Type-Checking and Type-Inference for Object-Oriented Programming Languages

by

Justin Owen Graver

B.S., University of Nebraska - Lincoln, 1984
M.S., University of Illinois at Urbana-Champaign, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1989

Urbana, Illinois

Abstract

Type systems for object-oriented programming languages have been studied a great deal over the past few years. Since Smalltalk was one of the earliest object-oriented languages, it is not surprising that there have been several attempts to provide a type system for it. Unfortunately, none of the attempts have been completely successful. In particular, none of the proposed type systems are both type-safe and capable of type-checking most common Smalltalk programs. Smalltalk violates many of the assumptions on which most object-oriented type systems are based, and a successful type system for Smalltalk is necessarily different from those for other languages.

We have designed a new type system for Smalltalk. The biggest difference between our type system and others is that most type systems for object-oriented programming languages equate classes with types and subclassing with subtyping. In our type system, types are based on classes (i.e. each class defines a type or a family of types) but subclassing has no relationship to subtyping. This is because Smalltalk classes inherit implementation and not specification.

Our type system uses discriminated union types and signature types to describe inclusion polymorphism and describes functional polymorphism (sometimes called parametric polymorphism) using bounded universal quantification and parameterized types. It handles side-effects by basing the definition of the subtype relation on equality of parameterized types. It is type-safe, i.e. it prevents an object from receiving a message it does not understand.

We describe an inter-procedural type-inference algorithm for the type system. Type-inference is important because Smalltalk programmers are used to a typeless language. We wish to retain as much of the "look and feel" of Smalltalk as is possible. The type-inference algorithm computes the general type of a method based on its definition and computes a specific type for a method at each call-site based on site-specific type information.

Because the type system uses class information, it can be used for optimization. Specifically, if methods can be bound to message sends at compile-time then aggressive in-line substitution and inter-procedural optimization can be performed. It has been implemented as part of the TS (Typed Smalltalk) optimizing compiler.

Acknowledgements

I thank my advisor, Ralph Johnson, for his inspiration, imagination, guidance, and confidence. I thank the rest of my committee for their professional and technical advice. I thank my parents for their never ending support. I thank my friends and acquaintances in the Department of Computer Science for making my time in Illinois enjoyable. A special thanks goes to Steve Goering, Carl McConnell, and Larry Zurawski for helping to make this thesis a reality.

Contents

Chapter 1

Introduction

Users always demand more from any computing system, especially when the users are programmers. This is particularly true in today's world of graphical user interfaces, user extensible application environments, and increasingly large and complex software systems. Twenty years ago, cries to provide better programming tools were answered with the advent of structured programming languages like Algol and Pascal. Today, these same cries will most likely be answered by the use of object-oriented programming languages like Smalltalk and C++ for bigger and better programming environments. While object-oriented programming languages are slowly gaining acceptance from the general programmer community, no existing language delivers both the flexibility and efficiency that will ultimately be required to meet tomorrow's software needs.

A type system is an important part of a programming language. Languages that rely completely on run-time type-checking provide a high degree of flexibility but must usually sacrifice performance to do so. Recently there have been a number of object-oriented programming languages with static type systems [BHJL86, SCB*86, Str86, Mey88]. These languages are usually more efficient but less flexible than those that rely on run-time type-checking, like Smalltalk.

This thesis describes a static type system for Smalltalk. It can make Smalltalk programs both safer and more efficient without damaging the flexibility of the language and its programming environment. The type system has been implemented as part of the TS (Typed Smalltalk) optimizing compiler [JGZ88].

1.1 The Whats, Whys, and Hows

"Proponents of [object-oriented programming] suggest that it aids design, implementation, and maintenance of complex systems by supporting modularity and that it aids code reuse and the construction of easily extensible systems by supporting inheritance" [DT88, p. 29]. Slowly but surely, evidence is accumulating in support of these claims. Until the final verdict is in, it is important to explore this "new" frontier not only for the sake of extending our horizons but to hopefully gain a better understanding of the knowledge we already possess.

Our research deals specifically with Smalltalk, although most results should be generally applicable. We assume the reader is at least casually familiar with Smalltalk and object-oriented programming concepts such as instances, classes, and message sending. We review the unique and unusual features of Smalltalk here. A full description of the Smalltalk-80[1] programming language can be found in [GR83].

[1]Smalltalk-80 is a trademark of ParcPlace Systems.

Smalltalk is a pure object-oriented programming language in that everything, from integers to text windows to the execution state, is an object. In particular, classes are objects. Since everything is an object, the only operation needed in Smalltalk is message sending. Smalltalk is a typeless language. Every message send is dynamically bound to an implementation depending on the class of the receiver. This unified view of the universe makes Smalltalk a compact yet powerful language.

Function abstraction is accomplished through blocks. Smalltalk has no built-in control structures; they are all implemented in terms of message sending and blocks. For example, the ifTrue:ifFalse: message (Smalltalk's equivalent to if-then-else) is sent to a boolean object with two block arguments. If the receiver is an instance of class True the "true block" is evaluated. Similarly, if the receiver is an instance of class False the "false block" is evaluated. Looping is implemented in a similar way using recursion.

Smalltalk is not just a language, but a complete and powerful programming environment. It has the following features, which support the software development process:

- graphical user interface
- incremental compilation
- polymorphism
- interactive symbolic debugger
- extensive code library and browsers.

Part of the Smalltalk philosophy, and a goal of object-oriented programming, is to allow programmers to think very abstractly by providing powerful building blocks such as generic collections, hash tables, semaphores, pop-up menus, etc. Smalltalk programmers think at an algorithmic level rather than in terms of assignments, loops, and print statements.

What is the price of all this wonderfulness? Smalltalk suffers from two main problems due to its uniform use of dynamic procedure call binding:

- traditional non-polymorphic, computation-intensive code cannot be optimized, and
- procedure call linking errors (type errors) cannot be statically detected.

The first problem is manifest in poor performance. All current Smalltalk systems are interpreted and slow even when compared to other interactive dynamic-memory systems.[2] Although reasonable performance has been achieved through innovative interpreter implementations [DS84, CW86] the next performance leap must come from innovative compilation and optimization. The second problem manifests itself in the Smalltalk "does not understand" run-time error, which happens when a message is sent to an object that does not have a definition for it, bringing the system to an inelegant halt.

We have solved both of these problems by adding a static type system to Smalltalk. The resulting type system is quite different from those proposed by others designing type systems for object-oriented programming languages (e.g. [Suz81, BI82, Car84, BHJL86]). This is mainly because Smalltalk has a unique combination of useful features that no other language has.

In general, a static type system has many uses:

- it provides information that a compiler can use during code optimization to produce a more efficient program,

[2] An ad hoc comparison of Tektronix Smalltalk-80 and MIT Scheme gave the following execution times for calculating 1000!: Smalltalk—335 sec., Scheme—55 sec.

- it provides assurance that certain kinds of run-time errors cannot happen,
- it provides documentation to the reader of a program, and
- it is a useful aid during the design process.

We are primarily interested in the first two uses.

In Smalltalk, message sends are bound to methods at run-time. This gives Smalltalk much of its flexibility but also prohibits traditional compilation and optimization. Optimization is possible only if method lookup can be performed at compile-time. The type system solves this problem because it is class-based. If the type of the receiver of a message describes the set of possible classes of the receiver then method lookup can be performed for each class in the set.

Being able to perform method lookup at compile-time allows (late-bound) message sends to be replaced with procedure calls. This, in turn, makes further optimization possible. The most notable optimization is in-line substitution of method definitions. This eliminates the high cost of frequent procedure call linkage code, eliminates the creation of block contexts for simple control flow messages, and makes code optimization across message send boundaries possible. Great potential for this scheme has already been demonstrated in the TS optimizing compiler, which attains an average speedup of 10 over interpreted code for a given set of benchmarks [JGZ88].[3]

A class-based type system also solves type-checking problems. Again, if method lookup can be performed at compile-time then it is possible to catch "does not understand" errors before any code is executed. Due to the fact that Smalltalk can discriminate between classes of objects, there are some methods that require more powerful type-checking techniques such as case analysis [Joh86]. Other methods are impossible to fully type-check without the context of a specific call site and must be type-checked on a site-to-site basis. We extend our type system to handle this problem using a static analysis technique called abstract interpretation [CC77].

While adding static typing to Smalltalk is good because it improves performance, it is also bad because it requires users to enter type declarations in a previously untyped environment. To recover some of this flexibility, we introduce a type-inference algorithm that can effectively infer over 90% of the type-declarations required by the compiler. Unfortunately, there is a trade-off between optimization and type-inference. The type-inference algorithm computes the most general type for a method or variable even though the actual use of the method or variable may be much more specific. In order to maximize optimization, the more specific type should be supplied by the user. On the other hand, type-inference provides types that are general enough to be used with various implementations, allowing specific design decisions to be postponed until later in the design process.

While we are primarily interested in using a static type system for optimization and type-checking, these are not its only uses. Recall that a type system is also useful for documentation and as a design aid.

The fact that Smalltalk is a typeless language makes it both flexible and difficult to learn. Since there are no type declarations for variables, trying to figure out what kind of objects a method expects as arguments is often an exhausting adventure. While good comments can help, they too are often hard to find and must be taken on good faith. A static type system is a good solution since some form of type information, whether declared or inferred, is a required part of each method or class definition.

A type system can be an invaluable aid during the design phase of a software project. It can be used as a specification language for component interfaces and can provide a basis for automated

[3] As of August 1989, the average speedup for the benchmarks was close to 20.

design tools. For example, our type system is being used in a tool that will provide a tree-like picture of the component types of an object or class.

1.2 Background

1.2.1 Object-Oriented Programming Languages

A programming language is *object-oriented* if it has the following characteristics:

- data abstraction,
- inheritance, and
- late-bound procedure calls.

Classes are an implementation technique for factoring a common set of attributes out of a group of similar objects. Classes, although an important part of Smalltalk, are not a necessary characteristic of an object-oriented programming language (e.g. the "Self" language does not have classes [US87]).

An object-oriented computation involves objects sending messages to other objects. A message send is a specialized form of a procedure call. Each message send specifies the name of a message, a receiver, and zero or more arguments. The receiver can be thought of as a specially designated argument. A procedure implementation is called a *method*. The particular method invoked for a message send is determined, either at compile-time or run-time, by the class (or type) of the receiver. Depending on the language, the bindings between message sends and methods can change dynamically throughout the execution of a program. In Smalltalk, the dynamic binding of a method to a message send (call site) is called *method lookup*.

In a typeless object-oriented programming language, such as Smalltalk, method lookup must be done at run-time. Since few compile-time checks can be performed, a common run-time error is attempting to send a message to an object that does not define or inherit an implementation for it. In Smalltalk this is known as the "does not understand" error. Execution is halted and cannot proceed until the problem is found and corrected. It is often tedious and time-consuming to find and correct such errors, and there is no guarantee of finding them all. Even "finished" application programs may occasionally fall prey to the "does not understand" error if unexpected or unusual circumstances arise. However, run-time method lookup is essential for polymorphism and in languages that allow programs to create and execute other programs at run-time.[4]

In statically typed object-oriented programming languages, such as Trellis [SCB*86] or Eiffel [Mey88], the "does not understand" error is caught at compile-time as a type error. Assuming that compilation (or execution) is not permitted until all type errors have been fixed, this kind of run-time error can never arise. Another advantage of static type-checking is that, frequently, method lookup can be performed at compile-time, thereby permitting optimization.

1.2.2 Polymorphism

Object-oriented programming languages have polymorphic methods. A polymorphic method is one that can be invoked with different types of arguments. There are two different varieties of polymorphism commonly associated with object-oriented programming languages: functional and inclusion [DT88].

[4] ThingLab [Bor79] is a good example of this facility being used in Smalltalk.

```
size
    "Answer how many elements the receiver contains."

    | tally |
    tally ← 0.
    self do: [:each | tally ← tally + 1].
    ↑tally
```

Figure 1.1: The size method from class Collection.

A method that exhibits functional polymorphism works for a range of argument types that have a common structure. An example is a method that accesses elements of any array, regardless of the type of the elements. The same code is executed for each invocation of such a method, regardless of the specific types of actual arguments.

This kind of polymorphism is usually described by allowing method types to contain universally quantified (possibly with bounded quantification) type parameters [Mil78, CW85]. Such type parameters may be syntactically implicit (as in [Mil78]) or explicit (as in [CW85]). At each call site, these type variables are instantiated to specific types based on the types of the actual arguments.

A method that exhibits inclusion polymorphism works for a range of argument types that have a common behavioral (but not necessarily a common structural) specification. An example is a method that computes the size of any collection (array, set, etc.). The same top-level method is executed for each call, but the methods invoked from within it may vary.

Inclusion polymorphism can be described using bounded quantification where the type of an acceptable argument must satisfy some predicate [CW85]. The predicate usually checks the argument type for some minimal behavioral specification. Types can also be interpreted as behavioral specifications. Thus, inclusion polymorphism is most commonly used to describe methods that can be invoked with different kinds of arguments that are related by inclusion (see Section 1.2.3).

A method may also exhibit functional and inclusion polymorphism simultaneously. An excellent example is the Smalltalk method size, implemented in class Collection. The definition of the size method is shown in Figure 1.1. The do: message iterates over each element of the receiver and is implemented in the subclasses of Collection (Array, Set, etc.). The size method will be executed whenever the size message is sent to an instance of any subclass of Collection (inclusion polymorphism) regardless of the kind of elements in the Collection (functional polymorphism).

1.2.3 Types and Subtyping

A type system usually consists of base types, such as *Integer*, *Character*, and *Boolean*, and more complex types built from type constructors. Traditional constructed types include product types ($S \times T$), function types ($S \rightarrow T$), and disjoint (discriminated) union types ($S + T$). There are also parameterized types, such as *Array(Integer)* and *Set(Character)*.[5]

The *subtype* (or *type inclusion*) relation between types is defined informally as follows: if T is a subtype of S (written $T \sqsubseteq S$) then any context requiring something of type S may be given something of type T instead. The $\sqsubseteq$ relation is:

[5]For a good introduction to these and other related concepts see [All86].

```
foo: anArray
        anArray at: 1 put: $d.
        ↑anArray
```

Figure 1.2: Example of side-effects in a method.

- reflexive, $S \sqsubseteq S$,
- transitive, if $S \sqsubseteq T$ and $T \sqsubseteq U$ then $S \sqsubseteq U$, and
- antisymmetric, if $T \sqsubseteq S$ and $S \sqsubseteq T$ then $S = T$.

Subtyping for constructed types is defined for product, function, and union types, respectively, as:

- $S' \times T' \sqsubseteq S \times T$ iff $S' \sqsubseteq S$ and $T' \sqsubseteq T$,
- $S' \rightarrow T' \sqsubseteq S \rightarrow T$ iff $S \sqsubseteq S'$ and $T' \sqsubseteq T$, and
- $T_1 + T_2 + \ldots + T_m \sqsubseteq S_1 + S_2 + \ldots + S_n$ iff for each $1 \leq i \leq m$ there exists a $1 \leq j \leq n$ such that $T_i \sqsubseteq S_j$.

Note that function types are monotonic in their codomains and antimonotonic in their domains [MS82]. Subtyping for parameterized types is language dependent.

Types for functional languages are sets of values. Computations in such languages are value-based and side-effect free. In such a context it makes sense to define subtyping for parameterized types as:

- $C'(T') \sqsubseteq_{fd} C(T)$ iff $C' = C$ and $T' \sqsubseteq T$ (functional definition).

For example, *Array(Integer)* $\sqsubseteq_{fd}$ *Array(Integer + Character).*

Functional languages have a semantic function mapping types to sets of values. The subtyping relation on types translates directly to the subset relation on corresponding value sets. The definition of subtyping for parameterized types in the context of imperative languages is necessarily different.

Consider, for example, the Smalltalk method shown in Figure 1.2, which takes an argument of type, say, *Array(Integer)*, inserts a Character[6] in the first position, and returns an Array of type *Array(Integer + Character).* In a functional language, the array that "goes in" and the array that "comes out" are considered distinct values. This is similar to passing arguments by value in imperative languages. Imperative languages also allow arguments to be passed by reference. In this case, suppose the caller of the foo: method supplies an actual argument of type *Array(Integer).* If the actual argument is a variable of type *Array(Integer)* then the caller is type-checked based on this assumption. The problem is that while this assumption may be valid before the call to foo: it is definitely not valid afterwards. This cannot be allowed, given compile-time type-checking, if the system is to be type-safe.[7]

Hence, for imperative languages that allow multiple typed references to objects, we define subtyping for parameterized types as:

[6] The phrase "a " refers to an instance of class . The phrase "class " refers to the class itself.

[7] This point can be also argued successfully based on the antimonotonic ordering on function types for the method as is done in [DT88].

- $C'(T') \sqsubseteq_{id} C(T)$ iff $C' = C$ and $T' = T$ (imperative definition).

For example, *Array(Integer)* $\not\sqsubseteq_{id}$ *Array(Integer + Character)*.

Unless otherwise stated, $\sqsubseteq$ refers to the imperative definition of subtyping. It should be noted that this is how we have chosen to deal with the problem of side-effects (Amber [Car85a] solves this problem in a similar fashion). There may very well be other solutions to this problem (for example, see [Rey81]).

1.2.4 Inheritance and Type Systems

Inheritance is an important part of an object-oriented programming language. There are many flavors of inheritance. Inheritance can be single (each class has a single superclass) or multiple (each class has one or more superclasses), implicit (based on class specification, structure, or both) or explicit (defined by the user or the system), typed or untyped. Classes may inherit implementation, specification, or both.

The implementation of a class consists of the methods and structural attributes (e.g. instance variables) that it either defines or inherits. The specification of a class is roughly its type. We loosely define a *signature* to be a collection of operation signatures. An *operation signature* consists of the name of the operation, the types of its arguments, and the type of the result. A class specification can be an explicit signature [BHJL86], the implicit signature implied by the name of a class [SCB*86], or a signature combined with method pre- and post-conditions, class invariants, etc. [Mey88].

In an implementation inheritance hierarchy a subclass inherits the implementation of its superclass (or superclasses). A subclass may redefine an inherited method in any way it chooses by providing a new method definition with the same name. By definition, untyped object-oriented programming languages use implementation inheritance hierarchies.

In a specification inheritance hierarchy a subclass inherits not only the implementation of its superclass but also its specification. A subclass may only redefine an inherited method if the type of the new method is a subtype of the type of the inherited method. Almost all typed object-oriented programming languages use specification inheritance hierarchies.

Notice that we have not said, "untyped object-oriented programming language = implementation inheritance hierarchy," or, "typed object-oriented programming language = specification inheritance hierarchy." It is conceivable to have an untyped object-oriented programming language with a specification inheritance hierarchy, although it is not clear how the specification inheritance would be enforced or if such a language would gain anything over a typed one. More importantly, we will show that it is both possible and useful to have a typed object-oriented programming language with an implementation inheritance hierarchy. In such a system, typed methods are inherited but, because types may be context dependent, the type of an inherited method in a subclass may have no relation to the type of the method in the superclass.

Inheritance hierarchies are usually designed so that a subclass has more attributes and is more specialized than its superclass; the superclass has fewer attributes and a more general behavior. This organization is strictly enforced in specification inheritance hierarchies. Systems using implementation inheritance hierarchies are not restricted in this way, so other inheritance organizations may be explored [HO87].

There are essentially two varieties of type systems used for object-oriented programming languages: class-based and specification-based. In a class-based type system, the appearance of a class name in a type expression means that only instances of that class may appear during execution in a context that is described, prior to execution, by the type expression. In a specification-based

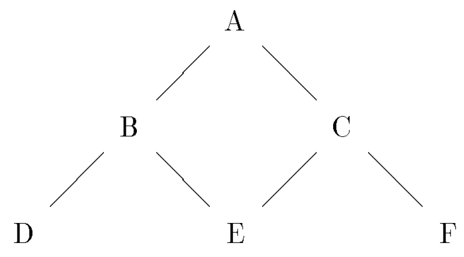

Figure 1.3: Part of a specification inheritance hierarchy.

type system, objects appearing during execution must conform to the specification denoted by the type expression, but do not necessarily have to be instances of any specific class.

When a specification-based type system is used for a language with a specification inheritance hierarchy, it is common (and natural) to use class names to denote the specification of the corresponding class. Since a subclass must conform to the specification of its superclass, by definition, a specification type denoted by the name of a class C automatically includes types denoted by the names of the subclasses of C. In such systems there is a direct and implicit correspondence between the subclass and subtype relations. Adding a new subclass to such a hierarchy automatically increases the type domain of existing definitions without requiring any visible change in their type descriptions.

Part of the allure of specification-based type systems is that the subtype relation can be defined as a partial order [Car84]. One advantage of this is that the union of two or more types can be replaced by its "nearest common supertype" in the type lattice.[8] This simplifies the type system by eliminating the need for explicit union types, but is much too restrictive for systems with explicit single inheritance (e.g. C++ and Smalltalk) because it makes a lot of programs type-incorrect that would execute with no problems.

For example, Figure 1.3 shows part of a specification inheritance hierarchy. Its corresponding type lattice will have the same structure. If a variable v can contain objects of type D or F then, under this scheme, its type would be A, the nearest common supertype of D and F. Type A represents the most general (least specific) specification shared by all of the subtypes of A in the type lattice. If the type of v is A then any expression whose type is a subtype of A can be assigned to v. While this does include types D and F, it also includes types A, B, C, and E. Furthermore, if the underlying language makes it possible to distinguish between Ds and Fs (certainly a reasonable request) then correct (executable) code can be written that sends D-messages and F-messages to v which are not in the specification for A. Hence, if the type of v is A, such message sends cannot be type-checked. C++ handles union types this way and, hence, suffers from this problem.[9]

One solution to this problem is to use multiple inheritance. In the previous example, classes D and F could be given an additional common superclass G that implements those methods common to both D and F. Type G would then represent the intersection of types D and F. The type of the

[8] This is the *join* ($\cup$) operation on types in a lattice. There is an analogous *meet* ($\cap$) operation which corresponds to taking the intersection of two or more types.

[9] C++ Version 2.0 uses multiple inheritance to solve this problem [Str87].

```
{  temporaries: aView <View + UndefinedObject>
   returnType: <Self> }

controlToNextLevel
        "Pass control to the next control level, that is, to the Controller of a subView
        of the receiver's view if possible. The receiver finds the subView (if any)
        whose controller wants control and sends that controller the startUp message."

        | aView |
        aView ← view subViewWantingControl.
        aView notNil ifTrue: [aView controller startUp]
```

Figure 1.4: The controlToNextLevel method for class Controller.

variable v would then be G and everything would type-check. Languages like Amber [Car85a] use implicit multiple inheritance so this problem does not arise (i.e. it gets solved automatically in the same manner since the intersection of any two types is a type).

A concrete example of this behavior appears in the controlToNextLevel method shown in Figure 1.4. The type of the temporary aView is declared to be *View + UndefinedObject*. In the last line of the method definition, the controller message, defined in class View, is sent to aView. To type-check this message send the compiler must know two things. First, it must know that aView could refer to an instance of class View. If *View + UndefinedObject* is replaced by its nearest common "supertype," *Object*, then this information is lost. Second, the compiler must know that aView, as the receiver of the controller message, cannot refer to an instance of class UndefinedObject. This is guaranteed at run-time using the notNil message to discriminate between nil and Views. We solve this problem using explicit union types and case analysis (see Section 2.2.2).

Another difficulty that uniformly plagues specification-based inheritance systems is caused by the antimonotonic ordering on function types. It seems reasonable that the specification of a class should include type information about its methods. It is also useful to define a subclass to be a more specific version of its superclass. The problem is that the antimonotonic ordering on function types, together with the restriction that an inherited method can only be redefined as a subtype of its original method, prevents a subclass from further restricting the argument types of inherited methods.

For example, suppose there is a class Channel whose instances act as carriers for instances of a class Message. Class Channel defines a method send: which takes Messages and inserts them into Channels. Now suppose SpecialChannel is a subclass of Channel and SpecialMessage is a subclass of Message, with the intention being that SpecialChannels carry only SpecialMessages. It is impossible to redefine the send: method in class SpecialChannel to take only SpecialMessage arguments. This is because the antimonotonic ordering on function (method) types says that if the type of the new send: method is to be a subtype of the old send: method then the argument type of the new send: method must be a *supertype* of the argument type of the old send: method. This problem is discussed at length in [DT88].

Class-based type systems are closely related to specification-based systems, but are more restrictive. In a class-based type system, a class name denotes the corresponding class specification, but the applicability of the specification is limited to instances of the named class, even if some subclasses conform to its specification.

In class-based type systems, subtype relations cannot arise implicitly from the subclass relation; they must come from explicit union types. The disadvantage of class-based systems, especially when used with specification inheritance hierarchies, is that adding a new subclass does not automatically increase the type domains of existing definitions. The advantage is that, since a finite number of classes are involved in a particular execution context and since this information is available at compile-time, class-based type information can be used for type-checking and optimization.

A class name in a specification-based type system denotes a possibly infinite set of classes (new classes may be added in the future) and, accordingly, a possibly infinite set of implementations. For this reason, specification-based type information cannot, generally, be used for optimization. However, since very few applications mutate the hierarchy at run-time, it is possible to perform compile-time analysis to determine which implementations match a given specification. If only one possible implementation exists then method lookup can be replaced with a procedure call. Eiffel uses a similar technique [Mey88].

1.2.5 Types and Smalltalk

Recall that the goals of this research are to develop a type system for Smalltalk that will make it safer and will allow it to be compiled and optimized. With this in mind, we examine the characteristics of Smalltalk that are relevant to achieving these goal.

Smalltalk is, in some regards, a hybrid imperative and higher-order programming language. It is imperative because assignment copies a reference to an object and methods can change the state of their arguments. It is higher-order because functions (blocks) are first class objects and every operation returns an object.

Smalltalk uses an untyped implementation inheritance hierarchy. This, together with the fact that Smalltalk was developed as a typeless language, makes retrofitting a type system a difficult task. Many parts of the Smalltalk inheritance hierarchy may, in fact, conform to specification inheritance, but there is nothing in the language that requires or enforces this.[10] It is common, however, to find classes that inherit implementation and ignore specification.

Dictionary is a good example of a class that inherits implementation but not specification; it is a subclass of Set. A Dictionary is a keyed lookup table that is implemented as a hash table of <key, value> pairs. Dictionary inherits the hash table implementation from Set, but applications using Sets would behave quite differently if given Dictionaries instead.

Abstract classes, a commonly used Smalltalk programming technique, provide other examples where classes inherit implementation but not specification. Consider the method definitions shown in Figure 1.5. These (and other) classes are used to track changes made to the system. The values method is defined, in terms of the parameters method, in the abstract class Change. The implementation of the values method is inherited by the subclasses of Change. The result of sending the values message depends on the result of sending the parameters message, which is different for each class in Figure 1.5. Hence, the specification of the values method is also different for each of the classes.

A type system for Smalltalk must describe the fact that Smalltalk classes inherit implementation and not specification. A class-based type system with explicit union types works for an implementation inheritance hierarchy and greatly simplifies optimization, but is not very flexible

[10] This is probably a consequence of the long evolution of Smalltalk coupled with the fact that specification inheritance is a logical organization.

```
class Change:

    values
        "Return the value to be used for the 'same as' filter."
        ↑Array with: self class with: self parameters

    parameters
        "Return the parameters used for the 'same as' text."
        self subclassResponsibility


class ClassRelatedChange:

    parameters
        ↑className


class MethodChange:

    parameters
        ↑Array with: className with: selector


class OtherChange:

    parameters
        ↑self text
```

Figure 1.5: ClassRelatedChange, MethodChange, and OtherChange are subclasses of Change.

when it comes to adding new classes to the system. To regain some of the flexibility we incorporate type-inference and signature-like constraints. This gives us both flexibility and the ability to optimize.

A type system suitable for an optimizing Smalltalk compiler must also have parameterized types. It is difficult to imagine a (non-trivial) Smalltalk application that does not use Collections. The compiler must be able to logically associate the type of objects being added to a Collection with the type of objects being removed from it. Without parameterized types, all Collections are reduced to homogeneous groups of generic objects. Due to the potentially imperative side-effects of any operation, the imperative definition of subtyping for parameterized types is used (see Section 1.2.3).

The type system must also be able to describe the functional and inclusion polymorphism exhibited by many Smalltalk methods. Functional polymorphism can be handled, in the usual manner, by using (implicit) bounded universal quantification of type variables. Inclusion polymorphism can be handled by using explicit union types or signature-like constraints.

1.3 Related Work

Given this background, we will now examine some type systems proposed for Smalltalk and for other object-oriented programming languages and see why they are not appropriate for our goals.

The majority of type systems proposed for object-oriented programming languages are inappropriate for a type-safe Smalltalk optimizing compiler because they are specification-based [BI82, Car84, CR85, CW85, BHJL86, SCB*86, Str86, Wan87, Car88, Mey88] rather than class-based [Suz81, ST84, Joh86]. In addition, many type systems for object-oriented programming languages do not include parameterized types [Suz81, Car84, ST84, Str86]. Parameterized types complicate a type system so this is rarely an oversight. It is usually a simplifying assumption whereby their omission is irrelevant with regard to the particular point being addressed. Type systems that do include parameterized types often use the functional version of the subtype relation, either because they are designed for functional languages [CW85, Wan87, Car88] or because they, incorrectly, take a functional view of Smalltalk [BI82].

1.3.1 Type Systems for Object-Oriented Programming Languages

Much of modern type system research is based on the work of Milner who published the first description of a polymorphic type-inference algorithm [Mil78]. Other important contributions from Milner's research include the use of unification [Rob65] in type inference, ideas concerning generic versus non-generic type variables, iterative type resolution, and the basic concepts of polymorphism.

One approach to designing a type system is to start with a theoretical model and then to examine its mathematical properties. It is easy to reason formally about the correctness and completeness of such systems because they are usually designed with that purpose in mind. On the other hand, such systems may be hard to implement efficiently or may not provide all of the desired functionality (perhaps because certain concepts evade formal treatment).

Cardelli models objects as records, a finite association of values to labels [Car84]. Inheritance is based implicitly on object structure rather than on a declarative hierarchy of classes as in Smalltalk. Informally, a record type t' (a list of *label: type* declarations) is a subtype of a record type t if t' contains all the fields of t, and possibly more. This provides for a natural notion of multiple inheritance. Cardelli's system is fairly simple and does not provide for parameterized types or explicit union types. He defines subtyping as simple set inclusion of type domains. Extensions to this system can be found in [CW85, Wan87, Car88]. Amber, a real programming environment based on this system, is discussed in [Car85a].

Trellis [SCB*86] and Eiffel [Mey88] are two "full scale" statically-typed object-oriented programming languages. They both have rich type systems incorporating many of the features previously mentioned (Eiffel does not have union types). Both languages are based on explicit specification inheritance hierarchies, which is reflected in their type systems. Also, Trellis treats discriminated union types with an explicit case statement, while our type system handles union types implicitly through case-analysis (see Section 2.2.2).

Emerald [BHJL86] is an object-based language used for constructing distributed applications. An Emerald object consists of a unique name, a data area (containing references to other objects), a set of operations (both public and private), and an optional process (some objects are active and others are passive). Objects are created by executing code that defines these properties.

A type is an Emerald object that specifies a signature. Type T is a subtype of type S iff for each operation signature $s \in S$ there exists a corresponding operation signature $t \in T$, with the same name and number of arguments, such that $t \sqsubseteq s$ (the standard antimonotonic subtyping relation for function types). Any Emerald object may serve as a type provided it implements the proper type operations. Types may be parameterized.

Emerald is strongly typed; all identifiers have explicit type declarations. Type-checking of procedure invocation and most type-checking of assignment is done statically at compile-time. When the specific type of an object being assigned to an identifier will not be known until run-time, a run-time conformity check is performed for the assignment.

Subtyping in Emerald is based on implicit specification inclusion, as in [Car85b]. This provides a clean conceptual model of types based on shared specification rather than on shared implementation (as in Smalltalk). Emerald also handles side-effects correctly by using the imperative definition of subtyping for parameterized types [BHJL87]. Unfortunately, Emerald has no notion of inheritance.

1.3.2 Smalltalk Type Systems

Section 1.3.1 discussed object-oriented programming languages that were designed together with a type system. In general, it is more difficult to design a type system for a language that was not designed with types in mind, as is the case with Smalltalk. Several researchers have attempted to introduce type systems for Smalltalk with varying degrees of success and formality.

The goal of Suzuki's research [Suz81] is to supply type declarations to existing Smalltalk programs for documentation, although he does mention the applications to efficient compilation. Suzuki defines the type of a variable to be the "union of classes that the variable will ever denote" [Suz81, p. 188]. The system requires no type information from the user. His approach is to extend Milner's polymorphic type-inference algorithm [Mil78] to handle union types. Sets of type equations and inclusion inequalities are obtained from the method definitions; these are then solved by unification and a transitive closure algorithm. The algorithm is applied iteratively to the entire Smalltalk image (at least in theory) until types converge.

Due to the enormous size of the data structures required by the algorithm and the space limitations imposed by Smalltalk-76,[11] Suzuki implemented a simplified version of the algorithm for a small subset of the Smalltalk kernel. It can be inferred from these facts and from his comments in [ST84] that the algorithm is not very successful.

Suzuki's system does not have parameterized types. The type-inference algorithm also assumes a static system (it seems inconceivable to apply the algorithm again, even incrementally, after the addition of each new method). This static state assumption is in direct conflict with the Smalltalk philosophy of allowing the user maximum experimental freedom. Also, the types inferred are not

[11] Suzuki's research was done on this earlier version of Smalltalk, but the results extend naturally to Smalltalk-80.

truly polymorphic but only an approximation based on the state of the image at the time the algorithm is applied. Some of these problems are addressed in [ST84] but there is still no treatment of parameterized types.

A primary goal of Borning and Ingalls' type system for Smalltalk-80 [BI82] is to provide machine-checkable documentation. User supplied type declarations are required for all variables except temporaries (whose types are inferred). Their specification-based type system assumes, incorrectly, that Smalltalk uses a specification inheritance hierarchy. Union types are handled by replacing the union of two types with the "nearest common supertype." They address many important issues such as parameterized types and the need for treating blocks with different numbers of parameters differently. They also lay the groundwork for much of the subsequent Smalltalk research. (Cardelli's type system [Car84] is very similar and shares many of the problems discussed below.)

One problem with Borning and Ingalls' approach is the restriction they place on the return type of certain methods. If an inherited method is redefined by a class, they require that its return type be a subtype of the return type of the method being overridden. This is not a requirement of Smalltalk nor is it even a convention. In fact, subclassing for generality is a legitimate programming methodology [HO87].

Their "nearest common supertype" treatment of union types is inadequate. As discussed in Section 1.2.4, this usually destroys the specific information needed for optimization. For example, the nearest common supertype of *Set* and *OrderedCollection* is *Collection*. The implementations of the two classes are, however, entirely different. If the union *Set + OrderedCollection* was preserved, a case statement could be generated and specific message sends could be bound at compile-time.

A major flaw with their system is the special treatment of nil. A common union type is *something + UndefinedObject*; the nearest common supertype is *Object*. Since, even for their system, this destroys any useful type information, they choose to include *UndefinedObject* as an implicit "subtype" of every type. Unfortunately, if every variable can legally contain either nil or some other kind of object then the only way to insure that no run-time errors occur (e.g. the "does not understand" error) is to provide an implementation for *every* message in the system in class UndefinedObject. This is clearly an unsatisfactory proposition. Furthermore, this makes any optimizations relying on the type of a variable impossible.

Cook and Roberts [CR85] build on the work of Borning and Ingalls. They handle union types more appropriately by keeping the union explicit, and identify many practical problems of trying to type-check Smalltalk that their system cannot handle. For instance, their method types are constant and must be inherited as defined. If a method type refers to the class of the receiver, the type may not be correct for a subclass that inherits the method definition. If the inheriting class wishes the type information of the inherited method type to be correct it must redefine the method, even though the actual method definition is unchanged. Our type system handles this problem by using a special type *Self* to refer to the type of the receiver in a method definition. If the method is inherited, *Self* (automatically) refers to the type of the inheriting class (see Section 2.1.5).

It is not surprising that some of the least formal research has achieved the best results. Smalltalk is a real programming language with a powerful programming environment that has evolved over time. Its design, while based on well conceived notions, has probably followed the path of least resistance. Fitting a formal description onto such an environment is hard. Nevertheless, the concepts of formal type systems manifest themselves in even the most pragmatic approaches.

Quicktalk [BMW86] uses a small subset of Smalltalk for writing user primitives which can be compiled directly into native machine code. Methods are bound at compile-time by requiring type declarations for variables. Although the system is small it incorporates many features of more powerful "typed" systems: type inference for temporaries, beta-reduction of control blocks, parameterized types, polymorphic message types, and the ability to take into account implementation dependencies. Weaknesses of the system include no notion of subtyping, no treatment of inheritance, no union types, no type-safety (if an unexpected object is encountered at run-time execution falls through to a fail case), and a very limited scope.

The Hurricane compiler [Atk86] uses type declarations and code optimization techniques similar to those in [Joh86] to improve Smalltalk performance. Like Quicktalk, Hurricane treats type declarations as hints not absolute truths. A check of the actual type of an object is inserted with a branch to a fail case if an unexpected object is encountered. It infers the types of literals, globals (assumed constant), and temporaries. Other features of the system include parameterized types, union types, special treatment of blocks (similar to [BI82]), beta-reduction of control blocks, and implicit treatment of inheritance. Weaknesses include the treatment of polymorphic message sends (keeping track of all encountered argument types rather than using type variables), no real use of type variables, and no subtyping concepts.

Johnson [Joh86] combines union types and parameterized types in a system whose main goal is to provide enough information for compile time optimizations. Our research is directly involved with expanding, defining, and implementing this system and solving the problems encountered therein.

1.3.3 Static Analysis

Type-checking and type-inference are both static analysis techniques. Our approach is similar in flavor, though not necessarily in intent, to the work of [Rey69, JM76, KU78, JM82].

A type-inference system very similar to ours is used by Mishra and Reddy [MR85], but for a different style of language. Their system includes full type-inference, arbitrary union types, parameterized types, and recursive types. Subtyping is based on subsets of union types. Their type-inference algorithm, like Suzuki's, solves a set of inclusion equations by unification and transitive closure. Although the system is more powerful than Suzuki's it is intended for applicative functional languages like Prolog, relying heavily on recursive list structures and axiomatic function descriptions. There is no analogy for inheritance or other object-oriented programming concepts.

Abstract interpretation [CC77] is a way of using method definitions to describe computations in a universe of abstract entities, e.g. types. Our technique of type-checking, using case analysis and union types, is actually a simple form of abstract interpretation in the type domain. While case analysis increases the power of the type-checker, it is a potentially major source of inefficiency, since there are, at worst, an exponential number of cases to consider. In practice, this potential exponential explosion seems not to be a problem.

An important side-effect of using abstract interpretation is that the order in which methods are type-checked becomes immaterial, since the method definitions are always available as the "full specification of the method type." However, always using full method definitions is quite inefficient. We show how to compress this information by doing partial type-inference.

1.4 Contributions

This research uses a unique combination of new and old type concepts in a new type system for Smalltalk that can be used by an optimizing compiler to achieve significant compile-time optimization. The following are some important features of this new type system:

1. It eliminates the "does not understand" run-time error.

2. It makes optimization possible through the use of explicit union types and automatic case analysis.
3. It handles side-effects by using the imperative definition of the subtype relation.
4. It describes inheritance in an implementation inheritance hierarchy by using explicit union types (i.e. subclasses are not subtypes).
5. It captures functional polymorphism by using implicit bounded universal quantification.
6. It allows polymorphic methods to be typed using constraints.
7. Its usability is enhanced by type-inference.

Chapter 2

TS Type System

2.1 Types

Most type systems for object-oriented programming languages equate classes with types and subclassing with subtyping. In our type system types are based on classes (i.e. each class defines a type or a family of types) but subclassing has no relationship to subtyping.

The abstract and concrete syntax for type expressions is shown in Figure 2.1 (abstract syntax on the left and concrete syntax on the right). In the concrete syntax grammar, and for other BNF-style grammars presented herein, terminals are underlined, optional expressions are enclosed in curly braces, (something)* represents zero or more repetitions of the something, and (something)$^+$ represents one or more repetitions of the something. Each of the type forms will be described in detail in the following sections.

We use the concrete syntax for examples and the abstract syntax for formal descriptions and inference rules. Type expressions are denoted by s, t, and u. Type variables are denoted by p, q, and r (abstract syntax) or by P, Q, and R (concrete syntax). Lists (tuples) of types are denoted by $\vec{t} = < t_1, t_2, \ldots, t_n >$. The empty list is denoted by $<>$, $t \cdot \vec{t}$ is the list $< t, t_1, t_2, \ldots, t_n >$, $\vec{s} \cdot \vec{t}$ is the list $< s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n >$, and $\mathbf{last}(\vec{t})$ is t_n. Similar notation is used for lists of type variables. C denotes a class name.

Strictly speaking, a type variable is not a type; it is a place holder (or representative) for a type. We assume that all free type variables are (implicitly) universally quantified at an appropriate level. For example, the local type variables of a method type are assured to be universally quantified

	t ::=	$Type$ ::=
(object type)	$C(\vec{t})$	$ClassName$ ($\underline{\textsf{of:}}$ $Type$)*
(block type)	$\mid \vec{t} \rightarrow t$	$\mid$ $\underline{\textsf{Block}}$ ($\underline{\textsf{of:}}$ $Type$)* $\underline{\textsf{returns:}}$ $Type$
(union type)	$\mid t + t$	$\mid Type \; \underline{+} \; Type$
	$\mid (t)$	$\mid \underline{(}Type\underline{)}$
(type variable)	$\mid p$	$\mid P$
($\top$)	$\mid Anything$	$\mid \underline{\textsf{Anything}}$
($\bot$)	$\mid Nothing$	$\mid \underline{\textsf{Nothing}}$

Figure 2.1: The abstract and concrete syntax of types.

over the type of the method; the class type parameters of a class are assumed to be universally quantified over all type definitions in the class. We also assume that all type variables are unique (i.e. have unique names) and are implicitly renamed whenever a type containing type variables is conceptually instantiated. For example, the local type variables of a method type are renamed each time the method type is "used" at a call site. We do use explicit range declarations for type variables to achieve bounded universal quantification (see Sections 2.1.4 and 2.1.5). Given these assumptions, we treat type variables as types.

2.1.1 Object and Union Types

Some variables contain only one kind of object. For example, all Characters have an instance variable that contains a SmallInteger ASCII value. Other variables can contain different kinds of objects. For example, a Set can contain SmallIntegers, Characters, Arrays, Sets, and so on. For each class, we define a set of zero or more type variables called *class type parameters* that can be used in type expressions to specify the types of instance variables. These are the only type variables that can appear in type declarations for instance variables.

An *object type* is a class name together with a possibly empty list of types. The type list of an object type provides values for the class type parameters defined for the corresponding class. Thus, the size of the type list that may accompany a class name is fixed by the corresponding class definition. For example, the classes SmallInteger and Character have zero class type parameters, so *SmallInteger* and *Character* are valid object types. The classes Array and Set each have one class type parameter, so *Array of: SmallInteger* and *Set of: (Array of: Character)* are valid object types. Class type parameters and other type variables will be discussed in more detail later.

The type system defined so far cannot describe an Array containing both Characters and Integers. In Smalltalk, a variable can contain many different kinds of objects over its lifetime. Thus, a type can be a nonempty "set" of object types (or other types). This kind of type is called a *union type*. An example is

(Array of: Integer) + (Array of: Character) + (Array of: (Integer + Character)).

Here, the plus operator is read as "or," so a variable of the above type could contain an Array whose elements are all Integers, all Characters, or a mixture of the two.

2.1.2 Subtyping

The subtype relation $\sqsubseteq$ (introduced and defined in Section 1.2.3) intuitively says that if $s \sqsubseteq t$ then everything of type s is also of type t. The subtyping relation for our type system can be formalized as a set of type inference rules (as in [CW85]). $\mathcal{C}$ is a set of inclusion constraints for type variables. The notation $\mathcal{C}.p \sqsubseteq t$ means to extend the set $\mathcal{C}$ with the constraint that the type variable p is a subtype of the type t. The notation $\mathcal{C} \vdash s \sqsubseteq t$ means that from $\mathcal{C}$ we can infer $s \sqsubseteq t$. A horizontal bar is logical implication: if we can infer what is above it then we can infer what is below it.

The following are some basic inference rules.

$$\mathcal{C} \vdash t \sqsubseteq \mathit{Anything}$$

$$\mathcal{C} \vdash \mathit{Nothing} \sqsubseteq t$$

$$\mathcal{C}.p \sqsubseteq t \vdash p \sqsubseteq t$$

$$\mathcal{C}.t \sqsubseteq p \vdash t \sqsubseteq p$$

$$\mathcal{C} \vdash p \sqsubseteq p$$

The following rules are used for the type parameter lists of object types and for the argument type lists for block types.

$$\mathcal{C} \vdash <> \sqsubseteq <>$$

$$\frac{\mathcal{C} \vdash s \sqsubseteq s' \quad \mathcal{C} \vdash \vec{t} \sqsubseteq \vec{t}'}{\mathcal{C} \vdash s \cdot \vec{t} \sqsubseteq s' \cdot \vec{t}'}$$

$$\frac{\mathcal{C} \vdash s \cdot \vec{t} \sqsubseteq s' \cdot \vec{t}'}{\mathcal{C} \vdash s \sqsubseteq s' \quad \mathcal{C} \vdash \vec{t} \sqsubseteq \vec{t}'}$$

Recall from Section 1.2.3 that subtyping is reflexive, transitive, and antisymmetric.

$$\mathcal{C} \vdash t \sqsubseteq t$$

$$\frac{\mathcal{C} \vdash s \sqsubseteq t \quad \mathcal{C} \vdash t \sqsubseteq u}{\mathcal{C} \vdash s \sqsubseteq u}$$

$$\frac{\mathcal{C} \vdash s \sqsubseteq t \quad \mathcal{C} \vdash t \sqsubseteq s}{\mathcal{C} \vdash s = t}$$

$$\frac{\mathcal{C} \vdash s = t}{\mathcal{C} \vdash s \sqsubseteq t \quad \mathcal{C} \vdash t \sqsubseteq s}$$

Recall from Section 1.2.5 that, due to the imperative nature of Smalltalk, the imperative definition of subtyping for parameterized types (object types) is used. This means that one object type is included in another if and only if they are equal. (Be careful to distinguish between a set of inclusion constraints $\mathcal{C}$ and a class name C.)

$$\frac{\mathcal{C} \vdash \vec{s} = \vec{t}}{\mathcal{C} \vdash C(\vec{s}\,) \sqsubseteq C(\vec{t}\,)}$$

$$\frac{\mathcal{C} \vdash C(\vec{s}\,) \sqsubseteq C(\vec{t}\,)}{\mathcal{C} \vdash \vec{s} = \vec{t}}$$

This is a subtle, but very important, distinction worth restating. The example given in Section 1.2.3 was

$$\textit{Array of: Integer} \not\sqsubseteq \textit{Array of: (Integer + Character).}$$

An object type t is included in a union type u only if t is included in one of u's elements.

$$\frac{\mathcal{C} \vdash s \sqsubseteq t}{\mathcal{C} \vdash s \sqsubseteq t + u}$$

A union type u is included in a union type u' only if each element of u is included in u'.

$$\frac{\mathcal{C} \vdash s \sqsubseteq u \quad \mathcal{C} \vdash t \sqsubseteq u}{\mathcal{C} \vdash s + t \sqsubseteq u}$$

Some examples are:

$$\begin{gathered}
\textit{SmallInteger} \sqsubseteq \textit{SmallInteger} \\
\textit{Array of: SmallInteger} \sqsubseteq \textit{Array of: SmallInteger} \\
\textit{Character} \sqsubseteq \textit{Character + SmallInteger} \\
\textit{SmallInteger} \not\sqsubseteq \textit{Character} \\
\textit{SmallInteger} \not\sqsubseteq \textit{Array of: SmallInteger} \\
\textit{Array of: Integer} \not\sqsubseteq \textit{Set of: (Array of: Integer)} \\
\textit{Array of: (Integer + Character)} \not\sqsubseteq \textit{(Array of: Integer) + (Array of: Character)} \\
\textit{Array of: Integer} \not\sqsubseteq \textit{Array of: Float.}
\end{gathered}$$

It is worthwhile to pause and understand these examples clearly before continuing.

2.1.3 Block Types

We distinguish between blocks (function abstractions) and other objects for our base types. A *block type* consists of the name *Block*, a possible empty list of types for block arguments, and a return type. For example,

Block of: (Array of: Character) of: SmallInteger returns: Character

represents the type of a block with two arguments, and *Block returns: SmallInteger* represents a block with no arguments. Block types differ from object types in several ways. Unlike object types, there is no class Block. Types beginning with the name *Block* can have different sized argument type lists.

A block type u is included in a block type u' only if the return type of u is included in the return type of u' and if each argument type of u' is included in the corresponding argument type of u (this is the standard subtype relation for function types).

$$\frac{\mathcal{C} \vdash \vec{s}' \sqsubseteq \vec{s} \qquad \mathcal{C} \vdash t \sqsubseteq t'}{\mathcal{C} \vdash \vec{s} \rightarrow t \sqsubseteq \vec{s}' \rightarrow t'}$$

$$\frac{\mathcal{C} \vdash \vec{s} \rightarrow t \sqsubseteq \vec{s}' \rightarrow t'}{\mathcal{C} \vdash \vec{s}' \sqsubseteq \vec{s} \qquad \mathcal{C} \vdash t \sqsubseteq t'}$$

For example,

Block returns: Character $\sqsubseteq$ *Block returns: Character*

Block returns: (Character + SmallInteger) $\not\sqsubseteq$ *Block returns: Character*

Block of: (Float + SmallInteger) returns: SmallInteger
$\sqsubseteq$ *Block of: Float returns: (Character + SmallInteger)*

2.1.4 Typed Class Declarations

All instance variables and class variables defined for a class must have type declarations. As stated previously, each class defines a set of type variables called class type parameters, which may be used to specify the types of instance and class variables. These are the only type variables that can appear in type declarations for such variables.

The class definition shown in Figure 2.2 declares OrderedCollection to be a subclass of SequenceableCollection with a class type parameter *ElementType*.[1] Note that the type of each instance variable is declared when the variable is declared and that angle brackets are used to set off type expressions from the regular Smalltalk code.

A class definition defines an "object type" in which the type list is the list of class type parameters (appearing in the same order as in the class definition). The type defined by the above class definition would be

OrderedCollection of: ElementType.

[1] To allow more expressive names, type variables in TS are capitalized identifiers. They must be explicitly declared to distinguish them from other names.

```
SequenceableCollection subclass: #OrderedCollection
      instanceVariables: 'firstIndex <SmallInteger>
                          lastIndex <SmallInteger>'
      classVariables: ''
      typeParameters: 'ElementType'
      poolDictionaries: ''
      category: 'Sequenceable-Collections'
```

Figure 2.2: Class definition for OrderedCollection.

The scope of the class type parameter *ElementType* is limited to the method and variable definitions in class OrderedCollection (the same scope as a class variable). Class type parameters are inherited by subclasses, just like instance and class variables. Any use of the OrderedCollection type outside of this scope must "instantiate" its class type parameter to a type constant (like *Character* or to a locally known type variable. Thus, the above type actually defines a family of object types.

The *range* (upper bound for type instantiation) of a class type parameter may be restricted by including an optional range type declaration. A class type parameter may only be instantiated to a type that is a subtype of its range type. The range declaration of a class type parameter is syntactically identical to the type declaration for a variable. In the above example, if we wished to restrict the elements of OrderedCollections to be Characters, the typeParameters: field would be declared as 'ElementType <*Character*>'. If the range declaration of a class type parameter is omitted it is assumed to be *Anything*.

2.1.5 Typed Method Declarations

Class variables and instance variables have their types declared when their corresponding class is defined. Method arguments, temporary variables, and block arguments have their types declared in methods. The only difference between a typed method and an untyped method is that the typed method has a type declaration. The type declaration must precede any part of the method definition. A typed method template is shown in Figure 2.3. The arguments for the fields arguments:, temporaries:, and blockArguments: are lists of identifier <*type*> declaration pairs. The arguments for the receiverType: and returnType: fields are single type declarations. The arguments for the localTypeVariables: field are capitalized identifiers. The range of a type variable can be restricted by using the range: modifier. For example, the local type variable declaration <*P range: (Integer + Character)*> declares P to be a type variable that can be associated with any of the types *Integer*, *Character*, or *Integer + Character*. Local type variables are instantiated during type-checking to types determined by the types of actual arguments at a call site. Local type variables will usually, though not necessarily, correspond to the class type parameters of some class.

The do: method for SequenceableCollection is shown in Figure 2.4 as an example of a typed method definition. Notice that certain fields of the type declarations have been omitted. The syntax rules for type declarations are fairly liberal. Fields may occur in any order. All fields, except receiverType: and returnType:, may have multiple occurrences. The minimum allowable type

```
{ localTypeVariables: <P1> <P2>
  receiverType: <Self>
  arguments: arg1 <arg1Type> arg2 <arg2Type>
  temporaries: temp1 <temp1Type> temp2 <temp2Type>
  blockArguments: blkArg1 <blkArg1Type> blkArg2 <blkArg2Type>
  returnType: <Self> }

message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

Figure 2.3: Template for typed methods.

```
{ localTypeVariables: <P>
  arguments: aBlock <Block of: ElementType returns: P>
  temporaries: index <SmallInteger> length <SmallInteger>
  returnType: <Self> }

do: aBlock
    "Evaluate aBlock for each of the receiver's elements."

    | index length |
    index ← 0.
    length ← self size.
    [(index ← index + 1) <= length]
        whileTrue: [aBlock value: (self at: index)]
```

Figure 2.4: The do: method for SequenceableCollection.

declaration for a TS method is "{}."[2] If either the receiverType: or returnType: field is omitted, the special type *Self* is assumed.

Self is intended to represent the type of the pseudo-variable self (and super). A problem with this is that the type of self when a method is defined is different from the type of self when a method is inherited by a subclass. The solution we have adopted is to use *Self* on a textual level as a macro. The special type *Self* is not really a type at all, but is "macro-expanded" to a type at compile-time. In the absence of a receiverType: declaration (which is over 99% of the time), *Self* defaults to the object type for the class in which the method is being compiled. If a receiverType: declaration is given, *Self* is replaced with that rather than with its default. This means that a legal receiverType: declaration must be a subtype of the default value of *Self.* The *Self* expansion allows the correct method type to be computed, from an inherited method definition, for a class different from the one in which the method was originally compiled.

To obtain a contextually correct method type for an inherited method, the method definition is recompiled in the context of the inheriting class. At present, no code is generated, since the only purpose of recompilation is to compute the method type. The result is that a single method definition has a specialized method type in every class that inherits it. In the future, a single method definition might also have a specialized implementation for each specialized type.

This solution also solves problems that arise from defining or overriding methods that affect the types of inherited methods. Consider, once again, the method definitions shown in Figure 1.5. The values method is defined in class Change and inherited by the other classes. The return type of values depends on the return type of parameters, which is different for each class. The types of the different parameters methods are

<*Change*> parameters ↑<*Change*>,
<*ClassRelatedChange*> parameters ↑<*Symbol*>,
<*MethodChange*> parameters ↑<*Array of: Symbol*>, and
<*OtherChange*> parameters ↑<*String*>,

where the types in "receiver position" are receiver types and the types following the "↑" are return types. Thus, the values method must be recompiled in the context of each subclass of Change to compute its correct return type for that context.

Returning to the example of Figure 2.4, SequenceableCollection has one class type parameter called *ElementType*, so the type substituted for *Self* will be *SequenceableCollection of: ElementType*. The argument aBlock will take one argument of type *ElementType* and will return something of unknown type. Hence, the return type of aBlock is the local type variable *P*. When this message is referenced in another method definition both *ElementType* and *P* will be associated with actual receiver and argument types. The temporary variables index and length have values dependent on the size of Collections. In Smalltalk implementations with 30+ bit SmallIntegers, the possible size of any Collection is well within the range of SmallIntegers.[3] There is no explicit return statement in the method so it implicitly returns self.

Type declarations and typed methods introduce the notions of message and method types. A *message type* consists of a message selector, a receiver type, a list of argument types, and a return

[2] At present, the TS compiler and the regular Smalltalk "compiler" must coexist. Requiring the first character of a TS method definition to be a "{" provides a quick test for dispatching the code to the proper compiler.

[3] Smalltalk defines infinite precision integer arithmetic using the classes , , and (the three subclasses of class). are essentially machine integers; instances of the classes are more complex. It is very important, for optimization, to be able to identify variables of type *SmallInteger* since operations on them can frequently be mapped directly to machine instructions. Unfortunately, since even the sum or difference of two is not guaranteed to be a , most integer variables must have type *LargeNegativeInteger + SmallInteger + LargePositiveInteger.*

type. A *method type* consists of a message type and certain constraints, which will be defined in later chapters. A method type denotes the type of a method relative to a specific class of receivers. Therefore, *Self* should already have been expanded and should never appear in a method type. The method type of the above method is denoted by

$$<\textit{SequenceableCollection of: ElementType}>\ \mathsf{do}\!:\ <\textit{Block of: ElementType returns: T}> \\ \uparrow<\textit{SequenceableCollection of: ElementType}>.$$

Message types use the same notation (method types with no constraints are roughly equivalent to message types). Local range constraints on type variables and other constraints (explained later) are shown enclosed in braces following the return type of a method type. For example,

$$<\textit{Object}>\ \mathsf{foo}\!:\ <P>\ \uparrow<P>.\ \{<P> \sqsubseteq\ <\textit{Integer + Character}>\}.$$

Subtyping for message types, although complicated by type variables (see Definition 2 in Section 5.2), is essentially the same as for block types, with the added condition that the selectors must be equal. A message type can be implemented by tagging a block type with a message selector, and letting the first argument type of the block type represent the receiver type of the message type.

2.2 Type-Checking

This section presents a specification for the type-checking algorithm. It also describes some of the unique features of our type system, identifies some problem areas, and gives more details on using the system.

The specification of the type-checking algorithm is presented in terms of type-inference rules similar to those used to describe type inclusion. Besides a set of type constraints $\mathcal{C}$, we need a set of assumptions $\mathcal{A}$ about the types of variables. The notation $\mathcal{A}.v : t$ means to extend the set $\mathcal{A}$ with the assumption that variable v has type t. The notation $\mathcal{C}, \mathcal{A} \vdash e : t$ means that from the constraints $\mathcal{C}$ and assumptions $\mathcal{A}$ we can infer that an expression e has type t. If a type can be inferred for an expression the expression is *type-correct.* The following rule describes the essence of subtyping:

$$\frac{\mathcal{C}, \mathcal{A} \vdash e : s \qquad \mathcal{C} \vdash s \sqsubseteq t}{\mathcal{C}, \mathcal{A} \vdash e : t}$$

2.2.1 Basic Type-Checking

A method is type-correct if each of its statements is type-correct and if the type of the expression in each return statement is a subtype of the declared return type of the method. Type-checking an expression requires knowing the types of its subexpressions.

The types of all variables are declared.

$$\mathcal{C}, \mathcal{A}.v : t \vdash v : t$$

The types of the pseudo-variables true, false, and nil are constants.

$$\vdash \underline{\mathsf{true}} : \textit{True}$$
$$\vdash \underline{\mathsf{false}} : \textit{False}$$
$$\vdash \underline{\mathsf{nil}} : \textit{UndefinedObject}$$

The type of a literal (constant) is inferred from its class.

$$\vdash n : Integer$$
$$\vdash \underline{\$}c : Character$$
$$\text{etc.}$$

If this type is not general enough then type coercion can be used to supply the "correct" type (see Section 2.3.3).

An assignment statement $v \leftarrow e$ is type-correct if both v and e are type-correct (variables are trivially type-correct) and if the type of e is a subtype of the type of v. The type of an assignment statement $v \leftarrow e$ is the type of e.

$$\frac{\mathcal{C},\mathcal{A} \vdash v : s \qquad \mathcal{C},\mathcal{A} \vdash e : t \qquad \mathcal{C} \vdash t \sqsubseteq s}{\mathcal{C},\mathcal{A} \vdash (v \leftarrow e) : t}$$

A return statement $\uparrow e$ is type-correct if e is type-correct and if the type of e is a subtype of the declared return type of the method. The set of assumptions $\mathcal{A}$ contains a special variable returnValue whose type is the declared return type of the method under consideration. Recall that to type-check an expression we must know the types of its subexpressions. The only place a return statement can appear is as the last statement in the statement list of a method or a block. The types of the top-level expressions in the statement list of a method can be ignored. Thus, the only place where a return statement can appear as a subexpression of a more complex expression is within a block. The return type of a block (see below) is determined by the type of the last statement in its statement list. The type of a block will always be checked for inclusion within some block type. To afford the maximum freedom to such blocks, the type given to return statements is the special type *Nothing*, which is included in any type.[4]

$$\frac{\mathcal{C},\mathcal{A} \vdash \underline{\mathsf{returnValue}} : t \qquad \mathcal{C},\mathcal{A} \vdash e : t}{\mathcal{C},\mathcal{A} \vdash (\uparrow e) : Nothing}$$

A statement sequence $e_1 \ldots e_n$ is type-correct if each statement in the sequence is type-correct. The type of a statement sequence is the type of the last statement in the sequence.

$$\frac{\mathcal{C},\mathcal{A} \vdash e_1 : t_1 \qquad \ldots \qquad \mathcal{C},\mathcal{A} \vdash e_n : t_n}{\mathcal{C},\mathcal{A} \vdash (e_1 \ldots e_n) : t_n}$$

The type of a block is inferred based on the declared types of its arguments and the inferred type of its statement list.

$$\frac{\mathcal{C},\mathcal{A}.\vec{y} : \vec{t} \vdash e : u}{\mathcal{C},\mathcal{A} \vdash [\vec{y} \mid e] : \vec{t} \rightarrow u}$$

An additional inference rule for blocks is given in Section 2.2.2.

Due to the simplicity of Smalltalk, the only remaining expression is the message send. Type-checking a message send involves looking up one or more method types. The message send is type-correct if, for each method type, there exists a mapping of type variables to types such that, under this mapping, the type of each actual argument is a subtype of the corresponding formal argument. The type of a message send is the union of the return types of each of these method types, evaluated in their respective type assignment environments.

Let H be a hierarchy of typed class definitions and $H < C, m >$ be the definition in class C for method m, i.e.

$$H < C, m >= \vec{p} \sqsubseteq \ \vec{u}\, C(\vec{s}\,)\ m\ \vec{y} : \vec{t} \uparrow u\ \vec{z} : \vec{t}' e$$

[4] It is important to distinguish between the type of a return statement and the type of its expression.

where $\vec{p} \sqsubseteq \vec{u}$ are local type variables with range declarations, $C(\vec{s})$ is the type of the receiver, $\vec{y} : \vec{t}$ are typed arguments, u is the return type, $\vec{z} : \vec{t}'$ are the typed temporaries, and e is the method body. The notation

$$C(\vec{s}) \cdot \vec{t} \rightarrow u \mid \vec{p} \sqsubseteq \vec{u}$$

denotes a method type where $C(\vec{s})$ is the receiver type, $\vec{t}$ is the list of argument types, u is the return type, and $\vec{p} \sqsubseteq \vec{u}$ are range constraints. For notational convenience, we use the following inference rule to extract method types from method definitions.

$$\frac{H < C, m >= \vec{p} \sqsubseteq \vec{u}\, C(\vec{s})\ m\ \vec{y} : \vec{t} \uparrow u\ \vec{z} : \vec{t}' e}{\vdash < C, m >_d : C(\vec{s}) \cdot \vec{t} \rightarrow u \mid \vec{p} \sqsubseteq \vec{u}}$$

The notation $\vdash < C, m >_d : t$ means it can be inferred that t is the declared method type of the method invoked when sending the message m to an instance of class C.

In the following inference rule for message sends, $e : C(\vec{s})$ signifies that the type of the receiver is an object type, $\vec{e} : \vec{t}$ denotes the typed arguments, and $e\ m\ \vec{e}$ denotes a message send.

$$\frac{\begin{array}{c}\mathcal{C}, \mathcal{A} \vdash e : C(\vec{s}) \qquad \mathcal{C}, \mathcal{A} \vdash \vec{e} : \vec{t} \\ \mathcal{C}, \mathcal{A} \vdash < C, m >_d : C(\vec{s}') \cdot \vec{t}' \rightarrow u' \mid \vec{p} \sqsubseteq \vec{u} \\ \mathcal{C}.\vec{p} \sqsubseteq \vec{u}.C(\vec{s}) \sqsubseteq C(\vec{s}').\vec{t} \sqsubseteq \vec{t}' \vdash u' \sqsubseteq u\end{array}}{\mathcal{C}, \mathcal{A} \vdash (e\ m\ \vec{e}) : u}$$

Message sends to receivers with union types are handled by an inference rule in Section 2.2.2.

Consider, for example, the message send anArray at: 2, where the type of the variable anArray is *Array of: Character*. If the method type associated with the method that would be invoked if the at: message were sent to an Array is

$$<\textit{Array of: } P> \texttt{at:} <\textit{SmallInteger}> \uparrow <P>$$

then the message send is type-correct under the mapping $\{P \mapsto \textit{Character}\}$. We will use such mappings later so it is worthwhile to elaborate on their meanings.

According to the inference rule for message sends, extending an initial set of constraints $\mathcal{C}$ with the inclusion equations

$$\begin{array}{ll} P \sqsubseteq \textit{Anything} & \text{(range of } P\text{)} \\ \textit{Array of: Character} \sqsubseteq \textit{Array of: } P & \text{(receiver type)} \\ \textit{SmallInteger} \sqsubseteq \textit{SmallInteger} & \text{(argument type)} \end{array}$$

should lead us to a solution for the inclusion equation

$$P \sqsubseteq ? \qquad \text{(return type).}$$

The inference proceeds as follows, with the desired solution boxed:

$$\frac{\dfrac{\textit{Array of: Character} \sqsubseteq \textit{Array of: } P}{\textit{Character} = P}}{\textit{Character} \sqsubseteq P \qquad \boxed{P \sqsubseteq \textit{Character}}}$$

The substitution mapping $\{P \mapsto \textit{Character}\}$ is the *solution* of the three inclusion equations. In general, there may not be a unique solution to a given set of inclusion equations. How to deal with (avoid) multiple solutions is discussed in Chapter 5. If no solution exists then there is a type error somewhere.

We also need to define an inference rule for type-checking method definitions. Recall that the abstract definition for a method m looks like

$$\vec{p} \sqsubseteq \vec{u}\, C(\vec{s}\,)\; m\; \vec{y} : \vec{t} \uparrow u\; \vec{z} : \vec{t}' e.$$

If, by adding type declarations appropriately to $\mathcal{C}$ and $\mathcal{A}$, it can be inferred that e has type u then it can be inferred that the definition of m is type-correct and is of type

$$C(\vec{s}\,) \cdot \vec{t} \rightarrow u \mid \vec{p} \sqsubseteq \vec{u}.$$

This is expressed in the following inference rule where $\mathsf{self} : C(\vec{s})$ is the implicit type of the pseudo-variable **self** and **returnValue** is a special variable used for type-checking return statements.

$$\frac{H < C, m >= \vec{p} \sqsubseteq \vec{u}\, C(\vec{s}\,)\; m\; \vec{y} : \vec{t} \uparrow u\; \vec{z} : \vec{t}' e \qquad \mathcal{C}.\vec{p} \sqsubseteq \vec{u},\; \mathcal{A}.\underline{\mathsf{self}} : C(\vec{s}\,).\; \vec{y} : \vec{t}.\; \underline{\mathsf{returnValue}} : u.\; \vec{z} : \vec{t}' \vdash e : u}{\mathcal{C}, \mathcal{A} \vdash < C, m >: C(\vec{s}\,) \cdot \vec{t} \rightarrow u \mid \vec{p} \sqsubseteq \vec{u}}$$

Note that $< C, m >: t$ denotes the true inferred type of a method m for class C, which must be included in the declared method type $< C, m >_d: t'$.

2.2.2 Case Analysis

Type-checking Smalltalk programs frequently requires case analysis of a union type. Even though the type of a variable may be a union type when its use throughout an entire method is considered, its type in any particular use may be considered an object type (or one of several object types). This reflects the fact that a variable may reference only one object at a time. It is therefore more accurate, when type-checking an expression containing a variable with a union type, to type-check the expression separately for each type in the union type. The type of the expression is then the union of the separate result types.

$$\frac{\mathcal{C}, \mathcal{A}.v : s \vdash e : u \qquad \mathcal{C}, \mathcal{A}.v : t \vdash e : u}{\mathcal{C}, \mathcal{A}.v : s + t \vdash e : u}$$

Case analysis is useful when used with the following rule: a block whose body is type-incorrect has type *IllegalBlock*. (i.e. if no better type can be inferred for a block it can always be inferred to be illegal)

$$\mathcal{C}, \mathcal{A} \vdash [\vec{y} \mid e] : \mathit{IllegalBlock}$$

Thus, a type-incorrect block will not necessarily cause the containing method to be type-incorrect. This rule, combined with case analysis and the definitions of the normal Smalltalk control structures, provides automatic type discrimination, as shown in the next example.

The typed method definition of **controlToNextLevel** for class **Controller** is shown in Figure 1.4. The **notNil** message is defined to return an object of type *True* for all classes except **UndefinedObject**, for which it is defined to return an object of type *False*. If the type of **aView** is assumed to be *View* then the type of the expression **aView notNil** is *True*. The **ifTrue**: method defined for class **True** has a type

$$<\mathit{True}>\ \mathsf{ifTrue}\!:\ <\mathit{Block\ returns\!:\ P}>\ \uparrow<P>$$

where P is a local type variable for the method. Since the type of aView is *View*, the controller and startUp messages are type-correct and P can be mapped to the return type for the actual block argument.

On the other hand, if the type of aView is assumed to be *UndefinedObject* then the type of the expression aView notNil is *False*. Class False defines the ifTrue: method to have type

$$<False> \text{ ifTrue: } <P> \uparrow <UndefinedObject>$$

so objects of type *False* accept ifTrue: messages with an argument of any type. The method has this type because it ignores its arguments and simply returns nil. Since the type of aView is *UndefinedObject*, the controller message is undefined; the body of the block is illegally typed and the block's type is *IllegalBlock*. However, P can be mapped to *IllegalBlock*, so the block can legally be an argument of the ifTrue: message for class False. Thus, the entire method ControlToNextLevel is type-correct.

2.2.3 Inheritance

Note that our type system does not reflect class inheritance. Even though class Integer is a subclass of class Number, type *Integer* is not a subtype of type *Number*. However, we can specify the type of all subclasses of Number by listing them, i.e. *Integer + Float + Fraction*.[5]

One way that inheritance complicates type-checking is that the type of a method is slightly different for a subclass that inherits it than it is for the class in which it is defined. For example, the type of the receiver is different, and the type of the returned value will be different when the receiver is returned. Type-checking is further complicated by the fact that Smalltalk classes inherit implementation, not specification. For example, class Collection has a number of methods that use the add: message, such as addAll:, but it does not implement or inherit the add: message itself.[6] The add: message is supposed to be implemented by the various subclasses of Collection. Some of its subclasses, such as Array, do not implement add: and thus cannot inherit methods like addAll:. Smalltalk relies on run-time type-checking and the "does not understand" error to detect when an undefined message is sent to an object. We want to detect all such cases at compile-time.

These problems are solved by retype-checking inherited methods in each subclass. The method definition is inherited and *Self* is expanded to refer to the current class. We assume that the hierarchy of typed class definitions H (see Section 2.2.1) contains not only the user declared method definitions for a class, but also any methods that can be meaningfully (i.e. type-correctly) inherited by a class. If H_0 is a partial function from $< C, m >$ pairs to method definitions representing user declared methods then H is derived by extending H_0 to include definition points for inherited methods. In other words, if $H_0 < C, m >$ is a user defined method then $H < C', m >$ has the same definition provided that C' is a subclass of C and $H_0 < C', m >$ is not defined. There are many possible H function. A type-correct H is one in which for all method types derivable from H the declared type of $H < C, m >$ is the same as the type inferred by type-checking.

$$\frac{(\vdash < C, m >_d : t) \Longrightarrow (\mathcal{C}, \mathcal{A} \vdash < C, m >: t)}{\vdash \mathcal{C}, \mathcal{A}}$$

[5] Common union types such as *Integer + Float + Fraction* and *True + False* are presently abbreviated using the global variables *NumberType* and *BooleanType*, respectively. Another approach is to use the object type of an abstract class (classes such as , , and that provide an implementation template for subclasses but have no instances of their own) to automatically denote the union of the types of all its concrete subclasses.

[6] Another example of this is the message in class , shown in Figure 1.5. The message is defined in terms of the message, but class does not inherit or (usefully) define the message. In practice, this is not a problem since is an abstract class and is not intended to have instances.

```
{  localTypeVariables: <P>
   receiverType: <Block returns: (True + False)>
   arguments: aBlock <Block returns: P>
   returnType: <UndefinedObject> }

whileTrue: aBlock
      ↑self value
            ifTrue:
                  [aBlock value.
                  self whileTrue: aBlock]
```

Figure 2.5: The whileTrue: method for class BlockContext.

Strictly speaking, if every method definition in H must type-check then certain methods in abstract classes must be removed in order to have a type-correct H (e.g. the addAll: method in class Collection).

In practice, inherited methods are type-checked upon demand, i.e. when code is first compiled that might cause that method to be invoked. Array can then be a legitimate subclass of Collection; no inherited code that invokes add: will be type-correct.

The benefits of delaying type-checking of inherited methods is that a method is only retype-checked for every subclass that actually inherits it, and that this type-checking is spread out over a large amount of time. A new class does not require any of the methods that it inherits to be type-checked when it is created. Adding a new method to a superclass does not require type-checking it for every subclass that inherits it, nor does adding a new variable. However, changing the type of a method or variable might require a lot of computation to ensure that each of its uses is still type-correct.

2.2.4 Specific Receivers

Some classes have methods that can be executed by only a subset of their instances. For example, the whileTrue: message should be sent only to blocks that return Booleans. We can specify this by using the receiverType: field in the type declaration of the method definition. Recall that if this field is omitted then *Self* defaults to the object type for the class in which the method is being compiled.

A good example (and nearly the only one in Smalltalk) is the whileTrue: method, shown in Figure 2.5. It is easy to imagine other uses of the receiverType: declaration such as the mySum method for class Array, shown in Figure 2.6.

2.3 Extensions to the Type System

2.3.1 Signature Types

A *signature type* is a type (i.e. a set of object and block types) specified by a set of message types. An object type (or block type) t is included in a signature type s if, for each message type $m \in s$, a message of type m sent to an object of type t is type-correct. In other words, an object type is in a

```
{ receiverType: <Array of: Integer>
  blockArguments: sum <Integer> each <Integer>
  returnType: <Integer> }

mySum
      ↑self inject: 0 into: [:sum :each | sum + each]
```

Figure 2.6: The mySum method for class Array.

signature type if it "understands each message in the signature." The message types in a signature type may have occurrences of *Self* to denote the type represented by the signature. Receiver types of *Self* may be omitted. When an object type is being checked for inclusion in a signature type *Self* is instantiated to the object type.

A signature type can be thought to include object types belonging to classes not yet created. Under this interpretation, signature types contain an infinite number of object types. In fact, the type specified by an empty signature contains every possible object type, since every object type understands every message type in the signature. Since signature types specify the largest possible types, procedures that use them exhibit the most polymorphism and so are the most flexible. However, use of signature types prevents the compiler from performing some important kinds of optimizations, since they do not provide enough information about the class of the receiver to allow compile-time binding of message sends.

A signature type is specified by a special kind of type declaration.

<understands: #(tm1 tm2 ...)>

Here, the *tm*'s are strings containing message type specifications. An example will be given shortly. Signature types may also be used to restrict the range of type variables. A local type variable with a signature range would have a declaration of the form

<P understands: #(tm1 tm2 ...)>.

Since such declarations may be recursive, complex mutually recursive signature types can be built.

As an example, consider the occurrencesOf: method for class Collection shown in Figure 2.7. The only restriction placed on the type of the argument anObject is that its corresponding class (or classes) must implement (or inherit) an = (equality) message that will take an argument of type *ElementType* and return a boolean. Another thing to notice about this method definition is the use of the do: message. It is helpful to compare the type information for this use with the type information for the definition of do: given in Figure 2.4.

2.3.2 Typed Metaclass Method Definitions

In regular class method definitions the type *Self* refers to the type of the receiver. It is useful to extend this notion to be able to refer to the type of the class of the receiver. This is done by using the *Self class* type specification.

Similarly, when defining methods in a metaclass it is useful to be able to refer to the type of instances of the metaclass. This is done by using the *InstanceType* type specification. For example, Figure 2.8 shows the typed method definition of new: for the Set metaclass.

```
{ receiverType: <Self>
  arguments: anObject <understands: #('= <ElementType> ↑<False + True>')>
  temporaries: tally <SmallInteger>
  blockArguments: each <ElementType>
  returnType: <SmallInteger> }

occurrencesOf: anObject
    "Answer how many of the receiver's elements are equal to anObject."

    | tally |
    tally ← 0.
    self do: [:each | anObject = each ifTrue: [tally ← tally + 1]].
    ↑tally
```

Figure 2.7: The occurrencesOf: method in class Collection.

```
{ receiverType: <Self>
  arguments: anInteger <SmallInteger>
  returnType: <InstanceType> }

new: anInteger
    "Answer a new instance of size anInteger."

    ↑(super new: (anInteger max: 1)) setTally
```

Figure 2.8: The new: method in Set class.

type	::=	$\underline{<}$ *primaryType* $\underline{>}$ | $\underline{<}$ *signatureType* $\underline{>}$
signatureType	::=	understands: $\underline{\#}$ $\underline{(}$ (*messageType*)$^+$ $\underline{)}$
messageType	::=	$\underline{'}$ {*type*} *unarySelector* $\underline{\uparrow}$ *type* $\underline{'}$
		| $\underline{'}$ {*type*} *binarySelector type* $\underline{\uparrow}$ *type* $\underline{'}$
		| $\underline{'}$ {*type*} (*keyword type*)$^+$ $\underline{\uparrow}$ *type* $\underline{'}$
primaryType	::=	*classType* (of: *unionType*)*
		| *blockType*
		| *varType* {*range*}
		| *unionType*
blockType	::=	Block (of: *unionType*)* returns: *unionType*
range	::=	range: *unionType* | *signatureType*
unionType	::=	*basicType* | *basicType* $\underline{+}$ *unionType*
basicType	::=	*classType* | *varType* | *specialType* | $\underline{(}$ *primaryType* $\underline{)}$
classType	::=	*ClassName* | *ClassName* class | *GlobalName*
varType	::=	*TypeVariable*
specialType	::=	Self | Self class | InstanceType

Figure 2.9: Complete grammar for TS type declarations.

A complete BNF-style grammar of type declarations is shown in Figure 2.9. *ClassName* denotes any Smalltalk class identifier, *GlobalName* denotes any global variable that is used to represent special predefined types such as *BooleanType* (defined to be *True + False*), and *TypeVariable* is any capitalized identifier that is not already used in the scope of the type declaration. Of the special types, Self refers to the type of the pseudo-variables self and super, Self class refers to the type of the metaclass of self and super, and, when compiling methods in a metaclass, InstanceType refers to the type of the instances of the corresponding class (this is useful for type-checking methods like new). *UnarySelector*, *binarySelector*, and *keyword* have the standard Smalltalk meanings.

2.3.3 Type Coercion

Some methods do not have well defined return types (e.g. error:). Others are too general in the sense that the total expressiveness of the method is never used in "real" applications (e.g. legal Array subscripts are always SmallIntegers, though LargePositiveIntegers, for instance, are correct as far as type-checking is concerned). To handle these awkward cases a type coercion mechanism is provided.

A type coercion is achieved by sending the typeIs: message to an expression. The argument of the typeIs: message is a string containing a type declaration. Sending the typeIs: message to an object has no effect on the object or the computation involving it (it is defined to return self). During type-checking, however, the type of the receiver is "coerced" to the type specified by the argument of the typeIs: message. For example, the value of the expression 3 typeIs: '<*Character*>' is the SmallInteger 3. The type of the expression is *Character*.

Coercion is needed in class OrderedCollection. An OrderedCollection is implemented with an Array and two indices indicating the first and last elements. Since the part of the Array be-

```
class Object:

        isLookupKey
            ↑false


class LookupKey:

        isLookupKey
            ↑true
```

Figure 2.10: Implementation of the isLookupKey message.

fore the first element and after the last element is filled with nil, the type of the Array must be *UndefinedObject + ElementType*, where *ElementType* is a type parameter for the class. Thus, when an element is read from the Array, it will have type *UndefinedObject + ElementType* even though it is guaranteed to be of type *ElementType* by virtue of being between the first and last indices. Unfortunately, the type system is not able to discover this on its own. Coercion allows the programmer to declare the "correct" type of the returned object so that uses of OrderedCollection will type-check.

Assigning types to literals sometimes requires coercion. For example, the inferred type of the literal Array #(1 2 3) is *Array of: SmallInteger*. This may or may not be correct, depending on which messages are sent to the Array in the future. If we will be storing Floats in the Array (it just happens to start with all SmallIntegers) then type coercion can be used to "declare" its type to be *Array of: (SmallInteger + Float)*. Typing of literals is discussed further in Section 4.4.

Type coercion is needed in methods that suspend or interrupt execution, such as halt and errorSubscriptBounds. Coercion is also needed in inspectors and debuggers. It seems hard to imagine how to avoid its use in these applications. An optimized code debugger is currently being developed for TS [Zur90].

Note that coercion currently has no effect at run-time. The actual type of the receiver could be anything. In particular, it might be an object that doesn't understand a certain message. This would cause a run-time error. Thus, coercion is dangerous and should be used as little as possible.

2.4 Beyond Signatures

The type system described so far, with explicit union types, case analysis, and signature types, is quite powerful. However, there are still some methods that elude type-checking.

While Smalltalk is a typeless language, it is still possible to define methods that can be used to discriminate between classes of objects and, thus, provide a simple kind of explicit run-time type-checking. An example is the isLookupKey message whose implementation is shown in Figure 2.10. Messages like isLookupKey can be used in a method to "type-check" an argument before sending it a message that it might not understand. Section 2.2.2 shows how the use of these "discriminator" messages can be handled with case analysis.

A typical example of this style of programming is the = (equality) message shown in Figure 2.11.

```
= aLookupKey

    aLookupKey isLookupKey
        ifTrue: [↑self key = aLookupKey key]
        ifFalse: [↑false]
```

Figure 2.11: The = (equality) method in class LookupKey.

```
{ arguments: aLookupKey <understands: #('isLookupKey ↑<False>')>
  returnType: <False> } |
{ localTypeVariables: <P1> <P2>
  arguments: aLookupKey <understands: #('isLookupKey ↑<True>' 'key ↑<P1>')>
  returnType: <P2>
  constraints: '<LookupKeyType> = <P1> ↑<P2>' }

= aLookupKey

    aLookupKey isLookupKey
        ifTrue: [↑self key = aLookupKey key]
        ifFalse: [↑false]
```

Figure 2.12: An unsuccessful attempt at typing = (equality) for class LookupKey.

Equality must be defined between any two objects and should not be subject to run-time errors. This style of implementation meets both of these criteria. The problem is how to type-check such a method.

The closest that we can come to a solution using the type system presented so far is shown in Figure 2.12. The constraints: field will be explained in Chapter 4. For now, assume that it is used to specify signature-like constraints that must be satisfied if the method is to be type-correct. For this method, not even a simple list of constraints is sufficient; we needed to use the logical disjunction of two method types.

The type of this method can be roughly stated as follows: given an argument that when sent the isLookupKey message returns something of type *False*, this method returns something of type *False*, or, given an argument that when sent the isLookupKey message returns something of type *True* and when sent the key message returns something of type *P1*, and if something of type *LookupKeyType* (class type parameter of LookupKey) when sent the = (equality) message with an argument of type *P1* returns something of type *P2*, this method returns something of type *P2*. There must be a simpler way to express what is going on here.

Upon closer examination of what the signatures and constraints say, one finds a fairly direct correspondence with what would happen at execution time with an actual argument. This observation, plus the desire to do type-inference in addition to type-checking, led us to use abstract interpretation. The expressiveness of signature types is absorbed in this new type-inference technique. Signature types are only needed for type-checking recursive methods (which usually do not have constraints related to the recursive message sends). Thus, for most cases, signature types become obsolete and will not be mentioned further.

Another point is that the type of the above method is dependent, to a large extent, on the type of the actual argument. Given a specific call site, it is more than likely that type-checking for this method could be completely resolved, with no constraints, to obtain an object type result. This suggests postponing some aspects of type-checking for a method definition until we have more information from a call site. These ideas will be explored in the next chapter.

Chapter 3

Type-Inference

Smalltalk programmers are used to a typeless language. Type-inference is important because it helps TS keep the feel of Smalltalk by minimizing the amount of type information that must be supplied by the programmer. In the current system, the only type information that must be supplied by the programmer is the types of instance and class variables in a class definition, the argument and return types of primitive methods, and the types of global variables (excluding classes). Types are inferred for method arguments and temporaries, block arguments, method return values, and literals. This means that types can potentially be inferred for over 95% of all methods and for over 90% of all variables.[1]

Optimization requires object types, block types, and (fairly small) union types. Therefore, we would like to be able to infer this kind of type information. This can usually be done for methods with no arguments (i.e. with no inputs of unknown type). Unfortunately, the inferred types for unknown method arguments are sometimes signature types or even more complex expressions (see Section 2.4). This happens when the only occurrence of an argument in a method definition is as the receiver of a message. It can also happen if an argument is used in a message send to another argument. In such cases, it is usually easier to infer types for a method at a call-site, where more is known about the types of the arguments. Although it would be possible to analyize all the potential call-sites of a method to learn the possible types of its arguments, this would produce overly restrictive type information, making it hard to reuse the method in the future without recomputing lots of type information.

Type-inference can also be described as incremental type-checking. When a method is defined, type-inference trys to infer non-signature types for arguments, etc. When this cannot be done, due to lack of outside information, type-inference is postponed until more information is available, say, at a call site. Type-inference then proceeds with site-specific information to give a site-specific result. Type-inference is suspended by keeping a copy of the type environment (inferred types of variables, etc.) being used to type-check the method definition along with type constraints that have yet to be satisfied.

3.1 High-Level Description

An inference system consists of a set of assumptions (the basis) and a set of inference rules (the algorithm). There is a tradeoff between the size of the basis and the complexity of the inference

[1] A survey of the standard Tektronix Smalltalk-80 image revealed that 96.8% of all methods (total = 5145) are not primitives, and that method arguments, temporaries, block arguments, and classes (global variables) make up 90.5% of all variables (total = 9080).

algorithm that depends on the level and accuracy of information required. Obtaining a predetermined amount of information from a small set of assumptions usually requires more powerful inference rules than if a larger set of assumptions were available. Depending on the goals and limitations of the system, the two factors must be balanced to achieve a desirable result. As stated earlier, our goal is to minimize the amount of user-supplied type declarations while still being able to infer enough information to allow compilation and optimization.

In TS, the basis is the implied types of literals (occasionally, an explicit type declaration is required) and the declared types of primitives, instance variables, class variables, and global variables. Types are inferred for method arguments and temporaries, block arguments, and the return types of message sends, blocks, and methods.

The main idea behind most type-inference algorithms, including ours, is to let the type of each variable without an explicit type declaration be represented by a type variable and to obtain values for these type variables using a set of assumptions and inference rules [Mil78, Suz81, MR85, ASU86].

3.1.1 Type Information Flow

Type constraints (i.e. inclusion equations) can be inferred for a given variable (or expression) by considering its definitions and uses. An example of the definition of a variable is the statement v ← 0, which implies that the type of v must include the type of the literal zero. An example of a use of v is the statement self at: v, which implies that the type of the formal argument of the at: message, as defined in the class of self, must include the type of the actual argument v. Type information obtained by considering definitions is called *data-flow type information*. Type information obtained by considering uses is called *requirement-flow type information*.

At the start of the type-inference process, the type of each variable is represented by a unique type variable. A *lower bound* l for a type variable P is an assertion that $l \sqsubseteq P$. Similarly, an *upper bound* u is an assertion that $P \sqsubseteq u$. Each definition of a variable adds data-flow type information to the lower bound of its corresponding type variable (as illustrated by the first example in the preceding paragraph). Each use of a variable adds requirement-flow type information to the upper bound of its type variable (illustrated by the second example).

Upper and lower bounds are calculated using the operations of union (+) and intersection (&) on types. Up to this point we have treated "+" as a type constructor, but it is actually an operator on types. For example, the result of the type expression *Integer* + *Integer* is the object type *Integer*, not a union type. The "+" operator is also commutative. Union types are always "flattened" by unparenthesizing union type elements of union types (i.e. a union type will never contain another union type as an element). Recall, from Section 2.1.1, that a union type is defined to be a nonempty set of object types, block types, or type variables. Given this definition of union types as sets, interpreting base types as singleton union types, and using Definition 2 in Section 5.2 as the definition of set inclusion, the operators + and & are the traditional set operators ∪ and ∩, respectively.

Initially, the lower bound of a type variable is the special type *Nothing* and its upper bound is the special type *Anything*. As each definition or use of a variable is encountered, the bounds of its corresponding type variable are adjusted according to the following inference rules:

$$\frac{\mathcal{C} \vdash s \sqsubseteq p \qquad \mathcal{C} \vdash t \sqsubseteq p}{\mathcal{C} \vdash s + t \sqsubseteq p}$$

$$\frac{\mathcal{C} \vdash p \sqsubseteq s \qquad \mathcal{C} \vdash p \sqsubseteq t}{\mathcal{C} \vdash p \sqsubseteq s \& t}$$

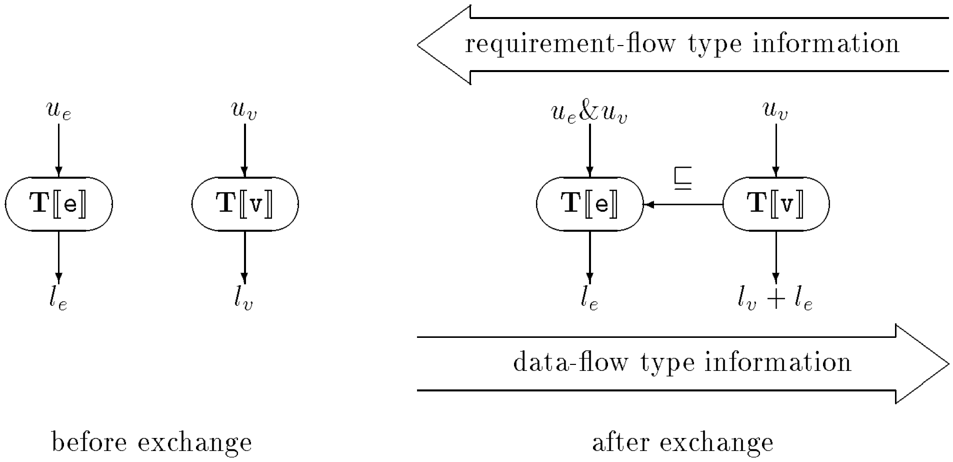

Figure 3.1: Flow of type information between type variables while inferring types for $\mathtt{v} \leftarrow \mathtt{e}$.

where $\mathcal{C}$ is a set on inclusion constraints for type variables, p is a type variable, and s and t are types. As more context is encountered for a variable, the lower bound of its type variable grows "larger" and the upper bound of its type variable grows "smaller."

One consistency check used in the type-inference system is that the lower bound of a type variable must always be included in its upper bound. If adjusting the upper or lower bound of a type variable causes this condition to be violated, an inconsistent assertion (type error) has been encountered at some point in the type-inference process. The inconsistent assertion is not necessarily the one most recently encountered. This can make it difficult to find the original type-error. ML also has this problem. Indeed, it seems to be an acknowledged failing of type-inference systems [Wan86].

The definition of a variable or expression involves the uses of other variables and expressions. At such places, data-flow and requirement-flow type information are propagated simultaneously, but in opposite directions. There are four kinds of places in a method definition where this two-way exchange of type information occurs: assignments, method returns, argument passing, and result passing.

The first place where type information is exchanged is assignment statements. The assignment $\mathtt{v} \leftarrow \mathtt{e}$ means that the type of the variable $\mathtt{v}$ must include the type of the (result of the) expression $\mathtt{e}$, i.e. if $\mathbf{T}$ maps variables and expressions to their corresponding type variables then $\mathbf{T}[\![\mathtt{e}]\!] \sqsubseteq \mathbf{T}[\![\mathtt{v}]\!]$. The type information contained in the lower bound of the type of $\mathtt{e}$ must be added to the lower bound of the type of $\mathtt{v}$. Similarly, any type requirements contained in the upper bound of the type of $\mathtt{v}$ must also be imposed on the upper bound of the type of $\mathtt{e}$. This simultaneous exchange of type information is illustrated in Figure 3.1.

In an assignment, $\mathtt{v} \leftarrow \mathtt{e}$, an upper bound on the type of $\mathtt{v}$ comes from an explicit type declaration. The variable $\mathtt{v}$ could be an instance variable, a class variable, a global variable, or a temporary variable. The scope of a temporary variable is limited lexically to the enclosing method or block definition. For this reason it is easy, in all but a few rare cases, to consider all possible definitions (by assignment) as comprising the de facto type declaration of the temporary variable.

```
foo1: myArg
      | t |
      t ← myArg test.
      ↑t fee


foo2: myArg
      | t |
      myArg test: [:arg | t ← arg].
      ↑t fee
```

Figure 3.2: Methods for which it is difficult to infer the types of temporary variables.

This kind of analysis is complicated by methods like those shown in Figure 3.2. In the foo1: method, the type of t depends entirely on the type of myArg, so it is only possible to tell whether foo1: is type-correct when it is called. In the foo2: method, the type of t exhibits a "higher order" type dependence since its definition appears in a block. The type dependence is more complex because blocks can escape upwards or downwards (as well as through assignment) and because, even if the type of myArg is known, the test: message may have argument dependencies.

In theory, the types of instance, class, and global variables could be inferred in the same way as the types of temporaries. The problem of exporting references in blocks still exists, as well as the fact that the scope of such analysis is potentially the entire class hierarchy. For these reasons, this kind of analysis is restricted to temporary variables.

A lower bound on the type of the expression e comes from an explicit type declaration (e.g. an instance variable), an implicit type assumption (e.g. the type of a literal), or from the type of the result of a message send.

The second place where data-flow and requirement-flow type information are exchanged is return statements. A type variable is associated with the return value of a method. Imagine that $\uparrow_m$ is a special temporary denoting the return value of a method m. Given the return statement, $\uparrow\mathsf{e}$, in the definition for a method m, the return type of the method must include the type of e, i.e. $\mathbf{T}[\![\mathsf{e}]\!] \sqsubseteq \mathbf{T}[\![\uparrow_m]\!]$. The exchange of type information is the same as shown in Figure 3.1 (replacing v with $\uparrow_m$).

Like temporary variables, and given the same exceptions, the return value of a method can only be defined within the lexical scope of the method definition. Similarly, we let the union of the lower bounds of the types of all such return values serve as the declared (inferred) type of a method.

A lower bound on the type of the expression e comes from the same sources as in an assignment statement. An upper bound on the method return type comes from either an explicit type declaration (required of recursive methods) or from a requirement on the result type of a message send that can potentially correspond to this method (this will be discussed in detail later).

The last two places where data-flow and requirement-flow type information are exchanged are in message sends. Suppose a message send m is known to invoke a corresponding method definition m' (exactly how this is determined is discussed later). The types of the arguments of m' must include the types of the arguments of m, i.e. for each argument a_i of m and a'_i of m', $\mathbf{T}[\![\mathsf{a}_i]\!] \sqsubseteq \mathbf{T}[\![\mathsf{a}'_i]\!]$. This is the first exchange of type information.

As with the return value of a method, a type variable is associated with the result of each message send. The type of the result of m must include the return type of m', i.e. $\mathbf{T}[\![\uparrow_{m'}]\!] \sqsubseteq \mathbf{T}[\![\uparrow_m]\!]$. This is the second exchange of type information in a message send.

The lower bounds of argument types of m come from data-flow type information such as assignments and the results of message sends. These bounds flow into the type of m' through its argument types and back out through its return type to the result type of m. Similarly, upper bound type requirements on the result type of m flow into m' through its return type and are subsequently imposed on the argument types of m via the argument types of m'. This is illustrated in Figure 3.3, and based on the relation

$$\begin{array}{l} <a'_0> \textsf{arg1}\text{: } <a'_1> \ldots \textsf{argN}\text{: } <a'_n> \uparrow <\uparrow_{m'}> \\ \sqsubseteq <a_0> \textsf{arg1}\text{: } <a_1> \ldots \textsf{argN}\text{: } <a_n> \uparrow <\uparrow_m> \end{array}$$

The important points of the preceding discussion can be summarized as follows. The lower bound of a type variable describes the kinds of objects that are known to have been stored in the corresponding variable. This information is obtained from explicit type declarations for variables (instance, class, and global variables), from the implicit or explicit types of literals, and from explicit return type declarations for primitive methods. This information is propagated, in the same direction as data-flow during execution, both within a method definition and across message send boundaries. The upper bound of a type variable describes the type requirements (limitations, or potential) placed on the corresponding variable based on its use in particular contexts. This information is also obtained from explicit variable and primitive method type declarations, but is propagated in the opposite direction of data flow.

Data-flow type information represents the most accurate description of a Smalltalk application and, hence, provides a good basis for both type-checking and optimization. Unfortunately, without any explicit type declarations, lower bounds of method arguments are dependent on (user) inputs or on a particular path of execution. In order to increase the resolution of type-checking and to allow more optimization, it is sometimes useful to provide type declarations for method arguments. This is especially true when the programmer knows something about the type of an argument that the type-inference system cannot figure out. For example, method selectors are often used, by convention, to convey type information (e.g. sumFromFloat:). Providing argument types allows requirement-flow type information to also be considered.

It is sometimes possible to infer data-flow type information from requirement-flow type information. Suppose the type variable for a method argument is P and that requirement-flow type information has provided the assertion $P \sqsubseteq Character$. It can be logically inferred, based on the definition of subtyping for object types, that the only legal lower bound for P is also *Character*. This is formalized in the following inference rule:

$$\frac{\mathcal{C} \vdash p \sqsubseteq C(\vec{t})}{\mathcal{C} \vdash C(\vec{t}) \sqsubseteq p}$$

Note that this does not hold in the opposite direction.

3.1.2 Type-Inference Algorithm

Inferring the type of a TS method consists of applying the type-inference rules to the method definition to obtain type inclusion constraints for the arguments and return value of the method. The type inclusion constraints (upper and lower bounds) for a variable define the set of possible types for the variable. The declared type of a variable must be included in this set.

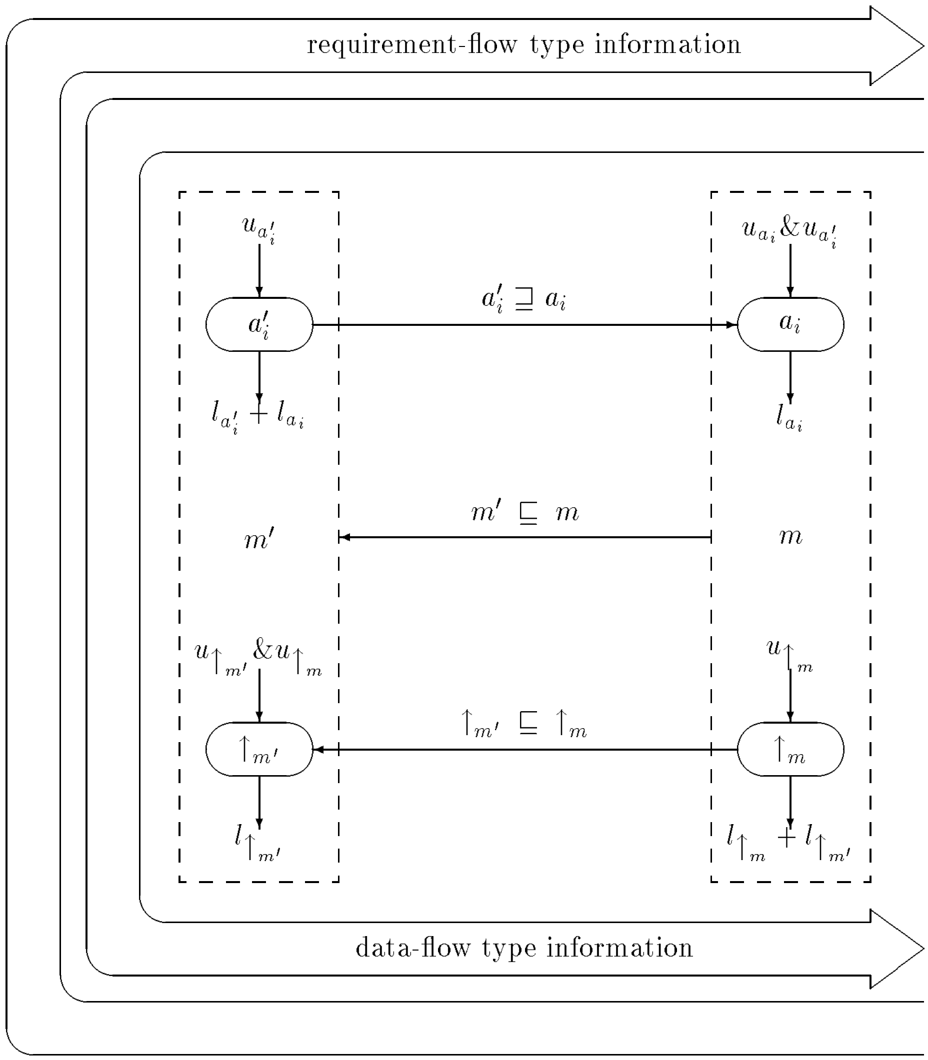

Figure 3.3: Flow of type information between method type m' and message send m.

Type-inference can be likened to the problem of computing a "type" attribute for each node in the abstract syntax tree (parse tree) for a method definition. Given the parse tree for a method definition, the type-inference rules of Section 2.2.1 effectively define the semantic equations for computing the synthesized type attribute. The notation $\mathcal{C}, \mathcal{A} \vdash e : t$ is the equivalent of setting the type attribute of node e to t.

Operationally, a *type tree* is an parse tree whose leaves are types rather than variables or literals. A type tree is an intermediate form used in inferring the type of an expression. The type of an expression is determined by the types of its subexpressions. Once the type of a subexpression is known the exact structure of the subexpression is irrelevant. Instead of using a complete attribute grammar, we simplify the discussion (and the implementation) by using a type tree to represent an intermediate step in the type attribute computation. If a type can be inferred for a node then the subtree rooted at the node is replaced with its inferred type. When a node is replaced, constraint relations between its subexpressions are added to the global constraint set $\mathcal{C}$.

Type-inference for an assignment $v \leftarrow e$ is represented as follows. If $\mathcal{C}, \mathcal{A} \vdash v : t_1$ and $\mathcal{C}, \mathcal{A} \vdash e : t_2$ then $v \leftarrow e$ can be represented by the type tree $t_1 \leftarrow t_2$. According to the type-inference rule for assignment given in Section 2.2.1, if $t_2 \sqsubseteq t_1$ then the type of $v \leftarrow e$ is t_2. This is represented by allowing $t_1 \leftarrow t_2$ to be replaced by t_2 (i.e. the expressions described by $t_1 \leftarrow t_2$ are a subset of the expressions described by t_2) and by adding $t_2 \sqsubseteq t_1$ to the global constraint set $\mathcal{C}$. Type-inference for other statements is represented similarly by the rules shown in Figure 3.4.

The first step in inferring the type of a TS method is to build a parse tree from the source code. Next, a type table is created with an entry for every variable referenced in the parse tree and a special entry "$\uparrow$" representing the return value of the method. Each entry in the type table is given a type value that is either the declared type of the variable or a fresh type variable. The next step is to convert the parse tree to a type tree by replacing each leaf node in the parse tree by a type. A variable node is replaced by its corresponding type table entry and a literal node is replaced by a type based on its class (see Section 2.2.1). An initial set of inclusion equations and message types (type constraints) is obtained by reducing the type tree to a single type using the rules shown in Figure 3.4. The last step in the type-inference process is to find a solution for the inclusion equations that satisfies the message types. This solution gives a type to the method.

Finding a solution for the inclusion equations is equivalent to propagating upper and lower bound type information to the farthest extent among type variables. How this is done is discussed at length in Section 4.1. The essence of solving a set of inclusion equations is simply taking the transitive closure of the $\sqsubseteq$ relation. A "solution" is the upper and lower bounds associated with each type variable. A solution can be interpreted in different ways, as will be seen shortly.

Calculating the type of a message requires simulating method lookup at compile-time. There are two ways to do this. The most accurate technique relies on data-flow type information. If the receiver type of a message type has a lower bound that is a known object type, block type, or union type, then method lookup can be performed to obtain a method type (or set of method types) that must, by definition, be a subtype of the message type constraint. This inclusion check will add more inclusion equations and, hopefully, more type information that will allow further constraint satisfaction.

Type-inference uses data-flow analysis so it suffers from some of the same problems as data-flow analysis. Specifically, a dependency cycle can exist between a message send type constraint and the inclusion equations used to satisfy it. For example, in the sumTotal method shown in Figure 3.5, the temporary sum is assigned in two places. Based on the first assignment, the type of sum is inferred to be *SmallInteger*. This type assumption is used to simulate method lookup for the + message which yields a result of, say, type t. The second assignment to sum requires its type to

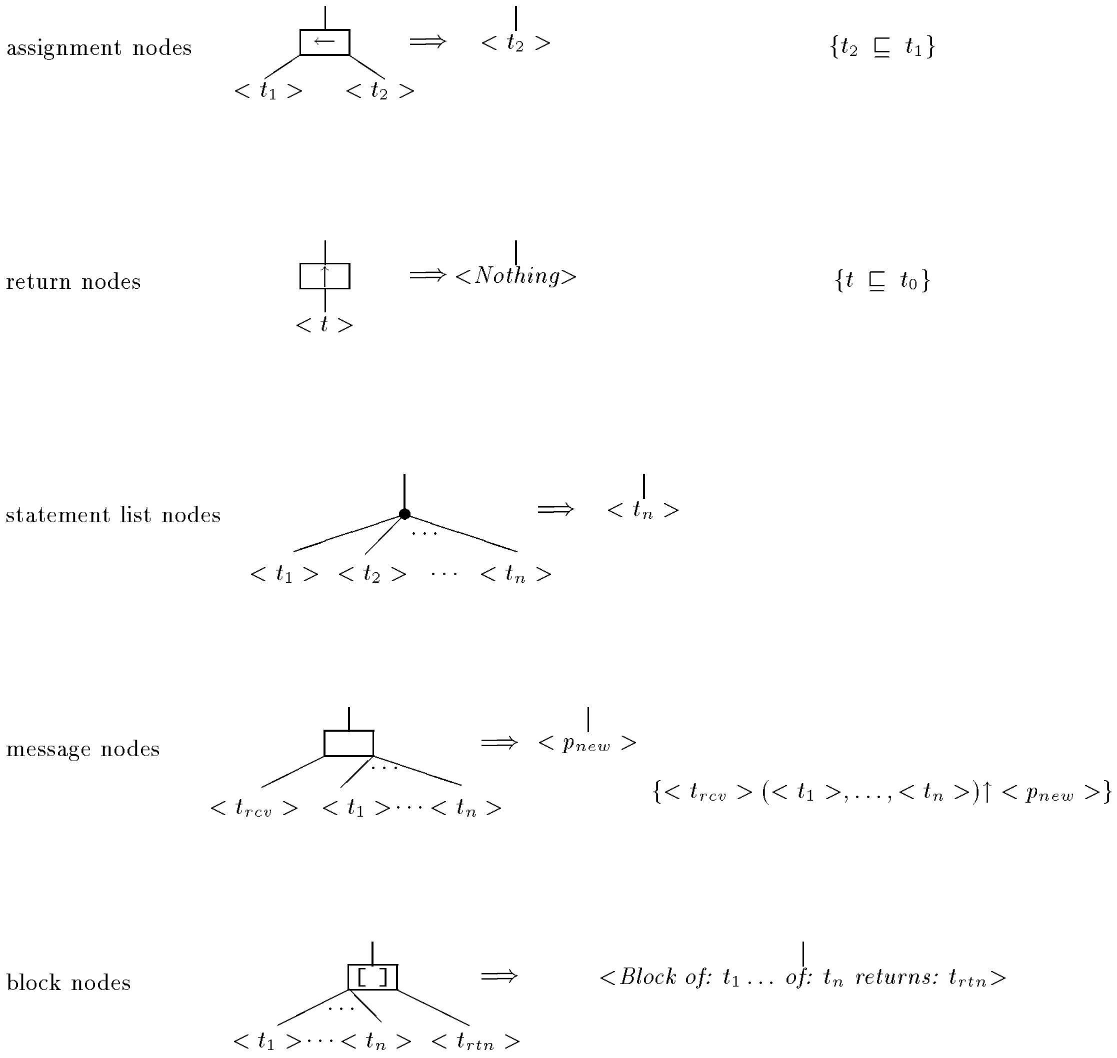

Figure 3.4: Rules for reducing type trees to inclusion equations. The t_i are types, t_0 is the return type of the method, and p_{new} is a fresh type variable. Side-effect inclusion equations and message types are delimited by braces (i.e. $\{\}$).

```
sumTotal
    "Answer the sum of all the receiver's elements."

    | sum |
    sum ← 0.
    self do: [:each | sum ← sum + each].
    ↑sum
```

Figure 3.5: Example of a method requiring iterative incremental type-inference.

also include t, which augments the assumption used to simulate method lookup for the + message. Thus, method lookup for + must be repeated, and so it continues until the type of sum converges.[2] This kind of cyclic dependency requires the type-inference algorithm to be both incremental and iterative.

If the type of the receiver has an upper bound (other than *Anything*) but a lower bound of *Nothing*, the upper bound can be used as an approximation to the lower bound. The results may not be as specific, but they are potentially more useful. Recall that the lower bound tells only what is known so far about a type variable. If an upper bound has a value (other than *Anything*) that will satisfy a type constraint then any legal lower bound is guaranteed to satisfy the constraint.

The last step in the type-inference process is to assign a method type to the method based on the solution of the inclusion equations. A method type consists of a message type (for the formal arguments and return type), a set of inclusion equations, and a set of message types (constraints). Chapter 4 shows how most of this information can be discarded, keeping only the main message type.

If method lookup cannot be simulated due to a lack of type information, it is usually best to postpone type-inference until more type information is available. In such cases, the method type is marked as having unsatisfied constraints. At a call site, data-flow type information may provide the information necessary to completely satisfy the constraints, and a site-specific return type can be computed. If not, the constraints on the call cannot be satisfied and the method containing the call will also be constrained.

3.2 An Example

As an extended example of type-inference and of how type information flows throughout the system, consider the four methods - (minus), differenceFromFloat: differencePrimitive: and negated in class Float, shown in Figure 3.6. These methods are used in the double dispatching implementation of subtraction [HJ90].

Consider the definition for the - (minus) method. Assuming no type declarations are given, the type table entries for self, aNumber, and "↑" (the return type of the method) are initialized to *Float*, *P1*, and *P0*, respectively, where *P0* and *P1* are fresh type variables. The body of the method consists of a return statement and a single message send. The parse tree is converted to a type tree by replacing the leaves aNumber and self by their respective types. The type tree is reduced

[2]The type of is guaranteed to converge since there are a finite number of methods in any given image.

```
- aNumber
    ↑aNumber differenceFromFloat: self


differenceFromFloat: aFloat
    ↑aFloat differencePrimitive: self


differencePrimitive: aFloat
    <primitive: 42 <Float>
    'RTL code for floating point subtraction'>
    self primitiveFailed


negated
    ↑0.0 - self
```

Figure 3.6: Double dispatched subtraction in class Float.

to give the message type

$$<P1> \text{differenceFromFloat:} <Float> \uparrow <P2>$$

and the inclusion equation $P2 \sqsubseteq P0$, where $P2$ is a fresh type variable.

The receiver type $P1$ of the message type has no upper or lower bound so method lookup cannot be simulated. Type-inference for this method can go no farther without more information about the argument aNumber. The type of the - (minus) method is marked as constrained and is denoted by

$$<Float> \text{ - } <P1> \uparrow <P0>.$$
$$\{<P1> \text{differenceFromFloat:} <Float> \uparrow <P2>, <P2> \sqsubseteq <P0>\}$$

which captures the constraint on the message send differenceFromFloat:.

Type-inference for the differenceFromFloat: method proceeds exactly the same as for the - (minus) method. The type tree is reduced to give the message type

$$<P1> \text{differencePrimitive:} <Float> \uparrow <P2>$$

and the inclusion equation $P2 \sqsubseteq P0$. Its method type is

$$<Float> \text{differenceFromFloat:} <P1> \uparrow <P0>.$$
$$\{<P1> \text{differencePrimitive:} <Float> \uparrow <P2>, <P2> \sqsubseteq <P0>\}.$$

Even though we use the same names for type variables here as in the method type for - (minus), they are different variables and will be given different names when necessary. In practice, the argument aFloat would have its type declared as *Float* in both differenceFromFloat: and differencePrimitive:.

This points out a weakness of type-inference in that inferred types are often much more general than is required by actual use or convention. Type-inference must consider all possible uses as

$\{$ returnType: $<Self>$ $\}$
primitiveFailed
    self error: 'a primitive has failed'

Figure 3.7: Typed method definition for primitiveFailed in class Object.

equally likely when only a few may actually arise in practice. Combining type-inference and explicit type declarations maximizes optimization and minimizes programmer effort. This is what we have strived for in TS. The point of this example is to show that type-inference can produce correct type information if given enough context.

The method definition for the differencePrimitive: method, shown in Figure 3.6, contains a primitive call. In Smalltalk-80 this primitive call looks like

<primitive: 42>

and indexes an assembly code routine in the interpreter primitive table. In TS, primitives are not "black box" routines, but are coded explicitly in a machine independent register transfer language (RTL) [GGJ*88]. The type declaration following the index refers to the type of the object returned by the primitive if it succeeds. This type declaration is required. However, since programmers generally do not write primitives, requiring type declarations for them is not a serious matter.

If the last statement in a method definition is not a return statement, the Smalltalk parser automatically (and invisibly) inserts a "↑self" statement. The actual code for the fail case is, therefore

self primitiveFailed.
↑self.

The types of self, aFloat, and "↑" are initialized to *Float*, *P1*, and *P0*, respectively. The type tree is reduced to give the message type

$<Float>$ primitiveFailed $\uparrow<P2>$

and the inclusion equation $Float \sqsubseteq P0$. The receiver of the message is of type *Float* so method lookup can be simulated.

The typed method definition for the primitiveFailed method, defined in class Object, is shown in Figure 3.7. It has the corresponding unconstrained method type

$<Object>$ primitiveFailed $\uparrow<Object>$.

This method is inherited from class Object by class Float for use in the differencePrimitive: method. Notice, however, that the receiver type in the method type for primitiveFailed is *Object* and not *Float*, causing the check

$<Object>$ primitiveFailed $\uparrow<Object> \sqsubseteq <Float>$ primitiveFailed $\uparrow<P2>$

to fail. Recompiling the definition for primitiveFailed in class Float, in turn, requires the method type for error: and the methods that error: calls to be recompiled. Assume the resulting method type for primitiveFailed is

$<Float>$ primitiveFailed $\uparrow<Float>$.

Since there are no constraints (by assumption), the check

$$<Float>\ \textsf{primitiveFailed}\ \uparrow<Float>\ \sqsubseteq\ <Float>\ \textsf{primitiveFailed}\ \uparrow<P2>$$

adds $Float \sqsubseteq P2$ to the set of inclusion equations ($Float \sqsubseteq Float$ is trivially true and can be ignored).

All constraints are satisfied and the resulting method type is

$$<Float>\ \textsf{differencePrimitive:}\ <P1>\ \uparrow<Float>.$$

Finally, consider type-inference for the **negated** method for class **Float** shown in Figure 3.6. The types of **self** and "↑" are initialized to *Float* and *P0*, respectively. Type tree reduction gives the message type

$$<Float>\ \text{-}\ <Float>\ \uparrow<P1>$$

and the inclusion equation $P1 \sqsubseteq P0$.

The receiver type permits method lookup and the previously computed method type

$$<Float>\ \text{-}\ <P1'>\ \uparrow<P0'>.$$
$$\{<P1'>\ \textsf{differenceFromFloat:}\ <Float>\ \uparrow<P2'>,\ <P2'> \sqsubseteq <P0'>\}$$

is obtained. Note that type variables have been renamed to distinguish them from those currently in use. The inclusion check

$$<Float>\ \text{-}\ <P1'>\ \uparrow<P0'>\ \sqsubseteq\ <Float>\ \text{-}\ <Float>\ \uparrow<P1>$$

adds $Float \sqsubseteq P1'$ and $P0' \sqsubseteq P1$ to the set of inclusion equations. The inclusion is, however, contingent on two constraints in the method type for - (minus), a message type and an inclusion equation. The inclusion equation $P2' \sqsubseteq P0'$ is simply added to the set of inclusion equations. $P1'$ now has a lower bound of *Float* so the message type can also be checked. Method lookup in class **Float** obtains the method type

$$<Float>\ \textsf{differenceFromFloat:}\ <P1''>\ \uparrow<P0''>.$$
$$\{<P1''>\ \textsf{differencePrimitive:}\ <Float>\ \uparrow<P2''>,\ <P2''> \sqsubseteq <P0''>\}.$$

The inclusion check

$$<Float>\ \textsf{differenceFromFloat:}\ <P1''>\ \uparrow<P0''>\ \sqsubseteq\ <P1'>\ \textsf{differenceFromFloat:}\ <Float>\ \uparrow<P2'>$$

adds $P1' \sqsubseteq Float$, $Float \sqsubseteq P1''$, and $P0'' \sqsubseteq P2'$ to the set of inclusion equations. As before, this inclusion is contingent on an inclusion equation $P2'' \sqsubseteq P0''$, which is added to the set of inclusion equations, and a message type. $P1''$ has a lower bound of *Float* so the message type can be checked. Method lookup for differencePrimitive: in class Float yields the method type

$$<Float>\ \textsf{differencePrimitive:}\ <P1\dagger>\ \uparrow<Float>.$$

The inclusion check

$$<Float>\ \textsf{differencePrimitive:}\ <P1\dagger>\ \uparrow<Float>\ \sqsubseteq\ <P1''>\ \textsf{differencePrimitive:}\ <Float>\ \uparrow<P2''>$$

adds the inclusion equations $P1'' \sqsubseteq Float$, $Float \sqsubseteq P1\dagger$, and $Float \sqsubseteq P2''$.

This time there are no constraints. If the inclusion checks can be verified by obtaining a consistent solution for the inclusion equations then type-inference for the message send will be

$$
\begin{array}{rcl}
P1 & \sqsubseteq & P0 \\
Float & \sqsubseteq & P1' \\
P0' & \sqsubseteq & P1 \\
P2' & \sqsubseteq & P0' \\
P1' & \sqsubseteq & Float \\
Float & \sqsubseteq & P1'' \\
P0'' & \sqsubseteq & P2' \\
P2'' & \sqsubseteq & P0'' \\
P1'' & \sqsubseteq & Float \\
Float & \sqsubseteq & P\dagger \\
Float & \sqsubseteq & P2''
\end{array}
$$

(a)

$$
\begin{array}{rcccl}
Float & \sqsubseteq & P0 & \sqsubseteq & Anything \\
Float & \sqsubseteq & P1 & \sqsubseteq & Anything \\
Float & \sqsubseteq & P0' & \sqsubseteq & Anything \\
Float & \sqsubseteq & P1' & \sqsubseteq & Float \\
Float & \sqsubseteq & P2' & \sqsubseteq & Anything \\
Float & \sqsubseteq & P0'' & \sqsubseteq & Anything \\
Float & \sqsubseteq & P1'' & \sqsubseteq & Float \\
Float & \sqsubseteq & P2'' & \sqsubseteq & Anything \\
Float & \sqsubseteq & P\dagger & \sqsubseteq & Anything
\end{array}
$$

(b)

Figure 3.8: Inclusion equations and solution.

```
{ returnType: <Self> }
negated
        ↑0.0 - self
```

Figure 3.9: Typed definition of negated for class Float.

complete. The complete set of inclusion equations is shown in Figure 3.8(a). Using the transitivity of $\sqsubseteq$ and the inference rules from Section 3.1.1, the solution shown in Figure 3.8(b) is obtained.

Since the negated method has no arguments or temporaries, only computation of its return type is required. *P0* has a lower bound of *Float* and the typed method definition of negated is shown in Figure 3.9.

There are several important observations to be made here. First, the only user supplied type information in this example is the return type of the subtraction primitive. All other type information was inferred. Second, to infer the precise type for the negated method it is necessary to infer types for the other methods first. The order in which the other three methods are processed is irrelevant. Third, if the negated method is processed before the other methods then it will have constraints. The constraints will go away if it is reprocessed after the other methods have been processed.

These observations suggest that type declarations will eventually reach a steady state if the type-checker continually reprocessed every method in the system.

Chapter 4

Implementation

Type-inference for a method m, as described in Chapter 3, consists of reducing m's type tree to an initial set of inclusion equations and a set of message types. These two sets define the method type of m. In practice, these steps are overlapped for the sake of efficiency. The implementation of the type-inference algorithm is divided roughly into two components. One reduces the type tree and processes message types and the other solves inclusion equations.

A *type tree* is a structural copy of a parse tree with types as leaves. Type trees store unsatisfied type constraints and keep track of structural (positional) dependencies between message types. The latter is important for two reasons. First, since data-flow type information is preferred, a post-order traversal of the type tree provides a natural order in which to process message types. This natural ordering is lost if message types are simply collected into a set. Second, if case-analysis is being performed at a call-site, the type-inference algorithm must know which message types are involved and which are not.

A *type graph* is a directed graph whose nodes are type variables and whose edges denote subtype relations. An edge in a type graph from $p1$ to $p2$ is an assertion that $p2 \sqsubseteq p1$.[1] A cycle in a graph means that all type variables in the cycle are equal. A type graph is used to solve sets of inclusion equations.

The type-inference algorithm uses these two components, in conjunction with a type environment (a database of type information), to try to infer a type for each node in a type tree. This may involve type computations on several type trees at once. In general, type-inference will be able to infer types for everything only if it is applied to a top-level method (one with no arguments) or if supplemental type information is declared.

4.1 Type Graphs

A type graph is a directed graph whose nodes are type variables. Each type variable has an upper and lower type bound. A completely unrestricted type variable has an upper bound of *Anything* and a lower bound of *Nothing*. A type variable whose value is completely determined will have identical upper and lower bounds.

A type graph is created from a set of type variables and from a set of inclusion equations. Each inclusion equation relates a type variable to another type variable or to a type. There are three kinds of inclusion equations,

- $p2 \sqsubseteq p1$

[1] A useful trick for remembering this relation is to note that $p2 \sqsubseteq p1$ is sometimes written $p2 \leq p1$ which looks a lot like $p2 \longleftarrow p1$.

- $p \sqsubseteq t$
- $t \sqsubseteq p$

where p, $p1$, and $p2$ are type variables and t is a type. Edges are added to a type graph when inclusion equations of the form $p2 \sqsubseteq p1$ are added to the set on inclusion equations for a method. Adding inclusion equations of the other forms may result in corresponding changes to upper or lower bounds information or may result in an error.

Recall that the reduction of a type tree yields an initial set of inclusion equations and a possibly empty set of message types. Simulating method lookup for a message type is based on the ability to determine a type value for the receiver of the corresponding message send. If the type of the receiver is a type variable then its initial value must come from the initial set of inclusion equations. If method lookup can be simulated for a message type, it may result in more inclusion equations and, hence, more information about the values of the type variables in the type graph.

The purpose of a type graph is to compute type bounds for its type variables by solving an evolving set of inclusion equations. At any point in the type-inference algorithm, the type graph represents the current set of inclusion equations and any type relations that can be inferred from this set. When a new inclusion equation is encountered, it is added directly to the type graph, in the form of a new edge.

Inclusion equations are added to the type graph one at a time. As new edges are created, the type graph computes the transitive closure of the inclusion equations encountered so far. Changes to the upper or lower bound of a node require a check that the lower bound is a subtype of the upper bound. Equations of the form $t \sqsubseteq p$ change the lower bound of p to be the union of its old lower bound and t. Changing the lower bound of a node updates the lower bounds of its predecessors. Similarly, equations of the form $p \sqsubseteq t$ change the upper bound of p to be the intersection of its old upper bound and t. Changing the upper bound of a node updates the upper bounds of its successors. Equations of the form $p2 \sqsubseteq p1$ add a new edge to the graph and update upper and lower bounds appropriately. This may result in a two-way exchange of type information. At any point in time, the lower bound of a node is the union of all the lower bounds of its successors, and its upper bound is the intersection of all the upper bounds of its predecessors.

4.2 Type Trees

Inferring the type of a new method may require information about its definition and its uses. Unfortunately, the later information is use-site-specific and not available when the new method is being compiled. Therefore, a certain amount of type-inference may have to be performed at each use-site. For such methods, it is desirable to repeat only the site-specific type computations at any use-site. This is accomplished through the use of type trees.

The type of a method should describe the requirements for arguments and, given argument types that satisfy these requirements, the characteristics of the result. In this sense, the type tree for a method *is* its type. Abstract interpretation of the type tree performs computations in the type domain analogous to those performed in the data domain when interpreting a parse tree. Parse trees are compiled into object code for efficient execution. Similarly, type trees are reduced (when possible) to message types for efficient type-inference.

A type tree is initially a simplified copy of a parse tree. Type-inference reduces the type tree for a method as much as possible using rules similar to those shown in Figure 3.4. The only differences are that a message send can only be reduced if method lookup can be simulated, and a block can only be reduced if its statement list reduces to a single type. This may result in partially reduced

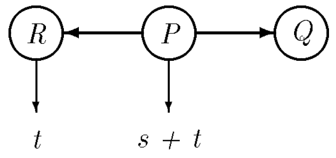

Figure 4.1: Example type graph resulting from inclusion equations.

type trees. The significance of a partially reduced type tree is that use-site type information must be supplied before further reduction can take place. The goal of type-inference is to completely reduce all type trees.

The type of a method, i.e. the root of a type tree, is represented by a *method type*. A method type consists of a type environment (see Section 4.3), the class in which the corresponding method is defined, the method selector, and, if the type tree could not be fully reduced, the partially reduced type tree for the method. In effect, a partially reduced type tree for a method represents the type constraints that must be satisfied by the context of a message send if the method can be invoked from the site, i.e. the type of the method that has yet to be abstractly interpreted.

4.3 Type Environments

A type environment holds much of the type information associated with a method. It also provides the basic mechanics for type-inference. A type environment contains a map from variable names to types, a type graph, a generator of new type variables, and error recovery support. All requests to update the type graph, to obtain the type of a variable, or to compare two types are sent to a type environment.

A type environment can be asked whether or not two types are equal, whether or not one type is a subtype of another, or to evaluate a type (replace type variables with more precise type values) based on the current bounds information in the type graph.

When a type is evaluated in a type environment, a new type is returned in which some type variables are replaced by more precise information. The value of a type variable in a type graph depends on its bounds as well as on its context in the graph. The value of a completely unrestricted type variable (its lower bound is *Nothing* and its upper bound is *Anything*) is itself. The value of a type variable that is an alias for another type variable is the value of the other type variable. The value of type variables with restricted bounds requires more explanation.

Consider the inclusion equations

$$R \sqsubseteq P \qquad Q \sqsubseteq P \qquad s \sqsubseteq P \qquad t \sqsubseteq R$$

where, P, Q, and R are type variables and s and t are types. According to the inference rules given in Section 3.1.1 the value of P is $s + t + Q$. The type graph resulting from these inclusion equations is shown in Figure 4.1. Nodes and edges are drawn with thick lines, upper and lower bounds are drawn with thin lines, and upper bounds of *Anything* and lower bounds of *Nothing*

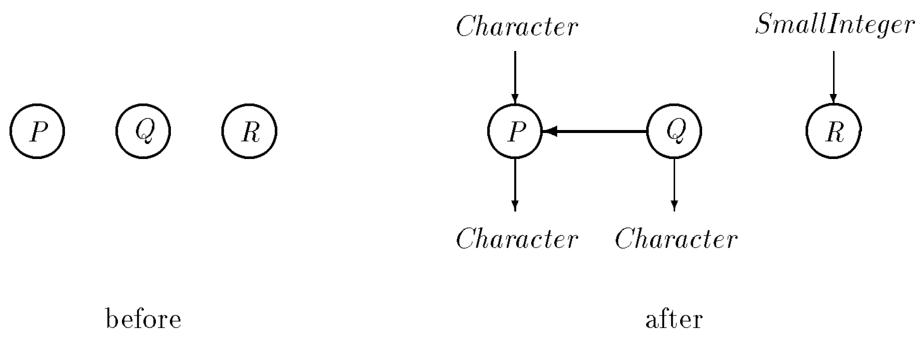

Figure 4.2: Type graph before and after a message type inclusion check.

are not shown. The correct value for P in the type graph is the union of its lower bound with all immediate successors whose lower bounds are *Nothing* (the lower bounds of successors with non-*Nothing* lower bounds have already been accounted for in P's lower bound). This rule is applied, in general, to obtain the value of a type variable with a (non-*Nothing*) lower bound. Analogously, if a type variable has no lower bound but has an (non-*Anything*) upper bound, its value is defined as its upper bound intersected with all immediate predecessors whose upper bounds are *Anything*.

A type environment determines whether or not one type includes another by comparing the two types using the subtyping rules given in Chapter 2. Comparisons involving type variables that appear in the type graph produce inclusion equations that must be added to the type graph. For example, checking that

$$<\!\mathit{Array\ of{:}\ P}\!>\ \mathsf{at{:}}\ <\!\mathit{SmallInteger}\!>\ \uparrow<\!P\!>\ \sqsubseteq\ <\!\mathit{Array\ of{:}\ Character}\!>\ \mathsf{at{:}}\ <\!R\!>\ \uparrow<\!Q\!>$$

produces the inclusion equations

- $P = \mathit{Character}$ ($P \sqsubseteq \mathit{Character}$, $\mathit{Character} \sqsubseteq P$),
- $R \sqsubseteq \mathit{SmallInteger}$, and
- $P \sqsubseteq Q$,

and affects the type graph as shown in Figure 4.2.

4.4 Type-Inference

Type-inference for a method definition proceeds as follows. A new type environment is initialized with information in the type table for the standard pseudo-variables **self**, **super**, **true**, **false**, **nil**, and **thisContext**. The return type of the method is represented by a special entry in the type table, denoted "$\uparrow$," which is always initialized to the fresh type variable *P0*.

Whenever a fresh type variable is generated, it is entered into the type graph as a new node. The type graph is initialized with nodes for the class type parameters of the context class, if any, and any range bounds (inclusion equations usually of the form $p \sqsubseteq t$, where, p is a class type parameter and t is a type) are added to the type graph.

A parse tree is created from the method definition text using a slightly modified version of the Smalltalk-80 recursive descent parser. The modified parser looks for a method prefix of type declarations, special primitive definitions, and explicit type coercions. The parse tree is used for optimization and code generation, but contains more information than is necessary for type-inference.

Nodes are added to the type graph for any local type variables declared. Any type declarations for method arguments, temporaries, or block arguments are entered into the type table. Method arguments, temporaries, and block arguments without explicit type declarations are assigned fresh type variables as types.

A type tree is produced by traversing the parse tree, copying its essential structure, and replacing leaf nodes with types. The type environment, with the type table and type graph, is placed in the root of the type tree.

The next step in the type-inference process is to traverse the type tree, attempting to reduce each node in the tree to a type. There are usually two pre-conditions that must be satisfied before a node can be reduced. Each of its children must have been reduced to a type (this is not always necessary if the child is a block) and the node must satisfy any peculiar type constraints. A post-order traversal is used to reduce the type tree in a bottom-up fashion. Nodes that cannot be reduced are left intact, perhaps with partially reduced children. Reductions are based on the rules given in Figure 3.4.

Leaves

Initially, the leaves of the type tree are the types of the corresponding variables or literals in the parse tree. Instance variables, class variables, and global variables have their types declared. These types are entered in the type table and used to create the type tree. Leaf nodes for method arguments, temporaries, block arguments, and pseudo-variables are assigned fresh type variables as discussed earlier. Leaf nodes for literals are converted to types based on the class of the literal. This conversion is always correct for numbers, characters, strings, and symbols. Literal Arrays without explicit type declarations require some assumptions.

Smalltalk literals are not considered constants, but are simply objects created at compile-time. Type-inference assumes that the initial value of a literal Array describes its intended type. For example, the type of the literal Array #(0 0 0) is assumed to be *Array of: SmallInteger*. Since it is possible to store any object into any Array at any time, this assumption might not be valid. Determining the *true* type of a literal Array may require extensive flow analysis and, in general, may not even be possible. For all known uses of literal Arrays in the standard Smalltalk-80 image, type assumptions for literal Arrays based on initial values appear to be valid. Where there is doubt, type coercion can be used to "declare" the type of a literal (see Section 2.3.3). An interesting case to consider is the type of the literal Array #() (i.e. the empty Array).

Assignments

An assignment node in the type tree consists of a variable node and an expression node. If the expression node can be reduced to a type (the variable node can always be reduced) then the type environment checks that the type of the expression node is a subtype of the type of the variable node. The check is performed by adding the inclusion equation

$$<\textit{type of expression node}> \sqsubseteq <\textit{type of variable node}>$$

to the type graph. Other such checks are performed in a similar fashion. If this check fails then type-checking for the method fails. If this check succeeds, type-inference continues with the assignment node being reduced to the type of its expression.

Returns

A return node consists of a single expression node. The type of the expression is a lower bound of the return type of the method (*P0*). A return node is reduced to the type *Nothing* for the reasons mentioned in Section 2.2.1.

Statement Lists

A statement list node consists of a list of expressions. If each expression in the list can be reduced to a type then the statement list node can be reduced. If a statement list node can be reduced then it is reduced to the type of the last expression in its list.

Blocks

A block node consists of argument nodes and a statement list node. Blocks are treated much like methods. The statement list is reduced as much as possible, but complete reduction may have to wait until actual arguments are supplied (via the value message). If the statement list can be reduced to a single type then the block node is reduced to a block type.

The return type of a block depends on many things. If a block contains no return statements then its return type is the type of the last statement in its statement list. If the last statement in a block is a return statement then, as before, the return type of the block is the type of its last statement (i.e. *Nothing*). If the last statement is not a return statement, but there are return statements present elsewhere, the return type is the type of the last statement unioned with a fresh type variable, which represents the type of the return statement.[2] (If a conditional statement contains returns in each branch, it is assumed that the returns are combined at the top-level.)

Block nodes are special because the code they represent may not ever be executed. If not, there is no reason for the code to be type correct, or even to waste time type-checking it. It may be possible to reduce a message node even if it has unreduced block nodes in its arguments. (Normally, a message send can only be reduced if all of its arguments can be reduced to types.) Control flow in Smalltalk is implemented by selectively evaluating one block out of several choices. Such control flow messages often ignore one or more of their (block) arguments (e.g. the ifTrue: message in class False). An argument that is ignored can have any type. This is achieved by letting the formal argument type be an unrestricted type variable that is not referenced anywhere else in the method type. Thus, in the context of any particular use of the method type, the formal argument type will include any actual argument type.

If a block with no arguments cannot be reduced it is most likely due to a type error. This was represented in Section 2.2.2 by letting the type of such blocks be *IllegalBlock*. Message types with *IllegalBlock* arguments would type-check if the corresponding formal argument was ignored, i.e. *IllegalBlock* could be bound to a type variable with no other restrictions. Since the exact form of *IllegalBlock* is arbitrary, we permit unreduced block nodes to behave in the same way.

[2] *Nothing* cannot be used since it would be absorbed into the block's normal return type, i.e. $t + Nothing = t$.

Messages

Next to reducing block nodes, reducing message nodes is the most complicated part of the type-inference process. If the receiver and argument nodes can be reduced to types, a message type is constructed consisting of the message selector, the type of the receiver node, the types of the argument nodes, and a fresh type variable for the return type. The type environment then checks the message type. There are three possible results: success, failure, or unknown. If the check is successful (if method lookup can be simulated for the message type and it is type correct), the message node is reduced to its corresponding return type and type-inference continues. If the check fails (e.g. incompatible argument types), type-inference for the method fails. If the result is unknown, the message node cannot be reduced, indicating a type constraint, and type-inference continues.

A message type is a block type with a name (the message selector). The receiver in a message type is the first argument. Subtyping for message types is the same as subtyping for block types with the additional requirement that the two selectors are equal. A method type consists of a message type, a possibly empty type tree, and a type environment.

Type-checking a message type proceeds as follows. If the type of the receiver cannot be evaluated in the type environment to obtain an object type, a block type, or a union type, method lookup cannot be performed and the result is unknown. If the type of the receiver can be evaluated to give an object or block type (each element of a union type is processed individually) then method lookup is performed.

If no method is found by method lookup then the result is unknown. This indicates either that an inappropriate message is being sent to an object or that the programmer has not yet defined the method in question. The system prevents the execution of any code that knowingly refers to the method being type-checked until one of these situations is remedied. If a method is found without a corresponding method type, the result is also unknown.

Assuming a method with a method type is found, the type environment must check that the message type in question includes the method's message type and that any constraints in the method type are satisfied. The main point of this exchange is to check that the types of the actual arguments are subtypes of the types of the corresponding formal arguments, and to infer the return type of the message from the types of the actual argument types.

The type environment checks for appropriate subtype relations between receiver types, argument types, and return types of the two message types. If any subtype relation fails, type-checking for the message type fails. If all inclusion checks succeed then the type-inference algorithm is recursively applied to the type tree of the method type. This is a strict operation in the sense that failure on "unknownness" is passed back through any depth of recursion as the result of the top-level message type type-inference.

A method type will have constraints (a non-empty type tree) if some aspect of type-inference for its definition could not be completely resolved when it was compiled. Its definition might contain a message send to an argument of unknown type. Or, perhaps, it contains a message send corresponding to an untyped or undefined method. In any case, type-inference was not resolved because it lacked certain local or global type information when it was compiled. Such a method type contains a type graph consisting of all information necessary to resume the suspended type-inference once more information is available.

If the result of type-checking the message type for a message node is unknown then the message node cannot be reduced and any changes to the type graph must be "backed out." This is accomplished by keeping a copy of the initial type graph. Also, if the result is unknown due to a non-reducible type tree (unsatisfiable constraints) in a method type then no reference to the type

```
- aNumber
      ↑aNumber differenceFromFloat: self
```

Figure 4.3: Subtraction in class Float.

tree or method type is retained in the message node. There are several reasons for this. First, if non-reducible type trees were grafted into message nodes it could cause an exponential growth in the size of the current type tree. Second, a grafted type tree would be like a stop-time snapshot of a method type. Method types can change (shrink or grow) over time, due to the fact that constraints are dependent on the types of other methods, and we always want to use the most recent version.

If a method type with constraints is encountered while checking a message type, its type graph is merged with the current type graph. When argument subtyping is checked, edges may be introduced that allow the exchange of type information between the two graphs. It is type information derived from actual argument types that allows unresolved constraints to be resolved. Constraint resolution usually has a cascading effect on other constraints. It can also refine the return type.

4.5 An Example (Revisited)

The example from Section 3.2 is repeated here in greater detail and from an implementation perspective. Consider type-inference for the four methods - (minus), differenceFromFloat: differencePrimitive: and negated in class Float. The definition for the - (minus) method is shown in Figure 4.3. Assuming no type declarations are given, the type table entries for self, aNumber, and "↑" (the return type of the method) are initialized to *Float*, *P1*, and *P0*, respectively. The body of the method consists of a return statement and a single message send.

When the type tree for the - (minus) method is created, the arguments of the message send are converted to types based on type table information. To reduce the message node, the message type

$$<P1>\ \textsf{differenceFromFloat:}\ <Float>\ \uparrow<P2>$$

is created, with a fresh type variable *P2*, and checked. The receiver type *P1* evaluates to itself in the type environment, because it is unrestricted. Method lookup cannot be performed so this is as far as type-inference can proceed for the - (minus) method. The type tree and the resulting (trivial) type graph are depicted in Figure 4.4.

The inferred type of a method is inserted into the method definition as a type declaration prefix. Types are shown for method arguments, temporaries, and block arguments. The return type of the method and any constraints are also shown. The type displayed for a variable is obtained by evaluating its type table entry in the type environment.

The types inferred for the - (minus) method are shown in Figure 4.5 (although they are not very helpful). Its method type is denoted (for convenience) by

$$\begin{array}{l} <Float>\ \text{-}\ <P1>\ \uparrow<P0>. \\ \{<P1>\ \textsf{differenceFromFloat:}\ <Float>\ \uparrow<P2>\} \end{array}$$

but really looks more like Figure 4.4. Note that this method type is slightly different from the one inferred for the - (minus) method in Section 3.2. This method type does not have the inclusion

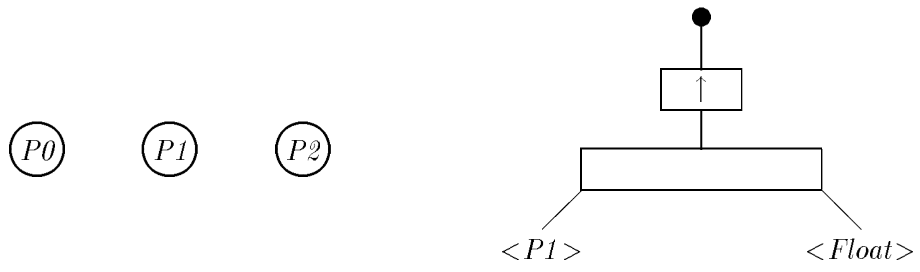

Figure 4.4: Type graph and type tree for - (minus) method.

```
{ localTypeVariables: <P0> <P1> <P2>
  arguments: aNumber <P1>
  returnType: <P0>
  constraints: '(<P1> differenceFromFloat: <Self> ↑<P2>)' }
- aNumber
      ↑aNumber differenceFromFloat: self
```

Figure 4.5: Minus method with inferred type information.

```
{ localTypeVariables: <P0> <P1> <P2>
  arguments: aFloat <P1>
  returnType: <P0>
  constraints: '(<P1> differencePrimitive: <Self> ↑<P2>)' }

differenceFromFloat: aFloat
    ↑aFloat differencePrimitive: self
```

Figure 4.6: The differenceFromFloat: method with inferred types.

```
differencePrimitive: aFloat
    <primitive: 42 <Float>
    'RTL code for floating point subtraction'>
    self primitiveFailed
```

Figure 4.7: Definition for differencePrimitive: in class Float.

equation $P2 \sqsubseteq P0$ in its constraints. The inclusion equation comes from reducing the return node in the type tree. This has not been done yet so the inclusion equation is not present, either in the type graph or in the constraints.

Type-inference for the differenceFromFloat: method proceeds exactly the same as for the - (minus) method. The inferred types and method definition are shown in Figure 4.6. Its method type is

$$\begin{array}{l} <\!\mathit{Float}\!> \textsf{differenceFromFloat:} <\!\mathit{P1}\!> \uparrow<\!\mathit{P0}\!>. \\ \{<\!\mathit{P1}\!> \textsf{differencePrimitive:} <\!\mathit{Float}\!> \uparrow<\!\mathit{P2}\!>\}. \end{array}$$

The method definition for the differencePrimitive: method, shown in Figure 4.7, is a primitive call. The types of self, aFloat, and "↑" are initialized to *Float*, *P1*, and *P0*, respectively. Reducing a primitive node requires checking that *P0* includes the return type of the primitive. The receiver self is reduced to *Float* and the message type

$$<\!\mathit{Float}\!> \textsf{primitiveFailed} \uparrow<\!\mathit{P2}\!>$$

is created and checked by the type environment. The receiver type evaluates (trivially) to *Float* and method lookup can be performed.

The typed method definition for the primitiveFailed method defined in class Object is shown in Figure 3.7. The inherited recompiled version has the unconstrained method type

$$<\!\mathit{Float}\!> \textsf{primitiveFailed} \uparrow<\!\mathit{Float}\!>.$$

Since there are no constraints (by assumption), the type environment checks the relation

$$<\!\mathit{Float}\!> \textsf{primitiveFailed} \uparrow<\!\mathit{Float}\!> \sqsubseteq <\!\mathit{Float}\!> \textsf{primitiveFailed} \uparrow<\!\mathit{P2}\!>$$

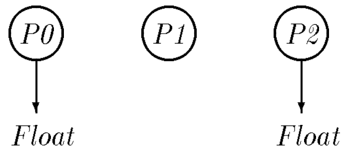

Figure 4.8: Type graph for differencePrimitive.

```
{ localTypeVariables: <P1>
  arguments: aFloat <P1>
  returnType: <Self> }

differencePrimitive: aFloat
    <primitive: 42 <Self>
    'RTL code for floating point subtraction'>
    self primitiveFailed
```

Figure 4.9: Typed definition for differencePrimitive: in class Float.

which results in a lower bound of *Float* for *P2* and the reduction of the message node to *P2*.

The implicit "↑self" statement is reduced, which adds the relation $Float \sqsubseteq P0$ to the type graph, and type-inference for this method is complete. The type graph is shown in Figure 4.8.

Type declarations are extracted from the type environment and inserted into the method definition, as shown in Figure 4.9. All nodes except *P1* are pruned from the type graph (see Section 4.6) and the resulting method type is

$$<Float>\ \text{differencePrimitive: } <P1>\ \uparrow<Float>.$$

Finally, consider type-inference for the negated method for class Float shown in Figure 4.10. The types of self and "↑" are initialized to *Float* and *P0*, respectively. The literal 0.0 is converted

```
negated
    ↑0.0 - self
```

Figure 4.10: The negated method for class Float.

to *Float*, based on its class, and **self** is converted to *Float*. The message type

$$<\mathit{Float}> \text{ - } <\mathit{Float}> \uparrow <\mathit{P1}>$$

is created for the return node expression and checked.

Looking up the - (minus) method in the receive type results in

$$\begin{array}{l} <\mathit{Float}> \text{ - } <\mathit{P1}'> \uparrow <\mathit{P0}'>. \\ \{<\mathit{P1}'> \ \mathsf{differenceFromFloat:} \ <\mathit{Float}> \uparrow <\mathit{P2}'>\} \end{array}$$

Note that type variables have been renamed to distinguish them from those in the current type environment. The type graph for this method type is merged with the current type graph. After the inclusion

$$<\mathit{Float}> \text{ - } <\mathit{P1}'> \uparrow <\mathit{P0}'> \sqsubseteq <\mathit{Float}> \text{ - } <\mathit{Float}> \uparrow <\mathit{P1}>$$

is checked, the type graph shown in Figure 4.11(a) results. The inclusion is, however, contingent on a constraint in the method type for - (minus). $P1'$ evaluates to *Float* so the constraint can be checked. Method lookup in class Float obtains the method type

$$\begin{array}{l} <\mathit{Float}> \ \mathsf{differenceFromFloat:} \ <\mathit{P1}''> \uparrow <\mathit{P0}''>. \\ \{<\mathit{P1}''> \ \mathsf{differencePrimitive:} \ <\mathit{Float}> \uparrow <\mathit{P2}''>\}. \end{array}$$

Once again, its type graph is merged and the inclusion check

$$<\mathit{Float}> \ \mathsf{differenceFromFloat:} \ <\mathit{P1}''> \uparrow <\mathit{P0}''> \sqsubseteq <\mathit{P1}'> \ \mathsf{differenceFromFloat:} \ <\mathit{Float}> \uparrow <\mathit{P2}'>$$

yields the graph shown in Figure 4.11(b). As before, this inclusion is contingent on a constraint. $P1''$ evaluates to *Float* and this constraint can also be checked. Method lookup for **differencePrimitive:** in class **Float** yields the method type

$$<\mathit{Float}> \ \mathsf{differencePrimitive:} \ <\mathit{P1}\dagger> \uparrow <\mathit{Float}>.$$

Its type graph is merged and the inclusion check

$$<\mathit{Float}> \ \mathsf{differencePrimitive:} \ <\mathit{P1}\dagger> \uparrow <\mathit{Float}> \sqsubseteq <\mathit{P1}''> \ \mathsf{differencePrimitive:} \ <\mathit{Float}> \uparrow <\mathit{P2}''>$$

yields the graph shown in Figure 4.11(c).

This time there are no constraints so the message node for **differencePrimitive:** in the **differenceFromFloat:** method type tree is reduced to $P2''$. The return node is reduced, adding the relation $P2'' \sqsubseteq P0''$ to the type graph, and all constraints for the **differenceFromFloat:** method are satisfied. The type graph, at this point, is shown in Figure 4.11(d).

The message node for **differenceFromFloat:** in the - (minus) method type tree is now reduced to $P2'$. The reduction of its return node adds the relation $P2' \sqsubseteq P0'$ to the type graph and all constraints for the - (minus) method are satisfied. The resulting type graph is shown in Figure 4.11(e).

Finally, back in the type tree for **negated**, the message node for - (minus) is reduced to *P1*. The return is reduced, adding the relation $P1 \sqsubseteq P0$, and the final type graph is shown in Figure 4.11(f). Since the **negated** method has no arguments or temporaries, only computation of its return type is required. *P0* evaluates to *Float* and the typed method definition of **negated** is shown in Figure 4.12.

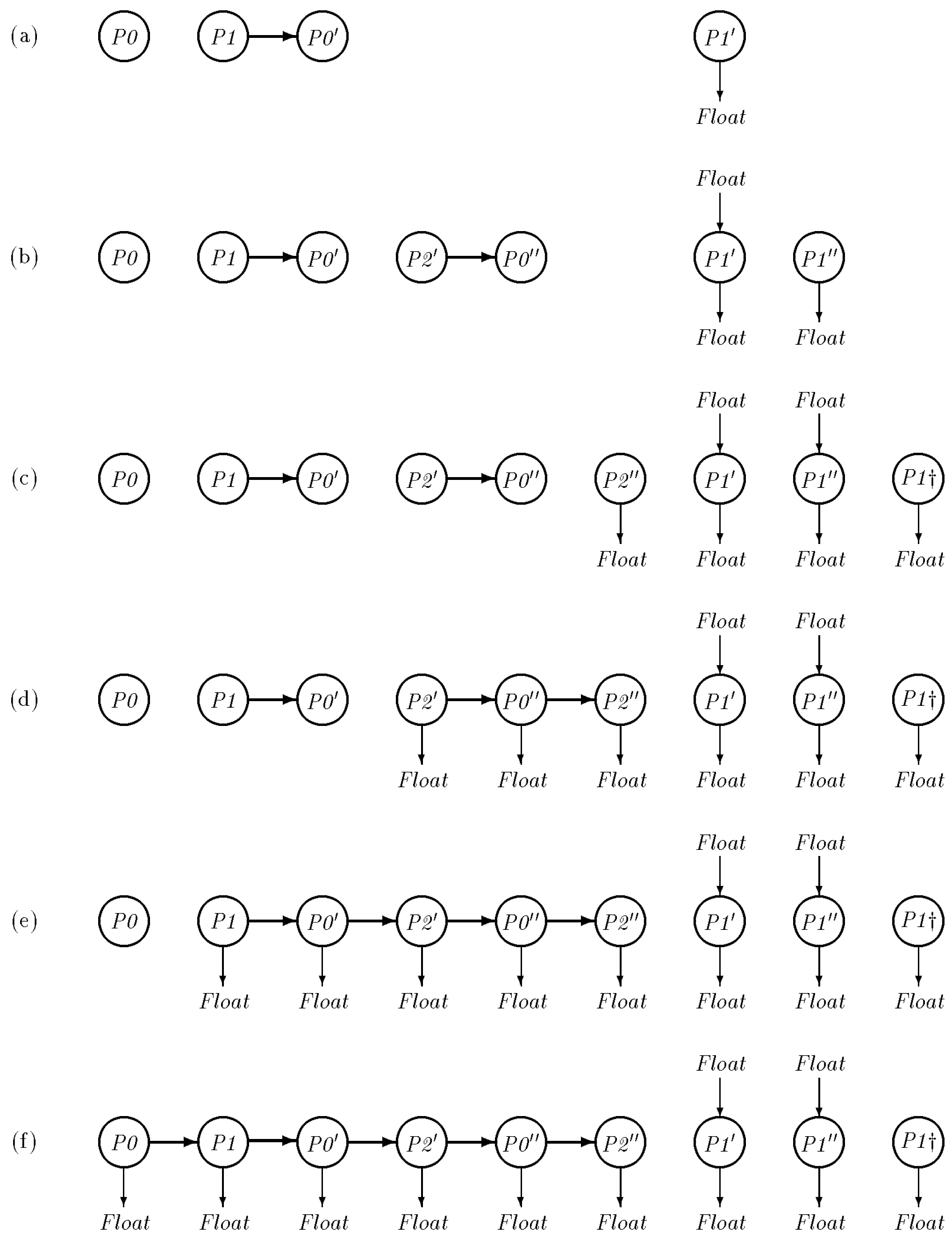

Figure 4.11: Type graphs used in type-inference for `negated`.

```
{ returnType: <Self> }
negated
    ↑0.0 - self
```

Figure 4.12: Typed definition of negated for class Float.

4.6 Graph Compression

To reduce the space and time complexity of type-inference, useless information is removed from the type graphs associated with the type of a method at the end of the type-inference process. Two type graphs are retained for future use for each method: one in the method type and the other in the parse tree. Each of these graphs may be pruned to different extents, depending on the amount of type information that needs to be retained. For a method type, everything can be removed from its type graph except the type variables referenced (perhaps indirectly) in its message type or in the constraints. In addition to this information, the type graph in the parse tree must retain information about the types of temporaries, block arguments, and the return type of the method. This is used when performing high-level optimization on parse trees [Loy88]. The return types in the parse tree and in the method type are kept in different forms.

Type graphs may be reduced by removing unreferenced nodes, collapsing cycles, and by replacing references to nodes with equal upper and lower bounds with the bound value. A node is removed by adding a new edge between every possible predecessor/successor pair and then removing the node and all of its adjacent edges. Cycles are collapsed by first partitioning the graph into strongly connected components. Each strongly connected component represents an equivalence class of nodes that can be collapsed into a single node.

A representative node for each equivalence class is determined based on the following priority scheme. An arbitrary node from the highest of three priority levels is chosen as a representative of the class. Nodes representing class type parameters have the highest priority, nodes representing local type variables of the original method are next, and nodes representing other type variables have the lowest priority. For each non-representative node of an equivalence class, its predecessors and successors are added to those of the representative node, references to it are removed from the representative node, and it is aliased (via table indirection) to the representative.

The non-representative nodes are aliased rather than being completely removed from the type graph because they may be referenced in upper or lower bound type expressions of other nodes. Removing them completely from the type graph would require traversing and possibly rebuilding every upper and lower bound type expression in the graph. This is also the reason why nodes with equal upper and lower bounds are not currently compressed out of type graphs.

Chapter 5

Formal Definition of the Type System

It is not obvious how to implement the type-checking rules specified by Chapter 2. This chapter provides an operational specification of both type-checking and type-inference.

This chapter assumes a working knowledge of Kamin's denotational semantics of Smalltalk [Kam88]. We have made an effort to use the same notation and to avoid name conflicts. We prove that our type-inference algorithm is correct relative to Kamin's semantics.

5.1 Type Domains

Type domains are often described as sets of values, as in the weak ideal model of MacQueen et al. [MPS84], but this is not necessary for our purposes. Our type domain is purely syntactic. The abstract syntax of types introduced in Section 2.1 is expressed using slightly different notation and with some minor changes in Figure 5.1. The letters at the left denote elements of their respective domains.

The type system that we have implemented has a slightly different type domain than that described here. A type can also be defined by the operations that it implements, i.e. by its behavior. A *signature type* is denoted by a set of message types. A signature type can be replaced by a union type comprised of the object types that implement all the methods represented in it. Thus, signature types add no theoretical power to the type system. However, they have the practical advantage of

`C`	$\in$	ClassName	::=	lexical domain of class identifiers
`p`, `q`, `r`	$\in$	TypeVar	::=	lexical domain of type variable identifiers
		ObjectType	::=	ClassName $\underline{(}$ Type* $\underline{)}$
		BlockType	::=	Type* $\underline{\rightarrow}$ Type
		UnionType	::=	Type $\underline{+}$ Type
`s`, `t`, `u`	$\in$	Type	::=	ObjectType | BlockType | TypeVar | UnionType

Figure 5.1: Abstract syntax of types.

Definition 1 (Basic type inclusion)

$$
\begin{array}{l}
\sqsubseteq \,:\, \mathsf{Type} \times \mathsf{Type} \rightarrow \mathsf{Boolean} \\
\mathtt{C}(\vec{\mathtt{t}}) \ \sqsubseteq \ \mathtt{C}'(\vec{\mathtt{t}}') \equiv (\mathtt{C} = \mathtt{C}') \wedge (\vec{\mathtt{t}} \sqsubseteq_* \vec{\mathtt{t}}') \wedge (\vec{\mathtt{t}}' \sqsubseteq_* \vec{\mathtt{t}}) \\
\vec{\mathtt{s}} \rightarrow \mathtt{t} \sqsubseteq \vec{\mathtt{s}}' \rightarrow \mathtt{t}' \equiv (\vec{\mathtt{s}}' \sqsubseteq_* \vec{\mathtt{s}}) \wedge (\mathtt{t} \sqsubseteq \mathtt{t}') \\
\mathtt{p} \sqsubseteq \mathtt{p}' \equiv \mathtt{p} \ \mathtt{=} \ \mathtt{p}' \\
\mathtt{s}_1 \ \mathtt{+} \ \mathtt{s}_2 \ \mathtt{+} \ \ldots \ \mathtt{+} \ \mathtt{s}_m \sqsubseteq \mathtt{t}_1 \ \mathtt{+} \ \mathtt{t}_2 \ \mathtt{+} \ \ldots \ \mathtt{+} \ \mathtt{t}_n \\
\qquad \equiv \textit{for each } 1 \leq i \leq m \textit{ there exists a } 1 \leq j \leq n \textit{ such that } \mathtt{s}_i \sqsubseteq \mathtt{t}_j \\
\\
\sqsubseteq_* \,:\, \mathsf{Type^*} \times \mathsf{Type^*} \rightarrow \mathsf{Boolean} \\
<> \sqsubseteq_* <> \equiv \textit{true} \\
\mathtt{s} \cdot \vec{\mathtt{s}} \sqsubseteq_* \mathtt{t} \cdot \vec{\mathtt{t}} \equiv (\mathtt{s} \sqsubseteq \mathtt{t}) \wedge (\vec{\mathtt{s}} \sqsubseteq_* \vec{\mathtt{t}})
\end{array}
$$

not needing to be redefined each time a new class is added to the system. Therefore, they are often used to define the ranges of type variables.

5.2 Solving Inclusion Constraints

The subtype relation $\sqsubseteq$ on types was first introduced in Section 1.2.3. For the purpose of defining the type-inference algorithm, we define $\sqsubseteq$ to be a binary boolean-valued function on types as shown in Definition 1.

Inferring types for message sends involves relating argument types to a result type via an abstract method type. In Prolog or ML this is done by unifying the actual and formal argument types to obtain a substitution function from type variables to types, and using this function to evaluate the formal return type. We perform a similar operation to obtain solutions for a set of inclusion equations. The difference is that unification is based on determining equality [Rob65] whereas we are interested in the weaker relation of inclusion.

Solving a set of inclusion equations was described in Chapter 4 using a type graph. While this is how the actual implementation works, we now describe a slightly different solution technique. The result, in either case, is a value for each type variable that satisfies the inclusion equations.

An inclusion equation is represented by a pair of types $< \mathtt{t}, \mathtt{t}' >$, which asserts that $\mathtt{t}$ includes $\mathtt{t}'$ i.e. $\mathtt{t}' \sqsubseteq \mathtt{t}$. A set of inclusion equations is represented (for notational reasons) by a list of type pairs. A type variable map (TVM) is a function from type variables to types representing one solution for a set of inclusion equations. A TVM list represents the set of all solutions for a set of inclusion equations.[1]

Inclusion equations have the following semantic domains.

$$
\begin{array}{lll}
\nu \in & \mathsf{IncEqn} & = \mathsf{Type} \times \mathsf{Type} \\
\vec{\nu} \in & \mathsf{IncEqn^*} & \\
\sigma \in & \mathsf{TypeVarMap} & = \mathsf{TypeVar} \rightarrow \mathsf{Type} \\
\vec{\sigma} \in & \mathsf{TypeVarMap^*} &
\end{array}
$$

[1] In the interest of uniformity and to reduce notational clutter we represent sets using list notation. Thus, the reader will notice frequent use of the phrase "the list representing the set." We also implicitly assume coercions for using set operators on lists and for injections into and projections out of sum domains.

Given a type $\mathtt{t}$ and a TVM σ, the function $\mathbf{evaluate}(\mathtt{t}, \sigma)$ returns a new type obtained by substituting types for type variables in $\mathtt{t}$.

$$
\begin{array}{l}
\mathbf{evaluate} : \mathsf{Type} \times \mathsf{TypeVarMap} \rightarrow \mathsf{Type} \\
\mathbf{evaluate}(\mathtt{C}(\vec{\mathtt{t}}), \sigma) \equiv \mathtt{C}(\mathbf{evaluate^*}(\vec{\mathtt{t}}, \sigma)) \\
\mathbf{evaluate}(\vec{\mathtt{s}} \rightarrow \mathtt{t}, \sigma) \equiv \mathbf{evaluate^*}(\vec{\mathtt{s}}, \sigma) \rightarrow \mathbf{evaluate}(\mathtt{t}, \sigma) \\
\mathbf{evaluate}(\mathtt{p}, \sigma) \equiv \text{if } \sigma(\mathtt{p}) = \bot \text{ then } \mathtt{p} \text{ else } \mathbf{evaluate}(\sigma(\mathtt{p}), \sigma) \\
\mathbf{evaluate}(\mathtt{t}_1 \texttt{ + } \mathtt{t}_2 \texttt{ + } \ldots \texttt{ + } \mathtt{t}_n, \sigma) \\
\qquad\qquad \equiv \mathbf{evaluate}(\mathtt{t}_1, \sigma) \texttt{ + } \mathbf{evaluate}(\mathtt{t}_2, \sigma) \texttt{ + } \ldots \texttt{ + } \mathbf{evaluate}(\mathtt{t}_n, \sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathbf{evaluate^*} : \mathsf{Type^*} \times \mathsf{TypeVarMap} \rightarrow \mathsf{Type^*} \\
\mathbf{evaluate^*}(<>, \sigma) \equiv <> \\
\mathbf{evaluate^*}(\mathtt{t} \cdot \vec{\mathtt{t}}, \sigma) \equiv \mathbf{evaluate}(\mathtt{t}, \sigma) \cdot \mathbf{evaluate^*}(\vec{\mathtt{t}}, \sigma)
\end{array}
$$

Note that **evaluate** is recursively applied to $\sigma(\mathtt{p})$. In order to guarantee termination, we do not allow TVMs to contain bindings of the form $\mathtt{p} \mapsto f(\mathtt{p})$.

Given a type $\mathtt{t}$, the function $\mathbf{typevars\text{-}of}(\mathtt{t})$ returns a list representing the set of all type variables in $\mathtt{t}$. Its definition is analogous to that of **evaluate** and is not shown.

Given two TVMs

$$\sigma = \{\mathtt{p}_1 \mapsto \mathtt{t}_1, \mathtt{p}_2 \mapsto \mathtt{t}_2, \ldots, \mathtt{p}_m \mapsto \mathtt{t}_m\} \text{ and } \sigma' = \{\mathtt{p}'_1 \mapsto \mathtt{t}'_1, \mathtt{p}'_2 \mapsto \mathtt{t}'_2, \ldots, \mathtt{p}'_n \mapsto \mathtt{t}'_n\}$$

the boolean function $\mathbf{consistent}(\sigma, \sigma')$ returns true if each type variable appearing in both σ and σ' has the same (or an equivalent) value, i.e.

$$
\mathbf{consistent}(\sigma, \sigma') \equiv \begin{cases} \text{true}, & \text{if for each } \mathtt{q} \in \{\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_m\} \cap \{\mathtt{p}'_1, \mathtt{p}'_2, \ldots, \mathtt{p}'_n\} \\ & \quad \mathbf{evaluate}(\sigma(\mathtt{q}), \sigma) = \mathbf{evaluate}(\sigma'(\mathtt{q}), \sigma') \\ \text{false}, & \text{otherwise.} \end{cases}
$$

The combination of two TVMs σ and σ' is defined iff $\mathbf{consistent}(\sigma, \sigma') = \text{true}$. The combination is computed by adding to σ all elements of σ' not already defined by σ, i.e.

$$\mathbf{combine}(\sigma, \sigma') \equiv \sigma \cup \{\mathtt{p}_i \mapsto \mathtt{t}_i \mid \sigma(\mathtt{p}_i) = \bot \wedge \mathtt{p}_i \mapsto \mathtt{t}_i \in \sigma'\}.$$

Notice that σ and σ' may have different, but consistent, bindings for a type variable so $\mathbf{combine}(\sigma, \sigma')$ is not necessarily the union of σ and σ'.

Given two TVM lists $\vec{\sigma}$ and $\vec{\sigma}'$ the function $\mathbf{merge}(\vec{\sigma}, \vec{\sigma}')$ returns a single TVM list representing the set of all TVMs that are a union of a pair of consistent TVMs, one from $\vec{\sigma}$ and one from $\vec{\sigma}'$, i.e.

$$\mathbf{merge}(\vec{\sigma}, \vec{\sigma}') \equiv \{\mathbf{combine}(\sigma_i, \sigma'_j) \mid \sigma_i \in \vec{\sigma} \wedge \sigma'_j \in \vec{\sigma}' \wedge \mathbf{consistent}(\sigma_i, \sigma'_j)\}.$$

The function $\mathbf{includes}(\mathtt{t}, \mathtt{t}', \vec{\mathtt{p}}, \vec{\mathtt{u}})$ takes two types, a list of type variables, and a list of corresponding range types, and returns a TVM list $\vec{\sigma}$ representing the set of TVMs mapping elements of $\vec{\mathtt{p}}$ to types that, upon evaluation, allow $\mathtt{t}$ to include $\mathtt{t}'$, i.e.

$$\mathbf{includes}(\mathtt{t}, \mathtt{t}', \vec{\mathtt{p}}, \vec{\mathtt{u}}) \equiv \{\sigma \mid \mathbf{evaluate}(\mathtt{t}', \sigma) \sqsubseteq \mathbf{evaluate}(\mathtt{t}, \sigma)\}.$$

Type variables appearing in $\mathtt{t}$ or $\mathtt{t}'$ but not in $\vec{\mathtt{p}}$ are treated as type constants (i.e. parameterless object types). If **includes** returns an empty TVM list (i.e. $\vec{\sigma} = <>$) then $\mathtt{t}$ does not include $\mathtt{t}'$ under

Definition 2 (Solving inclusion equations)

includes : $\mathsf{Type} \times \mathsf{Type} \times \mathsf{TypeVar^*} \times \mathsf{Type^*} \rightarrow \mathsf{TypeVarMap^*}$

$\mathbf{includes}(\mathtt{C}(\vec{\mathtt{t}}), \mathtt{C}'(\vec{\mathtt{t}}'), \vec{\mathtt{p}}, \vec{\mathtt{u}})$ *(object types)*

$$\equiv \quad \textit{if}\ \mathtt{C} = \mathtt{C}'\ \textit{then}\ \mathbf{merge}(\mathbf{includes^*}(\vec{\mathtt{t}}, \vec{\mathtt{t}}', \vec{\mathtt{p}}, \vec{\mathtt{u}}), \mathbf{includes^*}(\vec{\mathtt{t}}', \vec{\mathtt{t}}, \vec{\mathtt{p}}, \vec{\mathtt{u}}))\ \textit{else} <>$$

$\mathbf{includes}(\vec{\mathtt{s}} \rightarrow \mathtt{t}, \vec{\mathtt{s}}' \rightarrow \mathtt{t}', \vec{\mathtt{p}}, \vec{\mathtt{u}})$ *(block types)*

$$\equiv \quad \mathbf{merge}(\mathbf{includes}(\mathtt{t}, \mathtt{t}', \vec{\mathtt{p}}, \vec{\mathtt{u}}), \mathbf{includes^*}(\vec{\mathtt{s}}', \vec{\mathtt{s}}, \vec{\mathtt{p}}, \vec{\mathtt{u}}))$$

$\mathbf{includes}(\mathtt{p}, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}})$, $\mathbf{includes}(\mathtt{t}, \mathtt{p}, \vec{\mathtt{p}}, \vec{\mathtt{u}})$ *(type variables)*

$$\begin{aligned} \equiv \quad & \textit{if}\ \mathtt{p} = \mathtt{p}_i \in \vec{\mathtt{p}} \wedge \mathtt{p} \notin \mathbf{typevars\text{-}of}(\mathtt{t}) \wedge \mathtt{t} \sqsubseteq \mathtt{u}_i\ \textit{then} < \{\mathtt{p} \mapsto \mathtt{t}\} > \\ & \textit{else if}\ \mathtt{p} = \mathtt{t}\ \textit{then} < \{\} > \\ & \textit{else} <> \end{aligned}$$

$\mathbf{includes}(\mathtt{t} = \mathtt{t}_1\ \mathtt{+}\ \mathtt{t}_2\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{t}_m, \mathtt{t}' = \mathtt{t'}_1\ \mathtt{+}\ \mathtt{t'}_2\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{t'}_n, \vec{\mathtt{p}}, \vec{\mathtt{u}})$ *(union types)*

$$\begin{aligned} \equiv \quad & \textit{let}\ \vec{\mathtt{p}}' = \vec{\mathtt{p}} \cap (\mathtt{t} \cup \mathtt{t}') \\ & \textit{and}\ \vec{\mathtt{u}}' = \mathbf{get\text{-}ranges}(\vec{\mathtt{p}}', \vec{\mathtt{p}}, \vec{\mathtt{u}}) \\ & \textit{and}\ \vec{\sigma} =< \{\mathtt{p}'_1 \mapsto \psi_1, \mathtt{p}'_2 \mapsto \psi_2, \ldots, \mathtt{p}'_k \mapsto \psi_k\} \\ & \qquad\qquad |\mathtt{p}'_i \in \vec{\mathtt{p}}' \wedge (\psi_i \in \mathbf{P}((\mathtt{t} \cup \mathtt{t}') - \mathtt{p}'_i) - \{\} \vee \mathtt{p}'_i \in (\mathtt{t} \cap \mathtt{t}') \wedge \psi_i = \bot) > \\ & \textit{in}\ \ \textit{if}\ \vec{\mathtt{p}}' =<>\ \textit{then}\ \mathbf{match\text{-}merge}(\mathtt{t}', \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}}, < \{\} >) \\ & \qquad \textit{else} \bigcup_{\sigma \in \vec{\sigma}} (\mathbf{match\text{-}merge}(\mathbf{evaluate}(\mathtt{t}', \sigma), \mathbf{evaluate}(\mathtt{t}, \sigma), \vec{\mathtt{p}} - \vec{\mathtt{p}}', \vec{\mathtt{u}} - \vec{\mathtt{u}}', < \sigma >)\ ) \end{aligned}$$

match-merge : $\mathsf{Type} \times \mathsf{Type} \times \mathsf{TypeVar^*} \times \mathsf{Type^*} \times \mathsf{TypeVarMap^*} \rightarrow \mathsf{TypeVarMap^*}$

$\mathbf{match\text{-}merge}(\mathtt{t}' = \mathtt{t'}_1\ \mathtt{+}\ \mathtt{t'}_2\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{t'}_n, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}}, \vec{\sigma})$

$$\begin{aligned} \equiv \quad & \textit{let}\ M = \{M_i | 1 \leq i \leq n \wedge M_i = \mathbf{matchset}(\mathtt{t}'_i, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}})\} \\ & \textit{in}\ \ \textit{if}\ M_i = \{\}\ \textit{for any}\ 1 \leq i \leq n\ \textit{then} <> \\ & \qquad \textit{else}\ \ \textit{let}\ \Phi = \{\{m_1, m_2, \ldots, m_n\} | m_i \in M_i\} \\ & \qquad \textit{in}\ \mathbf{merge}(\vec{\sigma}, \bigcup_{\phi \in \Phi} \mathbf{merge}/(\phi \downarrow * \downarrow 2)) \end{aligned}$$

includes* : $\mathsf{Type^*} \times \mathsf{Type^*} \times \mathsf{TypeVar^*} \times \mathsf{Type^*} \rightarrow \mathsf{TypeVarMap^*}$

$\mathbf{includes^*}(<>, <>, \vec{\mathtt{p}}, \vec{\mathtt{u}})$

$$\equiv \quad < \{\} >$$

$\mathbf{includes^*}(\mathtt{s} \cdot \vec{\mathtt{s}}, \mathtt{t} \cdot \vec{\mathtt{t}}, \vec{\mathtt{p}}, \vec{\mathtt{u}})$

$$\equiv \quad \mathbf{merge}(\mathbf{includes}(\mathtt{s}, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}}), \mathbf{includes^*}(\vec{\mathtt{s}}, \vec{\mathtt{t}}, \vec{\mathtt{p}}, \vec{\mathtt{u}}))$$

any TVM. An algorithm for the **includes** function is defined, by cases, in Definition 2 (explanation follows).[2]

Definition 2 defines a first order unification algorithm based on inclusion rather than on equality. The main difference between it and Definition 1 is the treatment of union types, which we will explain in detail. Since we have defined $\sqsubseteq$: $\mathsf{Type} \times \mathsf{Type} \rightarrow \mathsf{Boolean}$, if $\mathtt{t}$ and $\mathtt{t}'$ are types then $\mathtt{t}' \sqsubseteq \mathtt{t}$ is a predicate equivalent to $\mathbf{includes}(\mathtt{t}, \mathtt{t}', <>, <>) = < \{\} >$. If the arguments to **includes** are an object type and a block type, the result is $<>$. If one of the arguments to **includes** is a union type and the other is not, the other argument is implicitly converted so that both are union types. In examples using functions that take a range type list argument, we omit this argument if all range types are *Anything* or if the list is empty. Some simple examples of **includes** without union types are:

- $\mathbf{includes}(\mathit{Character}, \mathit{Character}, < P, Q, R >) = < \{\} >$
- $\mathbf{includes}(\mathit{Integer}, \mathit{Character}, < P, Q, R >) = <>$
- $\mathbf{includes}(P, \mathit{Character}, < P, Q, R >) = < \{P \mapsto \mathit{Character}\} >$
- $\mathbf{includes}(\mathit{Array\ of{:}\ Integer}, \mathit{Array\ of{:}}\ P, < P, Q, R >) = < \{P \mapsto \mathit{Integer}\} >$
- $\mathbf{includes}(\mathit{Array\ of{:}}\ P, \mathit{Block\ returns{:}}\ Q, < P, Q, R >) = <>$
- $\mathbf{includes}(\mathit{Block\ of{:}}\ R\ \mathit{returns{:}}\ (\mathit{Array\ of{:}}\ R), \mathit{Block\ of{:}\ Integer\ returns{:}}\ P, < P, Q, R >)$ $= < \{R \mapsto \mathit{Integer}, P \mapsto \mathit{Array\ of{:}}\ R\} >$.

When the arguments to the $\mathbf{includes}(\mathtt{t}, \mathtt{t}', \vec{\mathtt{p}}, \vec{\mathtt{u}})$ function are union types, it tries to eliminate each element of $\mathtt{t}'$ by finding an element of $\mathtt{t}$ that includes it. If each element of $\mathtt{t}'$ can be eliminated then $\mathtt{t}$ includes $\mathtt{t}'$, otherwise it does not. This is complicated slightly by the fact that inclusion is defined relative to some TVM. Therefore, it may be possible to pair each element of $\mathtt{t}'$ with more than one element of $\mathtt{t}$, under different TVMs. Thus, for $\mathtt{t}$ to include $\mathtt{t}'$ there must exist at least one TVM σ for which $\mathbf{evaluate}(\mathtt{t}', \sigma) \sqsubseteq \mathbf{evaluate}(\mathtt{t}, \sigma)$.

For union types, the **includes** function is defined in terms of the helping function **match-merge** and the mutually recursive function **matchset**. Given a type $\mathtt{t}'_i \in \mathtt{t}' = \mathtt{t'}_1 + \mathtt{t'}_2 + \ldots + \mathtt{t'}_m$, a union type $\mathtt{t} = \{t_1, t_2, \ldots, t_n\}$, a list of type variables $\vec{\mathtt{p}}$, and a list of corresponding range types $\vec{\mathtt{u}}$, $\mathbf{matchset}(\mathtt{t}'_i, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}})$ returns the set of all possible pairs $< \mathtt{t}_j, \sigma >$, where $\mathtt{t}_j$ is an element of $\mathtt{t}$ and σ is a TVM such that $\mathbf{evaluate}(\mathtt{t}'_i, \sigma) \sqsubseteq \mathbf{evaluate}(\mathtt{t}_j, \sigma)$, i.e.

$$\mathbf{matchset}(\mathtt{t}'_i, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}}) \equiv \{< \mathtt{t}_j, \sigma > | \mathtt{t}_j \in \mathtt{t} \wedge \sigma \in \mathbf{includes}(\mathtt{t}_j, \mathtt{t}'_i, \vec{\mathtt{p}}, \vec{\mathtt{u}})\}.$$

For example (recall that range type list arguments are not shown if all range types are *Anything*),

[2] The binary operator "$\downarrow$" is the accessing function for lists and n-tuples. It is occasionally used when subscripting is inadequate or ambiguous. The notation $\mathbf{merge}(\phi \downarrow * \downarrow 2)$ is equivalent to

$\vec{\sigma}' \leftarrow < \{\} >$
for each $m_i \in \phi$ do
  $\vec{\sigma}' \leftarrow \mathbf{merge}(\vec{\sigma}', < m_i \downarrow 2 >)$
return($\vec{\sigma}'$).

$\textbf{matchset}(\textit{Float}, \textit{Character}, < P,\ Q,\ R >)$
$= \{\}$

$\textbf{matchset}(\textit{Integer}, \textit{Integer} + \textit{Character}, < P,\ Q,\ R >)$
$= \{< \textit{Integer}, \{\} >\}$

$\textbf{matchset}(\textit{Set of: } P, (\textit{Set of: Integer}) + (\textit{Set of: Float}) + \textit{UndefinedObject}, < P,\ Q,\ R >)$
$= \{< \textit{Set of: Integer}, \{P \mapsto \textit{Integer}\} >, < \textit{Set of: Float}, \{P \mapsto \textit{Float}\} >\}$

A type variable $\mathtt{p} \in \vec{\mathtt{p}}$ that appears as an element of either $\mathtt{t}$ or $\mathtt{t}'$ requires special treatment since it could, at least potentially, be bound to any subset of $\mathtt{t} \cup \mathtt{t}'$ (including itself, if it appears in both $\mathtt{t}$ and $\mathtt{t}'$). For example,

$\textbf{includes}(\textit{Integer} + P, \textit{Float}, < P >) = < \{P \mapsto \textit{Float}\}, \{P \mapsto \textit{Integer} + \textit{Float}\} >$
$\textbf{includes}(\textit{Float} + P, \textit{Float} + P, < P >) = < \{P \mapsto \textit{Float}\}, \{\} >$

The definition of **includes** for union types has two main parts, one to handle these "free standing" type variables, and one to handle the case where there are none.

Referring back to the case for union types in Definition 2, $\vec{\mathtt{p}}'$ is the list of "free standing" type variables in $\mathtt{t}$ and $\mathtt{t}'$. The **get-ranges** function is defined to return the list of corresponding range types $\vec{\mathtt{u}}'$ for $\vec{\mathtt{p}}'$ based on the correspondence between $\vec{\mathtt{p}}$ and $\vec{\mathtt{u}}$. The TVM list $\vec{\sigma}$ consists of all possible TVMs in which each type variable $\mathtt{p}'_i \in \vec{\mathtt{p}}$ is bound either to a subset (subtype) of $\mathbf{P}((\mathtt{t} \cup \mathtt{t}') - \mathtt{p}'_i)$ (the empty set excluded), or, if $\mathtt{p}'_i$ is an element of both $\mathtt{t}$ and $\mathtt{t}'$, to $\perp$ (i.e. it is not bound so that $\textbf{evaluate}(\mathtt{p}, \sigma) = \mathtt{p}$). If there are "free standing" type variables in both $\mathtt{t}$ and $\mathtt{t}'$, some of the TVMs in $\vec{\sigma}$ will contain "cycles" (e.g. $\{P \mapsto Q, Q \mapsto P\}$) which will, in turn, cause an infinite loop in the **evaluate** function. One way to handle this problem is to collapse each cycle onto one of its type variables and use this "representative" as the result of the evaluation.

If $\vec{\mathtt{p}}'$ is empty (i.e. there are no "free standing" type variables in $\mathtt{t}$ or $\mathtt{t}'$) then the algorithm uses **match-merge**, which tries to eliminate each element of $\mathtt{t}'$ by finding a corresponding element of $\mathtt{t}$ that includes it. For each $\mathtt{t}'_i \in \mathtt{t}'$, $M_i = \textbf{matchset}(\mathtt{t}'_i, \mathtt{t}, \vec{\mathtt{p}}, \vec{\mathtt{u}})$ is the set of elements of $\mathtt{t}$ that can be used to eliminate $\mathtt{t}'_i$. If the result of **matchset** for any $\mathtt{t}'_i$ is the empty set then not all $\mathtt{t}'_i$'s could be eliminated and the inclusion check fails, returning $<>$.

Even if each $\mathtt{t}'_i$ has at least one match, each match is obtained relative to a "local" TVM, isolated from the effects of the other matches. Each of these local TVMs must be merged together to form one TVM that applies to the entire union type $\mathtt{t}'$. If each $\mathtt{t}'_i$ has only one match it is straightforward to merge all the σ's together (recall that the elements of a matchset are of the form $< \mathtt{t}_i, \sigma >$). However, it is possible for matchsets to contain more than one element. The σ-merging must be carried out for every possible match vector $\{m_1, m_2, \ldots, m_n\}$, representing the type $\mathtt{t}'$ in which each element $\mathtt{t}'_i$ has been replaced by an element of its corresponding matchset M_i. Thus, the result of **includes** is the TVM list representing the union of all these merge results. (In this case, the final $\textbf{merge}(\vec{\sigma}, \ldots)$ in **match-merge** has no effect on its second argument since $\vec{\sigma} =< \{\} >$ is the identity TVM list for **merge**.)

If $\vec{\mathtt{p}}'$ is nonempty then, for a given $\sigma \in \vec{\sigma}$, all type variables bound by σ are removed from $\mathtt{t}$ and $\mathtt{t}'$ using **evaluate**. This has the effect of factoring out all "free standing" type variables relative to σ. Once this factoring is done, the algorithm proceeds as if $\vec{\mathtt{p}}'$ were empty. Each of the resulting TVMs must be merged with σ (the final $\textbf{merge}(\vec{\sigma}, \ldots)$ in **match-merge**) to check for conflicting bindings. This must be done for each σ, with all the resulting TVMs being collected into one big TVM list.

The following example contrasts the solutions given by **includes** for the two inclusion equations $P \sqsubseteq \mathit{Character} + \mathit{Integer}$ and $\mathit{Character} + \mathit{Integer} \sqsubseteq P$. Although the two equations are to be solved independently, it is instructive to consider their solutions in parallel. Key stages in the algorithm are traced in Figure 5.2.

Figure 5.2(a) shows the initial calls to **includes**. Since the calls contain mixed arguments, the non-union type arguments are implicitly converted to singleton union types. Following Definition 2, the set of "free standing" type variables $\vec{\mathtt{p}}'$ is computed for each call, as shown in Figure 5.2(b). The TVM lists $\vec{\sigma}$ of potential bindings for P are computed. The results, which happen to be the same for both calls, are shown in Figure 5.2(c). Since $\vec{\mathtt{p}}$ is not empty, each element of $\vec{\sigma}$ must be considered separately.

For $\sigma = \{P \mapsto \mathit{Character}\}$, the "free standing" type variable P is removed from the argument types using **evaluate**, i.e. **evaluate**$(P, \sigma) = \mathit{Character}$ and **evaluate**$(\mathit{Character} + \mathit{Integer}, \sigma) = \mathit{Character} + \mathit{Integer}$. With occurrences of P removed from the types, the M sets are easily computed, as shown in Figure 5.2(d). For the call on the right, M contains an empty set element so the resulting TVM list for this σ is empty (boxed). For the call on the left, the result for this σ is computed by merging $< \sigma >$ together with the TVMs from each element of M. M has only one element (with a trivial TVM) so the merge is easy (again, the result is boxed).

The case for $\sigma = \{P \mapsto \mathit{Integer}\}$, shown in Figure 5.2(e), is almost identical to the previous case for both calls. For the last case, $\sigma = \{P \mapsto \mathit{Character} + \mathit{Integer}\}$, each M set contains two elements. This means that three TVM lists must be merged together to find the result for this σ. As before, the TVMs for the elements of M are trivial so the merge is simple. This time the results for the calls are the same, as shown in Figure 5.2(f).

The final result for each of the calls is obtained by taking the union of the intermediate (boxed) results for each σ. The final results are shown in Figure 5.2(g). Notice that they are not the same.

5.3 Type-Inference Algorithm

This section gives a detailed description of the type-inference algorithm. We use Kamin's notation to aid in comparison. The abstract syntax is presented first, followed by a "bottom up" description of the algorithm.

5.3.1 Abstract Syntax

An abstract syntax of TS is given in Figure 5.3. We have modified Kamin's abstract syntax for Smalltalk to include type declarations, to use more traditional BNF-style productions, and to present a more unified and precise view of Smalltalk.[3] This syntax differs from that presented in Chapter 2. Type declarations are closely integrated with the method definition, as in [Gra87], rather than appearing as a prefix. This simplifies the semantics by making it easier to associate a variable declaration with its corresponding type declaration. The letters in typewriter font (at the far left) represent elements of the indicated syntactic domains. They are used later in the semantic definitions. For sake of readability, we use domain names rather than symbols in the grammar productions. Other syntactic tokens are underlined.

[3]In particular, we have changed the domain definitions of and from functions (not really abstract syntax) to pair lists, removed the distinction between sending messages to classes versus other objects (Smalltalk makes no such distinction and Kamin's semantics should be changed to reflect this—but that is not our purpose), and cleaned up the syntax for literals and variables.

$\textbf{includes}(Char + Int, P, < P >)$

$\textbf{includes}(P, Char + Int, < P >)$

(a)

$\vec{\mathrm{p}}' =< P >$

same

(b)

$\vec{\sigma} =< \{P \mapsto Char\}, \{P \mapsto Int\}, \{P \mapsto Char + Int\} >$

same

(c)

$\sigma = \{P \mapsto Char\}$:

$$
\begin{aligned}
&\textbf{evaluate}(P, \sigma) = Char \\
&\textbf{evaluate}(Char + Int, \sigma) = Char + Int \\
&M = \{\textbf{matchset}(Char, Char + Int, <>)\} \\
&\quad = \{\{< Char, \{\} >\}\} \\
&\textbf{merge}(< \{P \mapsto Char\} >, < \{\} >) \\
&\boxed{< \{P \mapsto Char\} >}
\end{aligned}
$$

$\sigma = \{P \mapsto Char\}$:

$$
\begin{aligned}
&\text{same} \\
&\text{same} \\
&M = \{\textbf{matchset}(Char, Char, <>), \\
&\quad \textbf{matchset}(Int, Char, <>)\} \\
&\quad = \{\{< Char, \{\} >\}, \{\}\} \\
&\boxed{<>}
\end{aligned}
$$

(d)

$\sigma = \{P \mapsto Int\}$:

$$
\begin{aligned}
&\textbf{evaluate}(P, \sigma) = Int \\
&\textbf{evaluate}(Char + Int, \sigma) = Char + Int \\
&M = \{\textbf{matchset}(Int, Char + Int, <>)\} \\
&\quad = \{\{< Int, \{\} >\}\} \\
&\textbf{merge}(< \{P \mapsto Int\} >, < \{\} >) \\
&\boxed{< \{P \mapsto Int\} >}
\end{aligned}
$$

$\sigma = \{P \mapsto Int\}$:

$$
\begin{aligned}
&\text{same} \\
&\text{same} \\
&M = \{\textbf{matchset}(Char, Int, <>), \\
&\quad \textbf{matchset}(Int, Int, <>)\} \\
&\quad = \{\{\}, \{< Int, \{\} >\}\} \\
&\boxed{<>}
\end{aligned}
$$

(e)

$\sigma = \{P \mapsto Char + Int\}$:

$$
\begin{aligned}
&\textbf{evaluate}(P, \sigma) = Int \\
&\textbf{evaluate}(Char + Int, \sigma) = Char + Int \\
&M = \{\textbf{matchset}(Char, Char + Int, <>), \\
&\quad \textbf{matchset}(Int, Char + Int, <>)\} \\
&\quad = \{\{< Char, \{\} >\}, \{< Int, \{\} >\}\} \\
&\textbf{merge}(< \{P \mapsto Char + Int\} >, < \{\} >, < \{\} >) \\
&\boxed{< \{P \mapsto Char + Int\} >}
\end{aligned}
$$

$\sigma = \{P \mapsto Char + Int\}$:

$$
\begin{aligned}
&\text{same} \\
&\text{same} \\
&\text{same} \\
&\quad \text{same} \\
&\text{same} \\
&\boxed{< \{P \mapsto Char + Int\} >}
\end{aligned}
$$

(f)

$$\bigcup_{\sigma \in \vec{\sigma}} =< \{P \mapsto Char\}, \{P \mapsto Int\}, \{P \mapsto Char + Int\} >$$

$$\bigcup_{\sigma \in \vec{\sigma}} =< \{P \mapsto Char + Int\} >$$

(g)

Figure 5.2: Intermediate results in computing $P \sqsubseteq Char + Int$ and $Char + Int \sqsubseteq P$.

$$
\begin{array}{rcll}
\mathtt{m} & \in & \mathsf{MsgName} & \\
\mathtt{C} & \in & \mathsf{ClassName} & \\
\mathtt{V} & \in & \mathsf{VarName} & \\
\mathtt{i} & \in & \mathsf{IntConst} & \\
\mathtt{w} & \in & \mathsf{ClassVar} & ::= \ \mathsf{ClassName}\ \mathsf{VarName} \\
\mathtt{x} & \in & \mathsf{InstVar} & ::= \ \mathsf{VarName} \\
\mathtt{y} & \in & \mathsf{ArgVar} & ::= \ \mathsf{VarName} \\
\mathtt{z} & \in & \mathsf{LocalVar} & ::= \ \mathsf{VarName} \\
\mathtt{p,\ q,\ r} & \in & \mathsf{TypeVar} & ::= \ \mathsf{VarName} \\
\mathtt{v} & \in & \mathsf{Var} & ::= \ \mathsf{ClassVar} \mid \mathsf{InstVar} \mid \mathsf{ArgVar} \mid \mathsf{LocalVar} \\
\mathtt{s,\ t,\ u} & \in & \mathsf{Type} & ::= \ \mathsf{ClassName}\ \underline{(}\ \mathsf{Type}^*\ \underline{)} \\
 & & & \quad\ \mid \mathsf{Type}^*\ \underline{\rightarrow}\ \mathsf{Type} \\
 & & & \quad\ \mid \mathsf{TypeVar} \\
 & & & \quad\ \mid \mathsf{Type}\ \underline{+}\ \mathsf{Type} \\
 & & & \quad\ \mid \underline{\mathsf{Self}} \\
 & & & \quad\ \mid \underline{\mathsf{Anything}} \\
 & & & \quad\ \mid \underline{\mathsf{Nothing}} \\
\mathtt{P} & \in & \mathsf{Prg} & ::= \ \underline{\mathsf{evaluate}}\ \mathsf{LocalVar}^*\ \mathsf{Type}^*\ \mathsf{Expr}\ \underline{\mathsf{in}}\ \mathsf{Hier} \\
\mathtt{H} & \in & \mathsf{Hier} & ::= \ (\mathsf{ClassName}\ \mathsf{ClassDef})^* \\
\mathtt{D} & \in & \mathsf{ClassDef} & ::= \ \mathsf{ClassName}\ \mathsf{ClassName}\ \mathsf{ClassVar}^*\ \mathsf{Type}^*\ \mathsf{InstVar}^*\ \mathsf{Type}^* \\
 & & & \qquad\qquad \mathsf{TypeVar}^*\ \mathsf{Type}^*\ \mathsf{Methods} \\
\mathtt{F} & \in & \mathsf{Methods} & ::= \ (\mathsf{MsgName}\ (\mathsf{Method} \mid \underline{\mathsf{no\text{-}def}}))^* \\
\mathtt{M} & \in & \mathsf{Method} & ::= \ \mathsf{TypeVar}^*\ \mathsf{Type}^*\ \mathsf{Type}\ \mathsf{MsgName}\ \mathsf{ArgVar}^*\ \mathsf{Type}^*\ \underline{\uparrow}\mathsf{Type} \\
 & & & \qquad\qquad \mathsf{LocalVar}^*\ \mathsf{Type}^*\ \mathsf{Expr} \\
 & & & \quad\ \mid \mathsf{TypeVar}^*\ \mathsf{Type}^*\ \mathsf{Type}\ \mathsf{MsgName}\ \mathsf{ArgVar}^*\ \mathsf{Type}^*\ \underline{\uparrow}\mathsf{Type} \\
 & & & \qquad\qquad \mathsf{LocalVar}^*\ \mathsf{Type}^*\ \underline{<}\ \underline{\mathsf{primitive}}\ \mathsf{IntConst}\ \underline{\uparrow}\mathsf{Type}\ \underline{>}\ \mathsf{Expr} \\
\mathtt{e} & \in & \mathsf{Expr} & ::= \ \mathsf{VarName}\ \underline{\leftarrow}\ \mathsf{Expr} \\
 & & & \quad\ \mid \mathsf{Expr}\ \mathsf{MsgName}\ \mathsf{Expr}^* \\
 & & & \quad\ \mid \mathsf{Expr}\ \mathsf{ClassName}\ \underline{::}\ \mathsf{MsgName}\ \mathsf{Expr}^* \\
 & & & \quad\ \mid \underline{\uparrow}\ \mathsf{Expr} \\
 & & & \quad\ \mid \underline{[}\ \mathsf{TypeVar}^*\ \mathsf{Type}^*\ \mathsf{ArgVar}^*\ \mathsf{Type}^*\ \underline{\uparrow}\mathsf{Type}\ \underline{|}\ \mathsf{Expr}\ \underline{]} \\
 & & & \quad\ \mid \mathsf{Expr}^* \\
 & & & \quad\ \mid \mathsf{ObjExpr} \\
\mathtt{O} & \in & \mathsf{ObjExpr} & ::= \ \underline{(}\ \mathsf{Literal}\ \mathsf{Type}\ \underline{)} \mid \mathsf{Var} \mid \underline{\mathsf{self}} \mid \underline{\mathsf{nil}} \mid \underline{\mathsf{true}} \mid \underline{\mathsf{false}} \\
\mathtt{l} & \in & \mathsf{Literal} &
\end{array}
$$

Figure 5.3: TS abstract syntax.

$$
\begin{array}{lllll}
\tau & \in & \mathsf{TypeTree} & ::= & \mathsf{Type} \\
 & & & & \mid \mathsf{TypeTree} \underline{\leftarrow} \mathsf{TypeTree} \\
 & & & & \mid \mathsf{TypeTree}\ \mathsf{MsgName}\ \mathsf{TypeTree}^* \\
 & & & & \mid \mathsf{TypeTree}\ \mathsf{ClassName}\ \underline{::}\ \mathsf{MsgName}\ \mathsf{TypeTree}^* \\
 & & & & \mid \underline{\uparrow}\ \mathsf{TypeTree} \\
 & & & & \mid \mathsf{TypeTree}^* \\
 & & & & \mid \underline{[}\ \mathsf{ArgVar}^*\ \mathsf{Type}^*\ \underline{\uparrow}\mathsf{Type}\ \underline{|}\ \mathsf{TypeTree}\ \underline{]}
\end{array}
$$

Figure 5.4: Abstract syntax of type trees.

Type declarations are required for all instance variables, class variables, literals, and primitives. Type declarations are "optional" for method arguments and return values, block arguments and return values, and all temporary variables. These declarations are optional in the sense that, while the abstract syntax requires a type declarations for every variable, method, and block, the declaration may be a type variable. Type-inference involves computing values for those type variables. The types of the pseudo-variables true, false, and nil are predefined constants. The type of self is determined by the receiver type of the method being type-checked. A list of variable declarations is followed by a list of corresponding type declarations as shown in the productions for Prg, ClassDef, and Method. The return types of methods, blocks, and primitives are indicated by a preceding "↑." The domains MsgName, ClassName, VarName, and IntConst are lexical. We also assume the implicit grammar rules

$$
\begin{array}{lll}
\mathsf{d} \in \mathsf{Domain} & & \\
\vec{\mathsf{d}} \in \mathsf{DomList} & ::= & \mathsf{Domain}^*
\end{array}
$$

for lists of syntactic domain elements.

5.3.2 Auxiliary Domains and Definitions

The abstract syntax of type trees is shown in Figure 5.4. A type tree is an expression tree whose leaves are types.

We need a few more semantic domains, in addition to TVMs and inclusion equations, to describe the semantics of the type-inference algorithm.

$$
\begin{array}{lllll}
 & & \mathsf{MethodType} & = & \mathsf{BlockType} \times \mathsf{TypeEnv} \times \mathsf{TypeTree} \\
\delta & \in & \mathsf{MethodEnv} & = & \mathsf{ClassName} \times \mathsf{MsgName} \to \mathsf{Method} \\
\eta & \in & \mathsf{MethodTypeEnv} & = & \mathsf{ClassName} \times \mathsf{MsgName} \to \mathsf{MethodType} \\
\mu & \in & \mathsf{TypeEnv} & = & (\mathsf{VarName} \to \mathsf{Type}) \times \mathsf{MethodTypeEnv} \times \mathsf{IncEqn}^* \times \mathsf{Hier}
\end{array}
$$

A method type consists of a BlockType to represent the argument and return types, a TypeEnv to hold miscellaneous type information, and a TypeTree to hold any type constraints. A method environment maps $< \mathtt{C}, \mathtt{m} >$ to method definitions. A method type environment is a function from $< \mathtt{C}, \mathtt{m} >$ pairs to method types. A type environment consists of a function mapping variable names to types, a method type environment, a set of inclusion equations, and a class hierarchy.

The following functions were defined in Section 5.2.

$$\begin{array}{l}\textbf{evaluate} : \mathsf{Type} \times \mathsf{TypeVarMap} \rightarrow \mathsf{Type} \\ \textbf{evaluate*} : \mathsf{Type*} \times \mathsf{TypeVarMap} \rightarrow \mathsf{Type*} \\ \textbf{includes} : \mathsf{Type} \times \mathsf{Type} \times \mathsf{TypeVar*} \times \mathsf{Type*} \rightarrow \mathsf{TypeVarMap*}\end{array}$$

In addition, we define the following basic look-up functions.

$$\begin{array}{l}\textbf{find-class} : \mathsf{ClassName} \times \mathsf{Hier} \rightarrow \mathsf{ClassDef} \\ \textbf{find-class}(\mathtt{C}, <>) \equiv \bot \\ \textbf{find-class}(\mathtt{C}, <\mathtt{C}', \mathtt{D}> \cdot \mathtt{H}) \equiv \text{if } \mathtt{C} = \mathtt{C}' \text{ then } \mathtt{D} \text{ else } \textbf{find-class}(\mathtt{C}, \mathtt{H})\end{array}$$

$$\begin{array}{l}\textbf{find-method} : \mathsf{MsgName} \times \mathsf{ClassName} \times \mathsf{Hier} \rightarrow \mathsf{Method} \\ \textbf{find-method}(\mathtt{m}, \mathtt{C}, \mathtt{H}) \equiv \text{let } \mathtt{C}\ \mathtt{S}\ \vec{\mathtt{w}}\ \vec{\mathtt{t}}\ \vec{\mathtt{x}}\ \vec{\mathtt{t}}'\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \mathtt{F} = \textbf{find-class}(\mathtt{C},\ \mathtt{H}) \\ \qquad\qquad\qquad\qquad\quad \text{in } \textbf{get-method}(\mathtt{m}, \mathtt{F})\end{array}$$

$$\begin{array}{l}\textbf{get-method} : \mathsf{MsgName} \times \mathsf{Methods} \rightarrow (\mathsf{Method} + \{\underline{\mathsf{no\text{-}def}}\}) \\ \textbf{get-method}(\mathtt{m}, <>) \equiv \mathsf{no\text{-}def} \\ \textbf{get-method}(\mathtt{m}, <\mathtt{m}', \mathtt{M}> \cdot \mathtt{F}) \equiv \text{if } \mathtt{m} = \mathtt{m}' \text{ then } \mathtt{M} \text{ else } \textbf{get-method}(\mathtt{m}, \mathtt{F})\end{array}$$

Find-class and **find-method** retrieve class and method definitions from the class hierarchy, respectively. **Get-method** retrieves a method definition from a list of method definitions.

The following functions are used to test the structure of type trees.

$$\begin{array}{l}\textbf{isType} : \mathsf{TypeTree} \rightarrow \mathsf{Boolean} \\ \textbf{isType}(\mathtt{t}) \equiv \text{true} \\ \textbf{isType}(\,[\vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \vec{\mathtt{y}}\ \vec{\mathtt{t}} \uparrow \mathtt{u} \mid \tau]\,) \equiv \text{true} \\ \textbf{isType}(\tau) \equiv \text{false}\end{array}$$

$$\begin{array}{l}\textbf{isTypeList} : \mathsf{TypeTree*} \rightarrow \mathsf{Boolean} \\ \textbf{isTypeList}(<>) \equiv \text{true} \\ \textbf{isTypeList}(\tau \cdot \vec{\tau}) \equiv \textbf{isType}(\tau) \wedge \textbf{isTypeList}(\vec{\tau})\end{array}$$

The following functions are used for updating type environments.

$$\begin{array}{l}\textbf{asgn-types} : \mathsf{Var*} \times \mathsf{Type*} \rightarrow \mathsf{TypeEnv} \rightarrow \mathsf{TypeEnv} \\ \textbf{asgn-types}(\vec{\mathtt{v}}, \vec{\mathtt{t}}) \equiv \lambda\mu.\mu[1 \rightarrow \vec{\mathtt{v}} \rightarrow \vec{\mathtt{t}}]^4\end{array}$$

$$\begin{array}{l}\textbf{asgn-class-types} : \mathsf{ClassName} \times \mathsf{Hier} \rightarrow \mathsf{TypeEnv} \rightarrow \mathsf{TypeEnv} \\ \textbf{asgn-class-types}(\mathtt{C},\ \mathtt{H}) \equiv \text{let } \mathtt{C}\ \mathtt{S}\ \vec{\mathtt{w}}\ \vec{\mathtt{t}}\ \vec{\mathtt{x}}\ \vec{\mathtt{t}}'\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \mathtt{F} = \textbf{find-class}(\mathtt{C},\ \mathtt{H}) \\ \qquad\qquad\qquad\qquad\qquad\quad \text{in } \lambda\mu.\ \textbf{asgn-types}(\vec{\mathtt{x}} \cdot \vec{\mathtt{w}} \cdot \vec{\mathtt{p}}, \vec{\mathtt{t}}' \cdot \vec{\mathtt{t}} \cdot \vec{\mathtt{u}})\mu\end{array}$$

$$\begin{array}{l}\textbf{merge-env} : \mathsf{TypeEnv} \times \mathsf{TypeEnv} \rightarrow \mathsf{TypeEnv} \\ \textbf{merge-env}(\mu, \mu') \equiv <\mu_1 \cup \mu'_1, \mu_2, \mu_3 \cup \mu'_3, \mu_4>\end{array}$$

$$\textbf{new-typevar} : \bot \rightarrow \mathsf{TypeVar}$$

[4]If f is an ordered pair whose second element is a function g then $f[2 \rightarrow a \rightarrow b]$ is the same as f except that the second element is now $\lambda(x)$. if $x = a$ then b else $g(x)$; $f[2 \rightarrow \vec{a} \rightarrow \vec{b}]$ represents a list of simultaneous function updates for g. This notation is taken from [Kam88].

Asgn-types adds the pairwise bindings ($\mathsf{Var} \rightarrow \mathsf{Type}$), indicated by its list arguments, to a type environment. **Asgn-class-types** adds bindings for instance variables, class variables, and class type parameters to a type environment. This environment is used to provide a context for type-checking a method in a particular class. **Merge-new** combines two type environments (the second $\mathsf{MethodTypeEnv}$ and fourth **Hier** components of any two type environments should be the same). **New-typevar** returns a fresh type variable.

These functions are used for solving sets of inclusion equations.

$$\textbf{solve-inclusions} : \mathsf{IncEqn*} \rightarrow \mathsf{TypeVarMap*}$$

$$\begin{aligned}
&\textbf{add-inclusion} : \mathsf{Type} \times \mathsf{Type} \rightarrow \mathsf{TypeEnv} \rightarrow \mathsf{TypeEnv}\\
&\textbf{add-inclusion}(\mathtt{t}, \mathtt{t}')\mu\\
&\qquad\qquad \equiv \text{let } \vec{\nu} = \mu_3 \cdot < \mathtt{t}, \mathtt{t}' >\\
&\qquad\qquad\quad\; \text{and } \vec{\sigma} = \textbf{solve-inclusions}(\vec{\nu})\\
&\qquad\qquad\quad\; \text{in if } \vec{\sigma} = <> \text{ then } \textit{wrong}\\
&\qquad\qquad\qquad\; \text{else } < \mu_1, \mu_2, \vec{\nu}, \mu_4 >
\end{aligned}$$

$$\begin{aligned}
&\textbf{add-inclusion*} : \mathsf{Type*} \times \mathsf{Type*} \rightarrow \mathsf{TypeEnv} \rightarrow \mathsf{TypeEnv}\\
&\textbf{add-inclusion*}(<>, <>)\mu \equiv \mu\\
&\textbf{add-inclusion*}(\mathtt{t} \cdot \vec{\mathtt{t}}, \mathtt{t}' \cdot \vec{\mathtt{t}}')\mu\\
&\qquad\qquad \equiv \textbf{add-inclusion}(\mathtt{t}, \mathtt{t}')(\textbf{add-inclusion*}(\vec{\mathtt{t}}, \vec{\mathtt{t}}')\mu)
\end{aligned}$$

$$\begin{aligned}
&\textbf{evaluate-in} : \mathsf{Type} \times \mathsf{TypeEnv} \rightarrow \mathsf{Type}\\
&\textbf{evaluate-in}(\mathtt{t}, \mu)\\
&\qquad\qquad \equiv \text{let } \vec{\sigma} = \textbf{solve-inclusions}(\mu_3)\\
&\qquad\qquad\quad\; \text{in } \textbf{evaluate}(\mathtt{t}, \vec{\sigma} \downarrow 1)
\end{aligned}$$

Solve-inclusions is based on Definition 2 and returns the set of all solutions for a set of inclusion equations. Its definition is not shown. **Add-inclusion** adds an inclusion equation to a set of inclusion equations, checks the consistency of the new set, and returns a type environment with the new set of inclusion equations. **Add-inclusion*** adds a list of inclusion equations to a type environment. **Evaluate-in** returns the evaluation of a type based on the set of inclusion equations in a type environment. We well later argue the validity of arbitrarily choosing the first TVM from a set of solutions.

5.3.3 Type-Inference Functions

The syntactic transformation function **T** converts Smalltalk expressions to type trees. The essence of this is to convert all leaf expressions to types; the structure of internal expressions remains intact. The types of variables and pseudo-variables are taken from the environment or are constants. For simplicity, we assume that the type of each literal is explicitly declared (precise typing for literals is discussed in Section 4.4. Blocks are passed unchanged and converted later. Other expressions simply convert their subexpressions.

$$\mathbf{T} : \mathsf{Expr} \rightarrow \mathsf{TypeEnv} \rightarrow \mathsf{TypeTree}$$

$$\begin{array}{l}
\mathbf{T}[\![\mathtt{v}]\!]\mu \equiv \mu_1[\![\mathtt{v}]\!] \\
\mathbf{T}[\![\underline{\mathsf{self}}]\!]\mu \equiv \mu_1[\![\underline{\mathsf{self}}]\!] \\
\mathbf{T}[\![\underline{\mathsf{true}}]\!]\mu \equiv \underline{\mathsf{True}} \\
\mathbf{T}[\![\underline{\mathsf{false}}]\!]\mu \equiv \underline{\mathsf{False}} \\
\mathbf{T}[\![\underline{\mathsf{nil}}]\!]\mu \equiv \underline{\mathsf{UndefinedObject}} \\
\mathbf{T}[\![\mathtt{(l\ t)}]\!]\mu \equiv \mathtt{t}
\end{array}$$

$$\begin{array}{l}
\mathbf{T}[\![\mathtt{[}\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \vec{\mathtt{y}}\ \vec{\mathtt{t}}\ \uparrow\!\mathtt{u}\ \mid\ \mathtt{e]}]\!]\mu \equiv \mathtt{[}\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \vec{\mathtt{y}}\ \vec{\mathtt{t}}\ \uparrow\!\mathtt{u} \mid \mathtt{e}\ \mathtt{]} \\
\mathbf{T}[\![\mathtt{v}\ \leftarrow\ \mathtt{e}]\!]\mu \equiv \mathbf{T}[\![\mathtt{v}]\!]\mu \leftarrow \mathbf{T}[\![\mathtt{e}]\!]\mu \\
\mathbf{T}[\![\uparrow\!\mathtt{e}]\!]\mu \equiv \uparrow \mathbf{T}[\![\mathtt{e}]\!]\mu \\
\mathbf{T}[\![<>]\!]\mu \equiv <> \\
\mathbf{T}[\![\mathtt{e}\cdot\vec{\mathtt{e}}]\!]\mu \equiv \mathbf{T}[\![\mathtt{e}]\!]\mu \cdot \mathbf{T}[\![\vec{\mathtt{e}}]\!]\mu\ \mathbf{T}[\![\mathtt{e\ m}\ \vec{\mathtt{e}}]\!]\mu \equiv \mathbf{T}[\![\mathtt{e}]\!]\mu\ \mathtt{m}\ \mathbf{T}[\![\vec{\mathtt{e}}]\!]\mu \\
\mathbf{T}[\![\mathtt{e\ m}\ \vec{\mathtt{e}}]\!]\mu \equiv \mathbf{T}[\![\mathtt{e}]\!]\mu\ \mathtt{m}\ \mathbf{T}[\![\vec{\mathtt{e}}]\!]\mu \\
\mathbf{T}[\![\mathtt{e\ C::m}\ \vec{\mathtt{e}}]\!]\mu \equiv \mathbf{T}[\![\mathtt{e}]\!]\mu\ \mathtt{C::m}\ \mathbf{T}[\![\vec{\mathtt{e}}]\!]\mu
\end{array}$$

The type tree reduction function $\mathbf{R}$ (also a syntax transformation function) attempts to reduce (simplify) its type tree argument. There are usually two pre-conditions that must be satisfied before a type tree can be reduced. Each of its subtrees must have been reduced to a type (this is not always necessary if the subtree is a block) and it must satisfy any local type constraints. Satisfying local type constraints may change the type environment so $\mathbf{R}$ must return the reduced type tree and a new type environment. Type trees that cannot be reduced are left intact, perhaps with partially reduced subtrees. In the context of type-inference, *fail* is considered a type value. Unless otherwise noted, the type-inference functions are strict with regard to *fail*. The functions also fail if calls to **add-inclusion** return *wrong* or if $\mu_2 < \mathtt{C}, \mathtt{m} >$ returns <u>no-def</u>. Statements like $< \ldots >= \mathbf{R}[\![\ldots]\!]\mu$ and $\mu' = \textbf{add-inclusion}(\ldots)\mu$ imply success. The type of $\mathbf{R}$ is

$$\mathbf{R} : \mathsf{TypeTree} \rightarrow \mathsf{TypeEnv} \rightarrow (\mathsf{TypeTree} \times \mathsf{TypeEnv}).$$

A type always reduces to itself.

$$\mathbf{R}[\![\mathtt{t}]\!]\mu \equiv < \mathtt{t}, \mu >$$

An assignment $\tau \leftarrow \tau'$ can be reduced if τ can be reduced to a type $\tau\dagger$ (τ will always be a type since it corresponds to a variable), if τ' can be reduced to a type $\tau\ddagger$, and if adding the inclusion equation $\tau\ddagger \sqsubseteq \tau\dagger$ to the type environment produces a consistent type environment. If an assignment can be reduced, it reduces to $\tau\ddagger$.

$$\begin{array}{rl}
\mathbf{R}[\![\tau \leftarrow \tau']\!]\mu \equiv & \text{let } < \tau\dagger, \mu\dagger >= \mathbf{R}[\![\tau]\!]\mu \\
& \text{and } < \tau\ddagger, \mu\ddagger >= \mathbf{R}[\![\tau']\!]\mu\dagger \\
& \text{in if } \mathbf{isType}(\tau\ddagger) \text{ then let } \mu' = \textbf{add-inclusion}(\tau\dagger, \tau\ddagger)\mu\ddagger \text{ in } < \tau\ddagger, \mu' > \\
& \quad \text{else } < \tau\dagger \leftarrow \tau\ddagger, \mu\ddagger >
\end{array}$$

A return$\uparrow$ τ can be reduced if τ can be reduced to a type τ' and if the inclusion equation $\tau' \sqsubseteq \mu'_1[\![\underline{\mathsf{returnValue}}]\!]$ can be added consistently to the type environment. If a return can be reduced it reduces to *Nothing*.

$$\begin{aligned}\mathbf{R}[\![\uparrow\tau]\!]\mu \equiv\ & \text{let} < \tau', \mu' >= \mathbf{R}[\![\tau]\!]\mu \\ & \quad \text{in if } \mathbf{isType}(\tau') \\ & \quad\quad \text{then let } \mu'' = \mathbf{add\text{-}inclusion}(\mu'_1[\![\underline{\mathsf{returnValue}}]\!], \tau')\mu' \text{ in } < \underline{\mathsf{Nothing}}, \mu'' > \\ & \quad\quad \text{else } < \uparrow\tau', \mu' >\end{aligned}$$

To reduce a sequence of statements $\vec{\tau}$ to a type, each statement in the sequence must be reduced to a type. If a statement sequence can be reduced it is reduced to the type of the last statement in the sequence.

$$\begin{aligned}\mathbf{R}[\![\vec{\tau}]\!]\mu \equiv\ & \text{let} < \vec{\tau}', \mu' >= \mathbf{check\text{-}stmts}(\vec{\tau}, \mu) \\ & \text{in if } \mathbf{isTypeList}(\vec{\tau}') \text{ then } < \mathbf{last}(\vec{\tau}'), \mu' > \\ & \quad \text{else } < \vec{\tau}', \mu' >\end{aligned}$$

$$\begin{aligned}&\mathbf{check\text{-}stmts} : \mathsf{TypeTree^*} \times \mathsf{TypeEnv} \to (\mathsf{TypeTree^*} \times \mathsf{TypeEnv}) \\ &\mathbf{check\text{-}stmts}(<>, \mu) \equiv << >, \mu > \\ &\mathbf{check\text{-}stmts}(\tau \cdot \vec{\tau}, \mu) \\ &\quad \equiv \text{let} < \tau', \mu' >= \text{let } \vec{\mu} = \mathbf{expand\text{-}union}(\mu) \\ &\quad\quad\quad\quad\quad\quad \text{and } \vec{\tau}' = < \mathbf{R}[\![\tau]\!](\vec{\mu}\downarrow 1), \mathbf{R}[\![\tau]\!](\vec{\mu}\downarrow 2), \ldots, \mathbf{R}[\![\tau]\!](\vec{\mu}\downarrow n) > \\ &\quad\quad\quad\quad\quad\quad \text{in if } \mathbf{isTypeList}(\vec{\tau}') \text{ then } < \bigcup_{1\le i\le n} \tau'_i, \mathbf{merge\text{-}env}_{1\le i\le n}(\vec{\mu}\downarrow i) > \\ &\quad\quad\quad\quad\quad\quad\quad \text{else } < \tau, \mu > \\ &\quad\quad \text{and } < \vec{\tau}', \mu'' >= \mathbf{check\text{-}stmts}(\vec{\tau}, \mu') \\ &\quad\quad \text{in } < \tau' \cdot \vec{\tau}', \mu'' >\end{aligned}$$

The essence of case analysis (Section 2.2.2) is embodied in the **check-stmts** function. A message that ignores the type of its arguments is allowed to have an unreduced block (see the definition of **R** for blocks) as an argument. Case analysis type-checks each element of a union type separately to take advantage of this situation. The function

$$\mathbf{expand\text{-}union} : \mathsf{TypeEnv} \to \mathsf{TypeEnv^*}$$

takes a type environment μ and creates a list of type environments $\vec{\mu}$ in which no variable has a union type. This list is represented by the set

$$\{\mu' \mid \mu_1[\![\mathtt{v}]\!] = \mathtt{t} \Longrightarrow \mu'_1[\![\mathtt{v}]\!] = \mathtt{t}' \wedge \mathtt{t}' \sqsubseteq \mathtt{t} \wedge (\mathtt{t}' \in \mathsf{ObjectType} \vee \mathtt{t}' \in \mathsf{BlockType})\},$$

where "$\Longrightarrow$" is logical implication. Most of the time $\vec{\mu}$ will contain exactly one element (i.e. $\vec{\mu} =< \mu >$). The statement τ is type-checked separately in each type environment in the list. If no failure occurs and if each case can be reduced to a type then τ is reduced to the union of all the result types of the separate cases.

A block is essentially a literal. A block can be reduced to a block type if its statement list can be reduced to a type. Note that reducing a block requires adding its argument types and local range constraints to the type environment. As discussed in Section 4.4, an unreduced block may sometimes be treated like a type.

$$\begin{aligned}&\mathbf{R}[\![\mathtt{[}\vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \vec{\mathtt{y}}\ \vec{\mathtt{t}} \mid \tau\mathtt{]}]\!]\mu \\ &\quad \equiv \text{let} < \tau', \mu' > = \mathbf{R}[\![\tau]\!](\mathbf{asgn\text{-}types}(\vec{\mathtt{y}}, \vec{\mathtt{t}})(\mathbf{add\text{-}inclusion^*}(\vec{\mathtt{u}}, \vec{\mathtt{p}})\mu)) \\ &\quad\quad \text{in if } \mathbf{isType}(\tau') \text{ then } < \vec{\mathtt{t}} \to \tau', \mu' > \\ &\quad\quad\quad \text{else } < \mathtt{[}\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \vec{\mathtt{y}}\ \vec{\mathtt{t}} \mid \tau'\mathtt{]}, \mu' >\end{aligned}$$

A message send can potentially be reduced if its receiver and each of its arguments can be reduced to types (blocks are special and may not have to be reduced). Several other conditions must also hold. First, the message selector must be defined for every object type (or block type) in the reduced receiver type; second, for each of these definitions, the type of an actual argument must be a subtype of the type of the corresponding formal argument. The type of the result of the message send is determined by the types of the actual arguments and the declared return types of these definitions.

$$
\begin{aligned}
\mathbf{R}[\![\tau\ \mathtt{m}\ \vec{\tau}]\!]\mu \equiv\ & \text{let } <\tau', \mu'> = \mathbf{R}[\![\tau]\!]\mu \\
& \text{and } <\vec{\tau}', \mu''> = \textbf{check-stmts}(\vec{\tau}, \mu') \\
& \text{in if } \textbf{isTypeList}(\tau' \cdot \vec{\tau}') \text{ then } \textbf{check-send}(\tau', \mathtt{m}, \vec{\tau}', \mu'') \\
& \quad \text{else } <\tau'\ \mathtt{m}\ \vec{\tau}', \mu''>
\end{aligned}
$$

$$\textbf{check-send} : \mathsf{Type} \times \mathsf{MsgName} \times \mathsf{Type}^* \times \mathsf{TypeEnv} \rightarrow (\mathsf{TypeTree} \times \mathsf{TypeEnv})$$

Check-send will be explained by cases.

The definitions of **check-send** where the receiver is an object type or block type are essentially identical. The method type $<\vec{\mathtt{t}} \rightarrow \mathtt{u}, \mu', \tau>$ for the method that would be executed if the message $\mathtt{m}$ were sent to an instance of class $\mathtt{C}$ (blocks are instances of class BlockContext) is obtained from the type environment μ. (We assume that if $\mu_2 <\mathtt{C}, \mathtt{m}> = \underline{\text{no-def}}$ then the call to **check-send** fails.) Recursion is handled automatically since μ contains the declared type of the method currently being type-checked (see the definition of the function **P** at the end of this section).

The type environment μ' is merged with the current type environment μ and inclusion equations for the argument types are added. if the type tree τ of the method type is empty then there are no constraints and the message send is reduced to the return type of the method type. Otherwise, we try to reduce τ using the new type environment. If it can be reduced to a type then the message send is reduced to the method return type, otherwise the message send cannot be reduced.

$$
\begin{aligned}
& \textbf{check-send}(\mathtt{C}(\vec{\mathtt{s}}), \mathtt{m}, \vec{\mathtt{t}}', \mu) \text{ (receiver type is an object type)} \\
& \qquad \equiv \text{let } <\vec{\mathtt{t}} \rightarrow \mathtt{u}, \mu', \tau> = \mu_2 <\mathtt{C}, \mathtt{m}> \\
& \qquad\quad \text{and } \mu'' = \textbf{add-inclusion*}(\vec{\mathtt{t}}, \mathtt{C}(\vec{\mathtt{s}}) \cdot \vec{\mathtt{t}}')(\textbf{merge-env}(\mu, \mu')) \\
& \qquad\quad \text{in if } \tau = <> \text{ then } <u, \mu''> \\
& \qquad\qquad \text{else let } <\tau\dagger, \mu\dagger> = \mathbf{R}[\![\tau]\!]\mu'' \\
& \qquad\qquad\quad \text{in if } \textbf{isType}(\tau\dagger) \text{ then } <\mathtt{u}, \mu\dagger> \\
& \qquad\qquad\qquad \text{else } <\mathtt{C}(\vec{\mathtt{s}})\ \mathtt{m}\ \vec{\mathtt{t}}', \mu>
\end{aligned}
$$

$$
\begin{aligned}
& \textbf{check-send}(\vec{\mathtt{s}} \rightarrow \mathtt{u}', \mathtt{m}, \vec{\mathtt{t}}', \mu) \text{ (receiver type is a block type)} \\
& \qquad \equiv \text{let } <\vec{\mathtt{t}} \rightarrow \mathtt{u}, \mu', \tau> = \mu_2 <\underline{\mathsf{BlockContext}}, \mathtt{m}> \\
& \qquad\quad \text{and } \mu'' = \textbf{add-inclusion*}(\vec{\mathtt{t}}, (\vec{\mathtt{s}} \rightarrow \mathtt{u}') \cdot \vec{\mathtt{t}}')(\textbf{merge-env}(\mu, \mu')) \\
& \qquad\quad \text{in if } \tau = <> \text{ then } <u, \mu''> \\
& \qquad\qquad \text{else let } <\tau\dagger, \mu\dagger> = \mathbf{R}[\![\tau]\!]\mu'' \\
& \qquad\qquad\quad \text{in if } \textbf{isType}(\tau\dagger) \text{ then } <\mathtt{u}, \mu\dagger> \\
& \qquad\qquad\qquad \text{else } <\vec{\mathtt{s}} \rightarrow \mathtt{u}'\ \mathtt{m}\ \vec{\mathtt{t}}', \mu>
\end{aligned}
$$

If the receiver is an unreduced block and it is being sent the value message then **check-send** proceeds as if it were a method type. Argument inclusion equations are added to the type environment and we try to reduce the block body τ. If it can be reduced to a type then the message send is reduced to that same type, otherwise the message send cannot be reduced.

$$
\begin{aligned}
&\textbf{check-send}(\texttt{[}\vec{\texttt{p}}\ \vec{\texttt{u}}\ \vec{\texttt{y}}\ \vec{\texttt{s}} \mid \tau\texttt{]}, \underline{\mathsf{value}}, \vec{\texttt{t}}', \mu)\ (\mathsf{value}\text{ message sent to an unreduced block})\\
&\qquad \equiv \text{let } \mu'' = \textbf{add-inclusion*}(\vec{\texttt{s}}, \vec{\texttt{t}}')\mu\\
&\qquad\quad \text{and } < \tau', \mu'' >= \mathbf{R}[\![\tau]\!]\mu'\\
&\qquad\quad \text{in if } \textbf{isType}(\tau') \text{ then } < \tau', \mu'' >\\
&\qquad\qquad \text{else } < \texttt{[}\vec{\texttt{p}}\ \vec{\texttt{u}}\ \vec{\texttt{y}}\ \vec{\texttt{s}} \mid \tau\texttt{]}\ \underline{\mathsf{value}}\ \vec{\texttt{t}}', \mu >
\end{aligned}
$$

If the receiver is a type variable that can be evaluated to something besides itself then **check-send** is called on the something, otherwise the message send cannot be reduced

$$
\begin{aligned}
&\textbf{check-send}(\texttt{p}, \texttt{m}, \vec{\texttt{t}}', \mu)\ (\text{receiver type is a type variable})\\
&\qquad \equiv \text{let } \texttt{t} = \textbf{evaluate-in}(\texttt{p}, \mu)\\
&\qquad\quad \text{in if } \texttt{t} = \texttt{p} \text{ then } < \texttt{p}\ \texttt{m}\ \vec{\texttt{t}}', \mu >\\
&\qquad\quad \text{else } \textbf{check-send}(\texttt{t}, \texttt{m}, \vec{\texttt{t}}', \mu)
\end{aligned}
$$

If the receiver is a union type then the message send can be reduced only if it can be reduced for each element of the union type. The result of a successful reduction is the union of all the individual reductions.

$$
\begin{aligned}
&\textbf{check-send}(\texttt{t}_1\ \texttt{+}\ \texttt{t}_2\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{t}_n, \texttt{m}, \vec{\texttt{t}}', \mu)\ (\text{receiver type is a union type})\\
&\qquad \equiv \text{let } < \tau_1, \mu_1 >= \textbf{check-send}(\texttt{t}_1, \texttt{m}, \vec{\texttt{t}}', \mu)\\
&\qquad\quad \text{and } < \tau_2, \mu_2 >= \textbf{check-send}(\texttt{t}_2, \texttt{m}, \vec{\texttt{t}}', \mu)\\
&\qquad\qquad \vdots\\
&\qquad\quad \text{and } < \tau_n, \mu_n >= \textbf{check-send}(\texttt{t}_n, \texttt{m}, \vec{\texttt{t}}', \mu)\\
&\qquad\quad \text{and } \vec{\tau} =< \tau_1, \tau_2, \ldots, \tau_n >\\
&\qquad\quad \text{and } \vec{\mu} =< \mu_1, \mu_2, \ldots, \mu_n >\\
&\qquad\quad \text{in if } \textbf{isTypeList}(\vec{\tau}) \text{ then } < \bigcup_{1 \le i \le n} \tau_i, \textbf{merge-env}_{1 \le i \le n}\mu_i >\\
&\qquad\qquad \text{else } < \texttt{t}_1\ \texttt{+}\ \texttt{t}_2\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{t}_n\ \texttt{m}\ \vec{\texttt{t}}', \mu >
\end{aligned}
$$

The function **check-super** (not shown) is almost exactly the same as **check-send**. The only differences are that the receiver type is guaranteed (by definition) to be an object or block type, and that the class containing the appropriate method type (the "C" in $\mu_2 < \texttt{C}, \texttt{m} >$) is supplied as an explicit argument to **check-super** rather than being obtained from the receiver type.

$$
\begin{aligned}
\mathbf{R}[\![\tau\ \texttt{C::m}\ \vec{\tau}]\!]\mu &\equiv \text{let } < \tau', \mu' >= \mathbf{R}[\![\tau]\!]\mu\\
&\quad \text{and } < \vec{\tau}', \mu'' >= \textbf{check-stmts}(\vec{\tau}, \mu')\\
&\quad \text{in if } \textbf{isTypeList}(\tau' \cdot \vec{\tau}') \text{ then } \textbf{check-super}(\tau', \texttt{m}, \vec{\tau}', \mu'', \texttt{C})\\
&\qquad \text{else } < \tau'\ \texttt{C::m}\ \vec{\tau}', \mu'' >
\end{aligned}
$$

$$\textbf{check-super} : \mathsf{Type} \times \mathsf{MsgName} \times \mathsf{Type*} \times \mathsf{TypeEnv} \times \mathsf{ClassName} \rightarrow (\mathsf{TypeTree} \times \mathsf{TypeEnv})$$

The functions **E** and **F** to compute a method environment in which every possible call point $< \texttt{C}, \texttt{m} >$ for each method is assigned a copy of the method definition. The *call points* for a given method consists of the class in which it is defined and every class that can inherit it. The *call sites* for a given method consists of all the message sends in the source code that could possible cause this method to be invoked.

The **E** and **F** functions use Kamin's technique of resolving inheritance with the paradoxical **Y** combinator [Kam88]. Given a class hierarchy, **E** produces a function from one method environment to another. The new environment is defined recursively in terms of the old environment. The class hierarchy H is used to base the recursion. Methods are inherited, one level per each successive environment, by all subclasses of the defining class. Taking the fixed point of $\mathbf{E}[\![\mathtt{H}]\!]$ resolves all inheritance and yields the desired method environment.

$$
\begin{array}{l}
\mathbf{F} : \mathsf{Hier} \to \mathsf{MethodEnv} \\
\mathbf{F}[\![\mathtt{H}]\!] \equiv \mathbf{Y}(\mathbf{E}[\![\mathtt{H}]\!]) \\
\\
\mathbf{E} : \mathsf{Hier} \to \mathsf{MethodEnv} \to \mathsf{MethodEnv} \\
\mathbf{E}[\![\mathtt{H}]\!]\delta \equiv \lambda < \mathtt{C},\ \mathtt{m}>. \\
\qquad\qquad \text{let } \mathtt{C}\ \mathtt{S}\ \vec{\mathtt{w}}\ \vec{\mathtt{t}}\ \vec{\mathtt{x}}\ \vec{\mathtt{t}}'\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \mathtt{F} = \textbf{find-class}(\mathtt{C},\ \mathtt{H}) \\
\qquad\qquad \text{and } \mathtt{M} = \textbf{get-method}(\mathtt{m}, \mathtt{F}) \\
\qquad\qquad \text{in if } \mathtt{M} = \underline{\mathsf{no\text{-}def}} \text{ then } \delta < \mathtt{S}, \mathtt{m} > \text{ else } \mathtt{M}
\end{array}
$$

Type declarations in some methods are expressed in terms of the special type *Self*. Once every call point has been assigned a method, *Self* can be expanded into a contextually correct object type. *Self* is expanded into a specific receiver type, if one is declared, or into the object type for the class C. The expansion value for *Self* is $\mathtt{t}_{self}$. It is computed by checking to see if the receiver type of the method is <u>Self</u>. If it is, no specific receiver type was declared and $\mathtt{t}_{self}$ is the object type for class C. Otherwise, $\mathtt{t}_{self}$ is the specific receiver type declaration. The actual method type is computed from the declared type by a uniform substitution of $\mathtt{t}_{self}$ for *Self*. The *Self* instantiation is accomplished by the **instantiate-Self** function.

In addition to expanding *Self*, the **instantiate-Self** function builds a method type for each call point $< \mathtt{C}, \mathtt{m} >$. A new type variable $\mathtt{t}_{rtn}$ is created to represent the return value of the method. Type declarations for instance and class variables are added to an empty type environment to give a new type environment μ. The environment μ is updated with *Self*-expanded types for <u>self</u>, <u>returnValue</u>, method arguments, and temporaries to give μ'. μ' is updated with inclusion equations for the method return type and type variable ranges to give μ''. The result of **instantiate-Self** is a *Self*-expanded method type based on the receiver and argument types, μ'', and the method body e.

$$
\begin{array}{l}
\textbf{instantiate-Self} : \mathsf{Hier} \to \mathsf{MethodEnv} \to \mathsf{MethodTypeEnv} \\
\textbf{instantiate-Self}[\![\mathtt{H}]\!]\delta \\
\qquad \equiv \lambda < \mathtt{C}, \mathtt{m} > . \\
\qquad\qquad \text{let } \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \mathtt{s}\ \mathtt{m}\ \vec{\mathtt{y}}\ \vec{\mathtt{t}}\ \Uparrow\!\mathtt{u}\ \vec{\mathtt{z}}\ \vec{\mathtt{t}}'\ \mathtt{e} = \delta < \mathtt{C}, \mathtt{m} > \\
\qquad\qquad \text{and } \mathtt{t}_{self} = (\text{if } \mathtt{s} = \underline{\mathsf{Self}} \\
\qquad\qquad\qquad\qquad \text{then let } \mathtt{C}\ \mathtt{S}\ \vec{\mathtt{w}}\ \vec{\mathtt{s}}\ \vec{\mathtt{x}}\ \vec{\mathtt{s}}'\ \vec{\mathtt{p}}'\ \vec{\mathtt{u}}'\ \mathtt{F} = \textbf{find-class}(\mathtt{C}, \mathtt{H}) \text{ in } \mathtt{C}(\vec{\mathtt{p}}') \\
\qquad\qquad\qquad\qquad \text{else } \mathtt{s}) \\
\qquad\qquad \text{and } \mathtt{t}_{rtn} = \textbf{new-typevar}() \\
\qquad\qquad \text{and } \mu = \textbf{asgn-class-types}(\mathtt{C}, \mathtt{H}) < \bot, \bot, \bot, \mathtt{H} > \\
\qquad\qquad \text{and } \mu' = \textbf{asgn-types}(\underline{\mathsf{self}} \cdot \underline{\mathsf{returnValue}} \cdot \vec{\mathtt{y}} \cdot \vec{\mathtt{z}}, (\underline{\mathsf{Self}} \cdot \mathtt{t}_{rtn} \cdot \vec{\mathtt{t}} \cdot \vec{\mathtt{t}}')[\mathtt{t}_{self}/\underline{\mathsf{Self}}])\mu \\
\qquad\qquad \text{and } \mu'' = \textbf{add-inclusion*}(\mathtt{u} \cdot \vec{\mathtt{u}}, \mathtt{t}_{rtn} \cdot \vec{\mathtt{p}})\mu' \\
\qquad\qquad \text{in } < (\mathtt{s} \cdot \vec{\mathtt{t}} \to \mathtt{t}_{rtn})[\mathtt{t}_{self}/\underline{\mathsf{Self}}], \mu'', \mathtt{e} >
\end{array}
$$

Even though every call point for a method has been given a type with *Self* expanded, no type trees have been reduced. Reducing the type tree for a method type requires type information for the messages it sends. The method type environment **instantiate-Self**($\mathbf{F}[\![\mathtt{H}]\!]$) provides a first approximation for this type information. Some method definitions may not be type-correct in **instantiate-Self**($\mathbf{F}[\![\mathtt{H}]\!]$) at all its call points. Other method definitions might be type-correct in

instantiate-Self($\mathbf{F}[\![\mathtt{H}]\!]$), but might not be type-correct in a version of **instantiate-Self**($\mathbf{F}[\![\mathtt{H}]\!]$) that has had types for bad call points removed. Thus, reducing type trees in the entire hierarchy is an iterative process that must be repeated until the method type environment reaches a steady state.

The function $\mathbf{G}[\![\mathtt{H}]\!]$ takes a method type environment and prunes out all methods that are not type-correct. Every method in the resulting type environment is type-correct in the original type environment, but not necessarily in the resulting one. This is because deleting one method might make another method type-incorrect. In order to obtain a type environment in which every method is type-correct, $\mathbf{G}[\![\mathtt{H}]\!]$ must be repeatedly applied until the resulting type environment does not change. This is performed by the **repeat** function. The initial method type environment **instantiate-Self**($\mathbf{F}[\![\mathtt{H}]\!]$) is a finite function (there are a finite number of classes and methods in the hierarchy) and applying $\mathbf{G}[\![\mathtt{H}]\!]$ either does nothing or removes information (i.e. $\mathbf{G}[\![\mathtt{H}]\!]$ is monotonic), so **repeat**($\mathbf{G}[\![\mathtt{H}]\!]$, **instantiate-Self**($\mathbf{F}[\![\mathtt{H}]\!]$)) is guaranteed to converge. Given a hierarchy, the **Method-types** function produces a method type environment in which every method is type-correct. The function $\mathbf{G}$ is described in detail below.

$$\begin{array}{l}
\textbf{Method-types} : \mathsf{Hier} \rightarrow \mathsf{MethodTypeEnv} \\
\textbf{Method-types}[\![\mathtt{H}]\!] \equiv \lambda\mathtt{H}.\textbf{repeat}(\mathbf{G}[\![\mathtt{H}]\!], \textbf{instantiate-Self}(\mathbf{F}[\![\mathtt{H}]\!]))
\end{array}$$

$$\begin{array}{l}
\textbf{repeat} : (\mathsf{MethodTypeEnv} \rightarrow \mathsf{MethodTypeEnv}) \times \mathsf{MethodTypeEnv} \rightarrow \mathsf{MethodTypeEnv} \\
\textbf{repeat}(\gamma, \eta) \equiv \text{if } \gamma(\eta) = \eta \text{ then } \eta \text{ else } \textbf{repeat}(\gamma, \gamma(\eta))
\end{array}$$

$$\begin{array}{l}
\mathbf{G} : \mathsf{Hier} \rightarrow \mathsf{MethodTypeEnv} \rightarrow \mathsf{MethodTypeEnv} \\
\mathbf{G}[\![\mathtt{H}]\!]\eta \equiv \lambda < \mathtt{C}, \mathtt{m} > . \\
\qquad\qquad \text{let } < \vec{\mathtt{t}} \rightarrow \mathtt{u}, \mu, \tau > = \eta < \mathtt{C}, \mathtt{m} > \\
\qquad\qquad \text{in if } \mathbf{R}[\![\tau]\!]\mu = \mathit{fail} \text{ then } \underline{\mathsf{no\text{-}def}} \\
\qquad\qquad\quad \text{else let } < \tau', \mu' > = \mathbf{R}[\![\tau]\!]\mu \\
\qquad\qquad\qquad \text{in if } \textbf{isType}(\tau') \text{ then } < \vec{\mathtt{t}} \rightarrow \mathtt{u}, \mu', <>> \\
\qquad\qquad\qquad\quad \text{else } < \vec{\mathtt{t}} \rightarrow \mathtt{u}, \mu', \tau' >
\end{array}$$

Given a hierarchy and a method type environment, $\mathbf{G}[\![\mathtt{H}]\!]\eta$ defines a type-inference function for a method m inherited by a class C denoted by $< \mathtt{C}, \mathtt{m} >$. If reduction of the type tree τ fails then the method is not type-correct, the type for $< \mathtt{C}, \mathtt{m} >$ is not defined, and no-def is returned. Otherwise, a method type is returned with either an empty type tree, if the tree reduced to a type, or a partially reduced type tree.

In Kamin's semantics of Smalltalk the semantic function $\mathbf{P}$ takes an expression, a class hierarchy, and an state (image), and then resolves all method inheritance based on the class hierarchy, creates a context, evaluates the expression, and returns an answer. We modify $\mathbf{P}$ to include type-inferencing of the inheritance hierarchy and the expression. We also introduce, in the true Smalltalk style, (typed) temporary variables for the expression. The semantic environment ρ is for objects and μ is a type environment for variables, methods, and inclusion equations.

$$\begin{array}{l}
\mathbf{P} : \mathsf{Prg} \rightarrow \mathsf{State} \rightarrow \mathsf{Answer} \\
\mathbf{P}[\![\underline{\mathsf{evaluate}}\ \vec{z}\ \vec{\mathtt{t}}\ \mathtt{e}\ \underline{\mathsf{in}}\ \mathtt{H}]\!]\psi \\
\qquad\qquad\qquad \equiv \text{let } \rho = \mathbf{D}[\![\mathtt{H}]\!] \\
\qquad\qquad\qquad\quad \text{and } \mu = \textbf{asgn-types}(\vec{z}, \vec{\mathtt{t}}) < \bot, \textbf{Method-types}[\![\mathtt{H}]\!], \bot, \mathtt{H} > \\
\qquad\qquad\qquad\quad \text{and } o = \textbf{new-object}(\psi) \\
\qquad\qquad\qquad\quad \text{and } \psi' = \psi[1 \rightarrow o \rightarrow < \phi_\bot[\vec{z} \rightarrow \vec{\bot}], \mathit{Context} >] \\
\qquad\qquad\qquad\quad \text{and } < \tau, \mu' > = \mathbf{R}[\![\mathtt{e}]\!]\mu \\
\qquad\qquad\qquad\quad \text{in if } \textbf{isType}(\tau) \text{ then } \mathbf{E}[\![\mathtt{e}]\!]\rho\ \mathit{Object}\ o\ \textbf{print}\psi'
\end{array}$$

Definition 3 (Object/type consistency) *An object o is consistent with a mapping from objects to types ξ, relative to a class hierarchy* H, *and a state ψ, if*

1. $\xi(o) = \mathtt{C}(\vec{\mathtt{t}})$,
2. *the class of o is* C, *and*
3. *for each instance variable x_i of o, the type of the object referred to by x_i is a subtype of $\mathtt{t}_i$, the declared type of x_i, as evaluated in σ, where σ is a TVM such that if* $\mathtt{C}\ \mathtt{S}\ \vec{\mathtt{w}}\ \vec{\mathtt{t}}\ \vec{\mathtt{x}}\ \vec{\mathtt{t}}'\ \vec{\mathtt{p}}\ \vec{\mathtt{u}}\ \mathtt{F} = \textbf{find-class}(\mathtt{C},\mathtt{H})$ *then* $\sigma = \textbf{includes}(\mathtt{C}(\vec{\mathtt{p}}),\mathtt{C}(\vec{\mathtt{t}}),\vec{\mathtt{p}},\vec{\mathtt{u}})$, *i.e.*

$$\textit{for each } x_i \textit{ of } o,\ \xi(\psi_1(o)_1(x_i)) \sqsubseteq \textbf{evaluate}(\mathtt{t}_i,\sigma).$$

5.4 Correctness

Usually a type is described as a set of objects. Under this interpretation, operations on types are set operations. Although we have defined operations like union and inclusion on types, our types are not sets of objects. However, we can assign an object type to each object.

The type of an object depends not only on its current state but also on its past and future states. Thus, it may not be possible to decide whether an object is in a particular type, though it is often possible to show that it is not. For example, a Set is never in *Array of: Character*, and a Set containing Characters is not in *Set of: SmallInteger*. However, it is not easy to tell whether a Set containing only Characters is in *Set of: (SmallInteger + Character)*—it all depends on whether or not the Set is referenced by a variable that *requires* it to contain only Characters.

If each object had a type assigned to it then we could check whether an object was consistent with its type by checking whether its class was consistent with its type and whether the types assigned to the values of its instance variables were included in the declared types of its instance variables. This assignment would be consistent if every object was consistent with the type assigned to it. Although this type-assignment might not be unique, any consistent type-assignment could be thought of as describing the types of all objects. These ideas are formalized in Definition 3.

A state is consistent with a type-assignment ξ if every object in the state is consistent with ξ.

Proposition: *If a type-correct program is in a state that is consistent with a type assignment ξ then any succeeding state will be consistent with ξ.*

The only way that a consistent type-assignment can become inconsistent is if the state changes. This is because the type-assignment itself is constant and, in Kamin's semantics, objects never change and the class hierarchy does not change. Thus, we can prove the proposition by showing that ξ is maintained as an invariant every place where the state can be changed.

There are two ways that the state changes in Kamin's semantics. The first is when a variable changes its value, which only occurs in the **assign** function. This happens in an assignment statement and in assigning arguments to formal parameters when evaluating a method or block.

The second way that the state changes is when a new object is created. This only occurs in the **new-object** function. This function is used to create literals or blocks, but their type is known in advance and the new object will always be of the correct type. This function is also used in the **new** primitive and to create a new context to evaluate a method. Its use in these places causes problems, because instance variables of new objects and local variables of new contexts are both initialized to nil, but the types of these variables usually do not include *UndefinedObject*. Thus, until the variables are initialized, they are not consistent. We ensure type-safety by requiring all variables whose types do not include *UndefinedObject* to be assigned before they are read, and use flow analysis to check this. However, flow analysis is not part of the type system presented here. Thus, we will change the statement of the proposition slightly.

> **Type-Safety Theorem:** *If a type-assignment is consistent with a state, except for unassigned variables, and if a variable is always assigned before it is read then any execution of a type-correct program will maintain the consistency of the type-assignment relative to the new state.*

Proof: In a type-correct program, the type of the expression of an assignment statement must be included in the type of the variable, so, if each expression returns a value that is consistent with its type then assignment statements maintain the consistency of the type-assignment. Once a variable is assigned, its value will have a type that is included in the type of the variable. Thus, if the value of any expression has a type that is included in the type of the expression then whenever a variable v is read, the type of the value of v will be consistent with the type of v.

The theorem is proved by structural induction on the number of message sends in the evaluation of an expression. The base case is where there are no message sends. Since there are no message sends there can only be assignment statements, whose right-hand sides are literals or variables. The variables either are type-consistent or will be assigned before they are read, so the assignment statements will all maintain the consistency of the type-assignment.

The induction step is to assume that any expression that can be evaluated in $n - 1$ message sends (or less) is type-safe and to prove that evaluating an expression

$$\mathsf{rcv\ k1{:}\ exp_1\ k2{:}\ exp_2 \ldots km{:}\ exp_{\mathit{m}}}$$

that requires n message sends is type-safe. Since an expression must be type-checked before it can be evaluated, we know that the types of the argument expressions to the k1:k2:...km: message are included in the types of the formal parameters of the method that is invoked. Evaluating any exp_i will take less than n message sends, so each argument is consistent with its corresponding type. When the new context is created and the method is executed, the formal parameters of the method will be bound to objects whose type is included in the type of the parameter. Methods also create temporary variables, but they are unassigned, so invoking a method maintains the consistency of the type-assignment (except for unassigned variables).

We assumed that the expression involves n message sends. One of them is the k1:k2:...km: message, so evaluating the body of the method will involve less than n message sends. Thus, every expression evaluation in it will result in an object whose type is included in the type of the expression that produced it. In particular, the type of the object returned from the method will be included in the type of the expression in the return statement, and, according to the type-checking rules for the return statement, the type of the expression in a return statement must be included in the declared return type of the method. Thus, the type of the object returned as a result of the k1:k2:...km: message will be included in the return type of the method.

The type of

$$\mathsf{rcv\ k1{:}\ exp_1\ k2{:}\ exp_2 \ldots km{:}\ exp_{\mathit{m}}}$$

is the union of the return types of a number of methods, but it certainly includes the return type of the method that was invoked. Thus, the result of the expression, which is the object being returned by the method, has a type that is included in the type of the expression. ∎

Type-safety also depends on all the primitives being given correct types. Since primitives are written in a language other than Smalltalk, it is impossible to reason about them within the framework presented here. A few primitives are inherently unsafe. These are primarily used by the debugger. Most of the primitives can have their types described by this type system, however.

The biggest weakness in the type system is that a program has to be analyzed to ensure that all variables whose types do not include *UndefinedObject* are assigned a value before they are read. This has been implemented, but the code has not been integrated with the compiler yet.

Chapter 6

Conclusion

We have shown that our type system for Smalltalk is type-safe. The type-inference system has been implemented in Smalltalk and used in the TS optimizing compiler for Smalltalk [JGZ88]. It has been able to solve most type-checking problems in the standard Smalltalk-80 class hierarchy. Our type system differs from other type systems for object-oriented programming languages by acknowledging that only implementation is inherited, not specification, and by handling case analysis of union types automatically.

It is interesting to consider the historical development of the type system from a user's perspective. The basic form of type information has not changed, only the way that it is obtained and used has changed. The first version was purely declarative and used signature types to represent large (unbounded) union types. All variables required type declarations and the return type of a method was completely determined by these declarations. This was good for type-checking and optimization since the type of a method depended only on the messages it sent. Object types were highly visible and methods could be optimized to their fullest extent based on their definitions alone. Type-checking had no noticeable effect on performance.

Unfortunately, there were several pragmatic problems with the initial version of the type system. Type-checking imposed a strict order on method compilation. For a method to type-check every method it called must have been type-checked. Another problem was the sheer effort involved in declaring types for methods, especially those with elaborate signature types. However, the biggest problem was that, even with signature types, we were not able to type-check methods like = (equality) for class LookupKey (see Figure 2.11). This elevated type-inferencing from being a mere convenience to being a necessity.

With the present implementation of the type-inference algorithm, it is possible to infer a "type" for any method in the Smalltalk-80 image. Method definitions can be processed in any order and creating a typed method requires no more user effort than creating an untyped one. Unfortunately, the inferred types usually have unsatisfied constraints and cannot be optimized. Constraint-free types can usually be inferred only for top-level methods (i.e. methods with no arguments) or for methods with argument type declarations. Type-inference takes noticeably longer than type-checking, but no performance tuning has been done and it still represents only a small fraction of the time required by the entire compilation process.

While it may seem that we have jumped from one extreme to the other, the current type system still provides all the power of the earlier system by allowing explicit type declarations for any or all variables. Given the best of both worlds, the next problem is to determine the right balance of type declarations and type-inference that will minimize programmer effort and maximize potential optimization. This will require extensive experimentation with the type system and using it on an entire large application. Unfortunately, due to instability in the code generation process, this is not possible at present. It is easy to envision how the system might work in practice.

In general, development of a Smalltalk application will follow the following path. At first, new classes will be created and pieced together with other classes. In the early stages of development, type-inference will be used to infer general types with little regard for optimization. As the application matures, standard components will be identified and new class hierarchies will arise. There are then two ways for more concrete type information to seep into the system. Type declarations may be added to methods once they have settled down, or top-level methods will provide type information to all lower-level methods. It will also become obvious where real polymorphism is required and where it is not.

Given the potentially exponential explosion of case analysis with union types, efficiency is a major concern. In practice, however, union types are limited by what actually happens at run-time. Our experience has shown that union types are either small (less than three elements, e.g. *True + False*, *View + UndefinedObject*, and *LargeNegativeInteger + SmallInteger + LargePositiveInteger*) and can be compiled into case statements, or are very large (e.g. everything that understands =) and are handled using method lookup. This way, traditional code runs fast plus we still have the power of true late-bound procedure calls.

Our type system is more complicated than other type systems for object-oriented programming languages. Whether this complication is justified depends partly on whether a new language is being defined or whether a type system is being defined for Smalltalk. Current Smalltalk programming practice requires a type system like ours.

Bibliography

[All86] Lloyd Allison. *A Practical Introduction to Denotational Semantics.* Volume 23 of *Cambridge Computer Science Texts*, Cambridge University Press, 1986.

[ASU86] Alfred C. Aho, Ravi Seth, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Atk86] Robert G. Atkinson. Hurricane: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 151–166, November 1986. printed as SIGPLAN Notices, 21(11).

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, November 1986. printed as SIGPLAN Notices, 21(11).

[BHJL87] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[BI82] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 133–139, 1982.

[BMW86] Mark B. Ballard, David Maier, and Allen Wirfs-Brock. QUICKTALK: A Smalltalk-80 dialect for defining primitive methods. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 140–150, November 1986. printed as SIGPLAN Notices, 21(11).

[Bor79] Alan Borning. *ThingLab — A Constraint Oriented Simulation Laboratory.* Technical Report SSL-79-3, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, July 1979.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science, n. 173*, pages 51–67, Springer-Verlag, 1984.

[Car85a] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages, Proceedings of the 13th Summer School of the LITP*, Le Val d'Ajol, Vosges (France), May 1985.

[Car85b] Luca Cardelli. *Basic Polymorphic Typechecking.* Technical Report 119, AT&T Bell Laboratories, Murray Hill, NJ, 1985.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–79, 1988.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CR85] S. Cook and G. Roberts. A compile-time type-checking system for Smalltalk-80. *Journées Langages Orientés Objet*, :40–45, 1985.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[CW86] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 119–130, November 1986. printed as SIGPLAN Notices, 21(11).

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, 1984.

[DT88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *Computing Surveys*, 20(1):29–72, March 1988.

[GGJ*88] P. Gilbert, J. Graver, R. Johnson, J. Loyall, and L. Zurawski. *Typed Smalltalk Working Papers: 1988.* Technical Report UIUCDCS-R-88-1457, University of Illinois at Urbana-Champaign, 1988.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Gra87] Justin Graver. *Adding Type Specification and Type-Checking Capabilities to Smalltalk-80.* Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[HJ90] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and double dispatching in Smalltalk-80. 1990. To appear in the *Journal of Object-Oriented Programming.*

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programs. *IEEE Software*, :71–79, September 1987.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88, Object-Oriented Programming Systems, Languages and Applications*, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[JM76] Neil D. Jones and Steven S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. In *Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 77–94, 1976.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Joh86] Ralph E. Johnson. Type-checking Smalltalk. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 315–321, November 1986. printed as SIGPLAN Notices, 21(11).

[Kam88] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[KU78] Kaplan and Ullman. A general scheme for the automatic inference of variable types. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 60–75, 1978.

[Loy88] Joseph Loyall. *High-level Optimization in a Typed Smalltalk Compiler.* Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MPS84] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, 1984.

[MR85] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, 1985.

[MS82] David MacQueen and Ravi Sethi. A higher order polymorphic type system for applicative languages. In *ACM Symposium of LISP and Functional Programming*, pages 243–252, 1982.

[Rey69] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Proceedings of IFIP Congress 1968*, pages 456–461, 1969.

[Rey81] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, North-Holland, Amsterdam, 1981.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, November 1986. printed as SIGPLAN Notices, 21(11).

[ST84] Norihisa Suzuki and Minoru Terada. Creating efficient systems for object-oriented languages. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 290–296, 1984.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., Reading, MA, 1986.

[Str87] Bjarne Stroustrup. The evolution of C++: 1985 to 1987. In *USENIX C++ Workshop Proceedings*, pages 1–21, 1987.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 187–199, 1981.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87, Object-Oriented Programming Systems, Languages and Applications*, pages 227–242, December 1987. printed as SIGPLAN Notices, 22(12).

[Wan86] Mitchell Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, 1986.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Symposium on Logic in Computer Science*, pages 37–44, 1987.

[Zur90] Lawrence W. Zurawski. *Source-Level Debugging of Globally Optimized Code with Expected Behavior*. 1990. Ph.D. thesis in preparation, University of Illinois at Urbana-Champaign.

Vita

Justin Owen Graver was born on May 9, 1962, in Lincoln, Nebraska. After graduating from Lincoln Southeast High School in May, 1980, he entered the University of Nebraska at Lincoln. He graduated magna cum laude in May, 1984, with a Bachelor of Science in Computer Science in the College of Engineering. He then entered graduate study at the University of Illinois at Urbana-Champaign, where he received the degree of Master of Science in Computer Science in January, 1987. He received the degree of Doctor of Philosophy in Computer Science in October, 1989, from the Graduate College of the University of Illinois. Upon completing his doctorate, Dr. Graver accepted an Assistant Professorship at the University of Florida at Gainesville.