considers all implementors of a given selector. For selectors with many, possibly completely unrelated implementors this often does not yield useful results. A possible strategy to improve results is the user-driven definition of lookup space for implementors, *e.g.,* users could exclude implementors in packages they know are not used in the code being checked. Further improvements could be achieved by making use of type information from the class-based type system, even when type checking in other type systems.
- TypePlug so far does not model *exceptions* in any way. It could be helpful to invent a way to declare exceptions thrown in a method, for documentation purposes but also to use the information during type checking and potentially implement Java-style checked exceptions.

Type Inference

- It would be interesting to combine TypePlug's type inference with a *dedicated third-party type inferencer* such as RoelTyper [Wuy]. Based on heuristics, RoelTyper infers the types of instance variables and is reportedly very fast at it. The results of an inferencer like RoelTyper could be used to seed some type annotations or they could be directly integrated into the type checker.
- In the current state of implementation, type inference is what can slow down the type checking algorithm considerably, to the point where it is unbearably slow for complex methods. Currently, inference depth must be kept low to attain acceptable performance, which decreases the usefulness of type inference. Work on performance of type inference and type checking can considerably improve the user experience of TypePlug.

Type Systems

- To further evaluate and evolve our model of pluggable type systems, *more type systems* should be implemented. Concrete ideas that have been floated are extensions to the confinement type system (*e.g.,* confinement over a range of classes, package confinement) and a string encoding type system (since strings do not explicitly carry information about their encoding in Squeak).
- As of now, all pluggable type systems implemented with TypePlug are completely shielded from each other. However, similar to how every type system has access to information from the static type system, it could be useful to allow type systems to *share type information* between each other. For example, many type systems could probably benefit from the additional information of class-based types.
- The *class-based type system* should be explored further. The pragmatic definition of the subtyping relation based on type interfaces works reasonably well for some medium-sized examples, but it remains to be shown how useful it is for large codebases. A set of type annotations for core Smalltalk classes such as numbers, strings and collections should be created to serve as a basis for useful type checking with the class-based type system.

Appendix A

Theoretical Background

Types and type systems form a broad field in computer science with a vocabulary that might not be consistent over time and across subfields. This appendix gives a brief overview of important concepts and defines the meaning of some terms as used in this thesis.

Types and Type Systems

For our purposes, the following is a suitable definition of a *type system* from the literature:

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pie02, page 1]

In other words, phrases—usually expressions of a programming language—are classified into types. A *type* represents certain properties of a variable or other arbitrary expression. A type system accepts or rejects a given program based on these properties.

Commonly, programming languages embody one single native type system; for object-oriented languages a type commonly describes what the class of a certain value is. But notice that the preceding definitions for types and type systems are very open. This leaves room for many more kinds of types and type system, giving rise to pluggable type systems.

Static and Dynamic Typing

Programming languages where every expression's type can be determined by static analysis (*i.e.,* by analyzing the source without actually running it)

are said to be *statically typed*, making it possible to type check a program prior to running it. Popular statically typed object-oriented languages such as Java and C# require the types of all variables to be declared explicitly. But statically typed languages can also employ type inference to determine types, the functional language ML being a typical example.

Languages where the type of an expression might not be known until runtime are said to be *dynamically typed*. Rules about types are enforced at runtime. Typically, live objects carry type information with them, raising a runtime type error if they are submitted to an operation incompatible with their type. For popular dynamically typed languages such as Python, Ruby or Smalltalk these dynamic type checks happen at access to members of an object (in case of Smalltalk, only at at message sends, *i.e.,* method lookups). It is not yet common, but static typing can be imposed on dynamically typed languages through optional type annotations and pluggable type systems—this is the main theme of this thesis.

Type Inference

Type inference (also referred to as *type reconstruction*) is the process of deducing by static analysis the type of an expression in the absence of or only partial presence of explicit type information about the expression. For example, the type signature of a method may be inferred if the method was not explicitly annotated with types.

Type inference in statically typed languages usually serves to relieve a programmer from adding explicit type annotations to a program. In dynamically typed languages it can be used to collect some type information in a pre-runtime phase, *e.g.,* for compilation optimized for specific types.

Bibliography

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, December 1996.

[All06] Tristan Allwood. Fleece, pluggable type checking for dynamic programming languages. Master's thesis, Imperial College of Science, Technology and Medicine, University of London, June 2006.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 57–74, New York, NY, USA, 2006. ACM Press.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

[Bra04] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[FL03] Manuel Fähndrich and Rustan Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, 2003.

[Gra89] Justin Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign, August 1989.

[Mar06] Philippe Marschall. Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Put] Colin Putney. OmniBrowser, an extensible browser framework for Smalltalk. http://www.wiresong.ca/OmniBrowser.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings, Scheme and Functional Programming Workshop 2006*, pages 81–92. University of Chicago TR-2006-06, 2006.

[Str] Strongtalk: A high-performance open source smalltalk with an optional type system. http://www.strongtalk.org.

[Wuy] Roel Wuyts. RoelTyper, a fast type reconstructor for Smalltalk. http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/.