THE TS TYPE SYSTEM

BY

JIN-HO TAN

B.S., University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

ABSTRACT

The Typed Smalltalk project is building an optimizing compiler for a variant of Smalltalk. Typed Smalltalk differs from Smalltalk in that programs can be annotated with type information, which the compiler can use to optimize programs. The type system for Typed Smalltalk is the first type system for a Smalltalk-like language that is both type-safe and flexible enough to handle the kind of polymorphic procedures used in Smalltalk. There is also a type inference system that can infer the type of each procedure. Previous papers have focused on the specification of the type system. This thesis describes the design and implementation of the current Typed Smalltalk (TS) type system.

ACKNOWLEDGEMENTS

I would like to thank Professor Ralph Johnson, my advisor, for his help and guidance; and Professor Justin Graver for his comments and previous work on the type system. Many of the classes he wrote are reused in the type system in this thesis. In addition, I would also like to express my thanks to Carl Mcconnell, Bob Hinkle, Jean Wallemacq and Alan Durham for their helpful comments. Last but not least, I thank Lim Chieng-Fai, Han Wen-Hsien, Bui Thuyanh Hong and the UIUC Singaporean Students Association for making my stay at Urbana/Champaign more than an academic experience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The Typed Smalltalk project is building an optimizing compiler for a variant of Smalltalk [JGZ88]. Typed Smalltalk differs from Smalltalk in that programs can be annotated with type information, which the compiler can use to optimize programs. The type system for Typed Smalltalk is the first type system for a Smalltalk-like language that is both type-safe and flexible enough to handle the kind of polymorphic procedures used in Smalltalk [Joh86, GJ90]. There is also a type inference system that can infer the type of each procedure [Gra89]. Previous papers have focused on the specification of the type system. This thesis describes the design and implementation of the current Typed Smalltalk (TS) type system.

Most type systems for object oriented languages unify types and classes, and ensure that subclasses are subtypes. C++[Str86], Eiffel[Mey87], and Trellis/Owl[SCB+86] all are good examples of this. Although this results in a type system that is simple, easy to use and implement, it reduces the flexibility of code reuse which is one of the strong suits of object oriented programming. Typed Smalltalk makes a strong distinction between types and classes and between subtyping and subclassing. Typed Smalltalk is not the only language to separate types and classes. For example, POOL-T[AvdL90] does as well, but uses signatures for types instead of using sets of parameterized classes, as in Typed Smalltalk. Part of this difference is because the Typed Smalltalk type system has been designed to make programs easier to optimize.

The main goal of the Typed Smalltalk type system is to provide a type system that can be used to describe typical Smalltalk programs. Thus, the type system does not presribe how to program, but describes what the program has done. This is another reason why subclassing is not subtyping. A Smalltalk class inherits the implementation but not the specification of its superclass. Programmers often make use of this fact, so that few subclasses in the Smalltalk-80 image are subtypes of their superclass.

The next chapter gives an informal description of the type system. Chapter 3 describes type inference. Chapter 4 describes the various parts of the implementation of the type system.

1.1 Types

The TS type system is powerful but at the cost of being complex. It has many different kinds of types. The most important are object types, block types, union types, signature types, <*Anything*>, <*Nothing*>, <*IllegalBlock*> and type variables. Every class in TS is associated with an object type. For example, the object type associated with Character is <*Character*> and the object type associated with SmallInteger is <*SmallInteger*>. Other classes like Array and Dictionary can contain elements of various types, so their object types are parameterized. For example, the object type of Array is <*Array of: ElementType*> and the object type of Dictionary is <*Dictionary of: KeyType of: ValueType*>. The type parameters, <*ElementType*>, <*KeyType*>, and <*ValueType*>, are instantiated to a real type when used. <*Array of: Character*> and <*Dictionary of: SmallInteger of: Character*> are instances of <*Array of: ElementType*> and <*Dictionary of: KeyType of: ValueType*> respectively. Syntactically, an object type consists of a class name with a possibly empty list of type parameters.

While object types describe regular Smalltalk objects, block types describe blocks. A block type consists of the class name 'Block', a possibly empty list of argument types, and a return type. For example, <*Block of: SmallInteger returns: (True+False)*> is the type of a block that takes a SmallInteger and returns a True object or a False object.

To handle polymorphism, union types and signature types are used. If a variable has a union type such as <*UndefinedObject + SmallInteger*>, then it may refer to an object of type <*UndefinedObject*> or <*SmallInteger*>. If a variable has a signature type such as <*'+ <Integer>* $\uparrow$*<Integer>; − <Integer>* $\uparrow$*<Integer> '*>, then it may refer to any object that satisfies the interfaces for '+' and '−' above. A union type is typically used for ad hoc polymorphism where individual types in the union may have few or no common interfaces. On the other hand, a signature type describes a set of object types with common interfaces. For instance, a signature type such as <*'printString* $\uparrow$*<String> '*> includes all the object types that are printable.

<*Anything*> and <*Nothing*> are the greatest and least type in the type domain. In addition, <*Anything*> may be the type of an argument that is not used in a method (see Figure 1.1). On the other hand, <*Nothing*> may be the return type of a method that never returns to the caller. A good example is Object>>primitiveFailed in Figure 1.2.

```
{  arguments: alternativeBlock <Anything>
   returnType: <UndefinedObject> }

ifFalse: alternativeBlock
        ↑nil
```

Figure 1.1: Usage of <*Anything*>

```
{  returnType: <Nothing> }

primitiveFailed
        ↑(self error: 'a primitive has failed') typeIs: '<Nothing>'
```

Figure 1.2: Usage of <*Nothing*>

When a type error occurs inside a block, we say that the block has type <*IllegalBlock*>. The type error in an <*IllegalBlock*> may not be fatal if the block is never evaluated. A naive example would be true ifFalse: [self msg]. Since the argument type of True>>ifFalse: is <*Anything*>, any type errors in [self msg] will be ignored. We shall see more of <*IllegalBlock*> when discussing case analysis.

Finally, a type variable represents an unknown type. An uninstantiated class type parameter is a free type variable.

1.2 Pseudo-types

The pseudo-type <*Self*> refers to the type of the pseudo-variable self. Using this pseudo-type, recursive types commonly found in object-oriented programming can be defined. For example, the user may declare the type of False>>and: to be <*'and: <Anything> ↑<Self>'*>. In this sense, <*Self*> is similar to the recursion variable in F-bounded polymorphism [CCH+89]. In F-bounded polymorphism, one would write the type of False as $\mu t.\{$ *..., and: <Anything> ↑t, ...*$\}$.

Pseudo-types also enable a subclass to inherit a method's interface together with its implementation. Like the pseudo-variable self, which is instantiated with a new class when inherited, the pseudo-type <*Self*> is instantiated with a new object type when inherited. However, this does not mean that subtyping is subclassing.

Similarly, the pseudo-types <*Self class*> and <*Self new*> denote the class type and instance type of an object type respectively. These types are used to type-check methods in the metaclasses. For example, we may write the type of new as <*'new* $\uparrow$<*Self new*>*'*> and the type of class as <*'class* $\uparrow$<*Self class*>*'*>.

1.3 Subtyping

The previous section mentioned subtyping without defining it; this section defines subtyping in Typed Smalltalk. A type s is a subtype of another type t, denoted $s \sqsubseteq t$, if and only if s can be used everywhere t is used. An object type is a subtype of another object type if and only if both object types are identical. Hence, <*SmallInteger*> $\sqsubseteq$<*SmallInteger*> and <*Array of: Character*> $\sqsubseteq$<*Array of: Character*>, but <*Array of: Character*> $\not\sqsubseteq$<*Array of: (Character + SmallInteger)*>. The type system described in this thesis does not assume any subtyping relationship between a class and its subclasses, which effectively flattens the Smalltalk class hierarchy. This is a consequence of the fact that a Smalltalk programmer may subclass from a parent class for code reuse only. A good example is Set and its subclass, Dictionary. Since there is no subtyping relationship between different object types, the compiler may inline a message send when its receiver type is an object type.

A block type s is a subtype of another block type t if the argument types of s are supertypes of the argument types of t and the return type of s is a subtype of the return type of t. We say that the block type is contravariant in its argument types.

Because no subtype relation exists between different object types, explicit union types are needed to express polymorphism. An object type or a block type is a subtype of a union type if it is an element of the union type. A union type s is a subtype of another union type t if s contains a subset of types of t. An object type or a block type is a subtype of a signature type if it 'understands' all the typed messages in a signature type. In addition, a signature type s is a subtype of another signature type t if for every typed message in t, the corresponding typed message in s is a subtype of the typed message in t. Like block types, a typed message s is a

```
{ classTypeParameters: KeyType ValueType <'= ValueType ↑<True+False>'>
  receiverType: <Self>
  arguments: anObject <ValueType>
  blockArguments: each <ValueType>
  returnType: <True+False>
  constraints: false
  onlyForTS: false }

includes: anObject
    self do: [ :each | anObject = each ifTrue: [↑true]].
    ↑false
```

Figure 1.3: Definition for Dictionary>>includes:.

subtype of another typed message t if the argument types of t are subtypes of the argument types of s, and the return type of s is a subtype of the return type of t.

1.4 Method Type Declaration

A method type declaration documents the argument types, the return type, and the types of temporaries and block arguments of a method. This type information is kept together with the source code. Furthermore, the method type declaration also allows the user to insert compiler directives. A typical typed method is shown in Figure 1.3. It belongs to class Dictionary which has two type parameters: *KeyType* and *ValueType*. The argument, anObject, and the block argument, each, must both be a subtype of <'= *ValueType* ↑<*True+False*>'>. The constraints flag, when set to false, tells the user that the current method is fully type-checked. If the user does not declare the argument types of a method, the type system may not be able to type-check all the expressions in the method. In this case, the constraints flag is set to true and type-checking is deferred until the current method is called by another method. The onlyForTS flag tells the compiler whether the current method is only used in TS. This allows TS methods and standard Smalltalk-80 methods to co-exist in one image.

1.5 Abstract Classes

Abstract classes are classes with incomplete implementation. A class is declared to be abstract by a menu option in the browser. It is a common practice to code an abstract class method as 'self subclassResponsibility'. Furthermore, some methods in an abstract class may call an unimplemented method. For instance, class Collection implements addAll: which calls add:. However, add: is implemented in the subclasses of Collection, but not in class Collection itself. Thus, the type system needs to be forgiving when type-checking methods in the abstract classes. If a called method has no implementation, the type system should issue a warning. The method is re-type checked again when inherited by a subclass. On the other hand, while type-checking a method in a concrete class, the type system must check that all the called methods are implemented. If an undefined method is encountered, an error must be reported.

2 Type-checking Methods

A method is type-correct if, for each of its argument types, each of its statements is type-correct and its return type is a subtype of the declared return type. To type-check a Smalltalk method, the type system verifies the type of each expression using the appropriate type inclusion rule in Table 2.1. Table 2.2 defines the types of six Smalltalk expressions so that more complicated expressions formed from these basic ones can be type-checked.

A statement list consists of a sequence of Smalltalk statements. It appears at the top level as the method body, and also inside a block. A statement list is type-correct if all its component statements are.

A return expression such as $\uparrow expr$ causes the context method to exit with the result of evaluating $expr$. To make sure that $\uparrow expr$ is type-correct, $expr$ must be type-correct and the type of expr must be a subtype of the context method's return type. Recall that when a message send does not return to the caller (e.g. self keyNotFound), its type is $<Nothing>$. Since control is never returned to the context method after executing a return expression, the type of $\uparrow expr$ is also $<Nothing>$.

2.1 Type-checking assignments

Temporary variables present special problems for the type-system, especially when multiple assignments are made to the same variable. Consider the method in Figure 2.1. The temporary,

```
foo
    | temp |
    temp ← 1.
    . . .
    temp ← Array new.
    . . .
```

Figure 2.1: Reuse of temporary variable.

Type of expressions	Generic form of expression	Type-correctness conditions
Assignment	$x \leftarrow y$	$<y> \sqsubseteq <x>$
Return	$\uparrow x$	$<x> \sqsubseteq <T0>$ where $<T0>$ is the return type of the method.
Message send	$x\ msg\ arg_1 \ldots arg_n$	Let $<ret>$ be the return type of the message send. Let $<x'>\ msg\ <arg'_1> \ldots <arg'_n> \uparrow$ $<ret'>$ be the type of the called method. $<x'> = <x>$, $<arg_1> \sqsubseteq <arg'_1>$, $<arg_2> \sqsubseteq <arg'_2>$, $\vdots$ $<arg_n> \sqsubseteq <arg'_n>$, $<ret'> \sqsubseteq <ret>$.
Cascaded message send	$x\ msg_1\ y_{11}\ y_{12} \ldots y_{1k_1}$; $msg_2\ y_{21}\ y_{22} \ldots y_{2k_2}$; $\vdots$ $msg_n\ y_{n1}\ y_{n2} \ldots y_{nk_n}$.	The following type expressions: $<x>\ msg_1\ <y_{11}>, <y_{12}>, \ldots, <y_{1k_1}>$; $<x>\ msg_2\ <y_{21}>, <y_{22}>, \ldots, <y_{2k_2}>$; $\vdots$ $<x>\ msg_n\ <y_{n1}>, <y_{n2}>, \ldots, <y_{nk_n}>$ are type-correct.
Block	$[:arg_1 :arg_2\ \ldots\ :arg_n \mid$ $expr_1.\ expr_2.\ \ldots\ expr_n]$	All the expressions in the block are type-correct
Statement list	$expr_1.\ expr_2.\ \ldots\ expr_n.$	All the expressions, $expr_i$, are type-correct.

Table 2.1: Type correctness conditions

Type of expressions	Generic form of expression	Type of expression
Assignment	$x \leftarrow y$	$<y>$
Return	$\uparrow x$	$<Nothing>$
Message send	$x\ msg\ arg_1 \ldots arg_n$	The return type of the message send.
Cascaded message send	$x\ msg_1\ y_{11}\ y_{12} \ldots y_{1k_1}$; $msg_2\ y_{21}\ y_{22} \ldots y_{2k_2}$; $\vdots$ $msg_n\ y_{n1}\ y_{n2} \ldots y_{nk_n}$.	The type of the message send: $msg_n\ y_{n1}\ y_{n2} \ldots y_{nk_n}$.
Block	$[:arg_1 :arg_2\ \ldots\ :arg_n \mid$ $expr_1.\ expr_2.\ \ldots\ expr_n]$	$<$*Block of:* $<arg_1>$ *of:* $<arg_2> \ldots$ $<arg_n>$ *returns:* $<expr_n>\ >$
Statement list	$expr_1.\ expr_2.\ \ldots\ expr_n.$	$<expr_n>$

Table 2.2: Type of Smalltalk-80 expressions

```
sumAll: anOrderedCollection
    | temp |
    temp ← 0.
    anOrderedCollection do: [ :aNumber | temp ← temp + aNumber ].
    ↑temp
```

Figure 2.2: Homogenous assignment.

temp, holds a SmallInteger in the first part of the program but is reassigned to take an array later on. To type-check the above method successfully, the type system must be aware that temp has changed its type during execution. To keep track of the type of temp, the type system could use flow analysis to determine the life-span of each type of temp; or to simulate the above code at compile-time, reassigning a new type to temp whenever an assignment occurs. However, one can argue that reusing a temporary variable for different purposes is a bad practice and seldom occurs. Furthermore, the programmer can rewrite a method easily so that each temporary variable is used for one purpose only. Therefore, the current TS type system assumes that all the assigned types of a temporary have the same interface. Multiple assignments to the same temporaries are allowed as long as the assigned types are homogeneous. This will probably be sufficient to type-check most Smalltalk codes. If not, it should not be hard to retrofit flow analysis into the type system.

Consider the method in Figure 2.2. The initial type of temp is $<$*SmallInteger*$>$. Later on, temp can take any type which understands '+' (i.e. $<$*Number*$>$). Since $<$*Number*$>$ and $<$*SmallInteger*$>$ are compatible, the above method type-checks without any problem.

A more severe restriction of the above rule is that Smalltalk programmers may not 'nil out' the instance variables of an object for garbage collection. Otherwise, $<$*UndefinedObject*$>$ may need to understand potentially all the messages in the Smalltalk image. There are two ways to solve this problem. The first is to let the programmer define specialized 'null classes' to play the role of UndefinedObject. For instance, the programmer may define EmptyBinaryTree to act as the UndefinedObject for BinaryTree, EmptyList for List, and so forth. Each 'null class' understands the standard protocol for its associated 'regular' class. Alternatively, the type system may flow-analyze the program to decide whether <*UndefinedObject*> should appear in the type of a variable or not. The TS type system does not yet perform either of these tasks.

2.2 Type-checking message send

To type-check a message send, the Smalltalk method lookup routine is simulated to find the declared type of the called method. According to the type inclusion rule for message sends, the receiver type in the type declaration must match the actual receiver type. If the receiver type is parameterized, the type parameter in the formal receiver type is instantiated to the actual type parameter at this point. In addition, the formal argument type(s) must include the actual argument type(s) of the message send, and the formal return type must be included in the actual return type of the message send.

For example, suppose the declared type for Array>>at: is

$$<'< Array\ of: ElementType > at: < SmallInteger > \uparrow < ElementType >'>$$

and the actual message send is

$$<\textit{Array of: Character}> \mathsf{at}\text{:} <\textit{SmallInteger}>$$

According to the message send inclusion rule, <*Array of: Character*> must match <*Array of: ElementType*>, instantiating the formal type parameter <*ElementType*> to <*Character*>. At the same time, the formal argument <*SmallInteger*> must include the actual argument <*SmallInteger*>, which is trivially satisfied. Finally, the type of the message send must include the declared return type, <*ElementType*>. Since <*ElementType*> has been matched with <*Character*>, the type of

$$<\textit{Array of: Character}> \mathsf{at}\text{:} <\textit{SmallInteger}>$$

is <*Character*>.

```
dummy
        ↑true or: true
```

Figure 2.3: A method that requires pseudo-type conversion

2.2.1 Handling types for pseudo variables

There is an additional detail to take care of when type-checking a message send. Prior to the message send type inclusion test, any pseudo-types, <*Self*>, <*Self class*> or <*Self new*>, in the called method's declared type must be converted to the appropriate types (i.e. to the type of the receiver, the type of its class, or the type of its instances respectively). Otherwise, the type system may infer the return type of a message send incorrectly. For example, the type of True>>or: can be written as <*'or: <Anything>* ↑<*Self*>*'*> Now consider the method, `dummy`, in Figure 2.3. Suppose `dummy` is defined in class SmallInteger and the message send, `true or: true`, is being type-checked. The declared type of True>>or: must be changed to <*'or: <Anything>* ↑<*True*>*'*> so that the inferred return type of SmallInteger>>dummy is <*True*>. If the declared type of True>>or: were not converted when type-checking `true or: true` then the return type of `true or: true` would be <*Self*>. As a result, the type system will infer that the return type of SmallInteger>>dummy is <*Self*>, which is wrong. Therefore, any pseudo-types present in the declared method type must be converted to the correct form when type-checking a message send.

The receiver type of the message send determines how the pseudo-types should be converted. Suppose a similar method, `dummy2`, is defined in class True (see Figure 2.4). This time to type-check the message send, `self or: true`, the declared type of True>>or: does not require any conversion (i.e. <*'or: <Anything>* ↑<*Self*>*'*>). The rules for converting pseudo-types (with respect to the receiver type of the message send) are listed in Table 2.3.

In addition, if a subclass inherits a method, the method body with its type declaration is copied down and re-type-checked in the context of the subclass. In this case, the pseudo-types <*Self*>, <*Self class*> and <*Self new*> must be instantiated to the appropriate types in the subclass.

```
dummy2
        ↑self or: true
```

Figure 2.4: Another method that requires pseudo-type conversion

Receiver type of message send	original pseudo-type in callee's interface	converted pseudo type in caller's environment
<*Self*>	<*Self*>	<*Self*>
<*Self*>	<*Self new*>	<*Self new*>
<*Self*>	<*Self class*>	<*Self class*>
<*Self new*>	<*Self*>	<*Self new*>
<*Self new*>	<*Self new*>	– Not Applicable – for inheriting purposes only.
<*Self new*>	<*Self class*>	<*Self*>
<*Self class*>	<*Self*>	<*Self class*>
<*Self class*>	<*Self new*>	<*Self*>
<*Self class*>	<*Self class*>	<*Metaclass*> or <*Metaclass class*> depending on what the context class is.
basic type	<*Self*>	the actual type
basic type	<*Self new*>	the actual type
basic type	<*Self class*>	the actual type

Table 2.3: Conversion of pseudo-types

```
{ arguments: arg <ClassA + ClassB>
  returnType: <True+False> }

test: arg
      ↑arg foo: arg
```

Figure 2.5: Simple example of case analysis.

2.2.2 Case analysis

Sometimes case analysis is required to type-check a method successfully. Suppose class ClassA implements a method foo: that takes an object of type <*ClassA*> and returns <*True+False*>. Similarly, ClassB>>foo: takes an object of type <*ClassB*> and returns <*True+False*>.

Now consider the typed method in Figure 2.5. If no case analysis is performed on the statement arg foo: arg then type-checking will fail because the declared argument type of both ClassA>>foo: and ClassB>>foo: do not include <*ClassA + ClassB*>. Case analysis may be necessary whenever a variable with union type occurs in the receiver and the argument(s) of a message send simultaneously. Using case analysis, the expression arg foo: arg is type-checked separately as <*ClassA*> foo: <*ClassA*> and <*ClassB*> foo: <*ClassB*>. More importantly, the case analysis described here allows statements of the following form to be type-checked.

temp isClassA ifTrue: [temp msg1] ifFalse: [temp msg2].

Because Smalltalk is an untyped language, it is common to write code that performs explicit class checking before sending a message to an object. The TS type system type-checks the above statement automatically. Suppose the variable temp has a union type <*ClassA + ClassB*> where ClassA and ClassB have the specification listed in Table 2.4. Consider the case where temp has type <*ClassA*>. After type-checking temp isClassA, temp msg1 and temp msg2, the type-checker proceeds to type-check the following message send:

<*True*> ifTrue: <*Block of: ClassA*> ifFalse: <*IllegalBlock*>

ClassA	ClassB
isClassA ↑<*True*>	isClassA ↑<*False*>
$msg_1 \uparrow < ClassA >$	Does not implement msg_1.
Does not implement msg_2.	$msg_2 \uparrow < ClassB >$

Table 2.4: Specification of ClassA and ClassB

```
{ arguments: arg <SmallInteger + LargePositiveInteger + LargeNegativeInteger>
  temporaries: temp <SmallInteger + LargePositiveInteger + LargeNegativeInteger>
  returnType: <SmallInteger + LargePositiveInteger + LargeNegativeInteger> }

smallestPowerOfTwoGreaterThan: arg
    | temp |
    temp ← 1.
    [temp < arg] whileTrue: [temp ← temp + temp]
    ↑temp
```

Figure 2.6: Another example for case analysis.

Since the second argument of True>>ifTrue:ifFalse: has type <*Anything*>, <*IllegalBlock*> is essentially ignored. As a result, the above message send type-checks. Similarly, using the definition for False>>ifTrue:ifFalse:, we are also able to type-check the case where temp has type <*ClassB*>.

On the other hand, if we have the Smalltalk expression

```
(x < y) ifTrue: [...] ifFalse: [...]
```

where (x < y) returns <*True+False*> then type-checking will fail if one of the arguments of ifTrue:ifFalse is an <*IllegalBlock*>. This is because it is impossible to determine conditions such as (x < y) at compile-time.

Case analysis may also fail when the variable being analyzed is re-assigned to another type in the same expression. For example, consider the method in Figure 2.6. Since temp has a union type and occurs in the receiver and the arguments of whileTrue:, case analysis is performed on temp in the expression

$$\text{[temp} < \text{arg] whileTrue: [temp} \leftarrow \text{temp} + \text{temp]}$$

Consequently, type checking will fail at the assignment temp ←temp + temp . To see this, suppose that temp has type <*SmallInteger*> during case analysis. Type-checking the expression temp + temp will produce the return type <*SmallInteger + LargePositiveInteger + LargeNegativeInteger*>. Then, to type-check the assignment, temp ←temp + temp, the type of temp must include the return type of temp + temp (i.e. <*SmallInteger*> must include <*SmallInteger + LargePositiveInteger + LargeNegativeInteger*>), which is not satisfied.

For this reason, case analysis is not performed on assigned variables and type-checking continues as usual. By taking this conservative approach, statements such as

$$\text{temp isNil ifTrue: [temp} \leftarrow \ldots\text{] ifFalse: [temp} \leftarrow \ldots\text{]}$$

may fail. In which case, the programmer may rewrite the statement to be

$$\text{temp} \leftarrow \text{temp isNil ifTrue: [...] ifFalse: [...].}$$

This allows case analysis to occur on the right hand side of the assignment if necessary. A message send that requires case analysis and contains assignments to the same variable is rare. In cases where it does appear, the type system forces the programmer to make minor changes to the code (using intermediate variables). With the above case analysis rules, the type system can handle most situations that require case analysis.

2.3 Type checking cascaded message sends

A cascaded message send is an extension to the usual message send construct. To type-check a cascaded expression such as x foo; far: y1; fee: y2, the type of x is determined before verifying the following type expressions:

- $<x>$ foo.
- $<x>$ far: y_1.

- $<x>$ fee: y_2.

This is because x itself may be a message send such as (y bar) foo; far: y1; fee: y2 . For case analysis purposes, each cascaded message is treated as a collection of simple message sends. Thus, case analysis may occur in part but not all of a cascaded message send.

2.4 Type-checking blocks

Since a block may be executed zero or more times, a type error in a block may not be fatal as shown in the case analysis examples above. If a block appears on the right hand side of an assignment statement, then the type of the temporary variable or instance variable must include the type of the block. Thus, the block must be type-correct even though it may not be used after assigned to a variable. If a block occurs as the receiver of a message send, then it must be type-correct for its context method to be type-correct. If the block occurs as one of the arguments of a message send, then the block may not be evaluated at all. In this case, the type system relies on the declared type of the called method (e.g. True>>ifTrue:) to type-check the block.

2.5 Type-checking polymorphic methods

Many methods in the standard Smalltalk image are polymorphic. For instance, the method SmallInteger>>+ can take any arithmetic object such as $<\mathit{SmallInteger}>$ and returns another arithmetic object. Therefore, its type can be written as:

$$<\mathit{SmallInteger}> + <\mathit{SmallInteger} + \mathit{Float} + \ldots> \uparrow <\mathit{SmallInteger} + \mathit{Float} + \ldots>$$

However, by defining the type of SmallInteger>>+ this way, the dependency between each of the possible argument types and its return type is lost. That is, the above definition cannot describe the fact that $<\mathit{SmallInteger}> + <\mathit{SmallInteger}>$ evaluates to $<\mathit{SmallInteger} + \mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$ whereas $<\mathit{SmallInteger}> + <\mathit{Float}>$ evaluates to $<\mathit{Float}>$. Note that this problem also exists in double dispatching methods. In fact,

Argument types	Return types
$<\mathit{SmallInteger}>$	$<\mathit{SmallInteger} + \mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$
$<\mathit{LargeNegativeInteger}>$	$<\mathit{LargeNegativeInteger} + \mathit{SmallInteger}>$
$<\mathit{LargePositiveInteger}>$	$<\mathit{LargePositiveInteger}>$
$<\mathit{Float}>$	$<\mathit{Float}>$
⋮	⋮

Table 2.5: The type of SmallInteger>>+.

SmallInteger>>+ can be written using double-dispatching technique as follow:

```
+ aNumber
    ↑aNumber sumFromSmallInteger: self
```

Depending on how sumFromSmallInteger: is implemented in the argument's class, the return type of SmallInteger>>+ can be very different from one argument type to another. In this case, the type of SmallInteger>>+ is described by Table 2.5. The ability to infer a specific return type given the argument type(s) of a message send is useful for optimization. In fact, the type of a method can be viewed as a table relating argument types to return types. The return type of a message send can be determined by looking up the argument types of the message send in this table.

3 Type Inference

The type system described in this thesis is based on the theory developed by Graver [Gra89]. A complete and useful type inference system should perform the following two tasks when type-checking a method:

- Infer the types of arguments and temporaries in a method when the user does not declare them
- Infer a specific return type for each of the method's possible argument types.

Due to time constraints, this thesis focuses on the second task. As in Graver's system, the programmer is required to declare the types of instance variables and class variables. Unlike Graver's system, the user needs to declare (at least) the argument types of a method. The types of the temporary variables can be inferred by the type system from the assignment statements. The types of block arguments can be inferred from messages sent to the block or messages with the block as its argument.

If the argument types of a method are not declared then the type system produces a constraint that can be used as the type of the method. A constraint is an abstraction of the method that indicates the messages that it sends to its arguments and how it computes its return value. Type inference builds a tree that mirrors a method and reduces it to determine the return type of the method. The tree can be completely reduced unless some of the argument types are not declared, in which case it is only partly reduced. The partly reduced tree becomes the constraint for the method.

Constraints let the type system defer type-checking a method until the point that it is called. Given a particular set of argument types, the type system can infer the return type of any method. Since a constraint is a method reduced as far as possible without knowing the argument types, the type system can infer the return type of a call on a method with a constraint by applying the argument types to the constraint. This is equivalent to in-lining the called method and type-checking it at the point of call, except that the type-system can handle recursion.

```
foo: arg
        arg view.
        arg + 1
```

Figure 3.1: A naive method that will cause a type error.

A method's constraint is not completely satisfactory as its type. One problem is that a constraint might not be satisfiable, i.e. there might be a type error in the method that the constraint will hide and that will only be shown when callers of the method are type-checked. A naive example is given in Figure 3.1. Note that no single class in the Smalltalk image understands both the messages, view and +. However, if the user does not declare the type of arg then all the message sends in foo: cannot be type-checked and the method foo: will have a constraint as its type. Later when foo: is type-checked at the call site with an actual argument, at least one of the messages, arg view or arg + 1, will cause a type error. Another problem is that a constraint does not provide any information for optimizing its method, though it will help in optimizing the callers of the its method.

On the other hand, there are methods whose only type is a constraint [Gra89]. Moreover, polymorphic methods whose arguments must be described by signatures cannot be improved by optimization, so constraints are just as precise a description of a method's type as the declared types. Constraints provide little information to the programmer, so signature types make better documentation, but constraints are just as good for code optimization.

3.1 Forward and backward type inference

If the argument types of a method are known, then the type system may verify all the expressions in a method using the type rules described in Chapter 2. In doing so, type information from the current method is propagated to the callees until all the message sends are checked. This is data-flow analysis because, like program data, type information flows from the arguments to the return type of a method, from the caller to its callees. This is called forward type inference.

Type information may also propagate backwards from the callee to its caller. The type system can infer the type of any method by examining the declared types of its callees. Intuitively, if the declared type of True>>or: is <*'or: <BooleanType>* $\uparrow$*<BooleanType>'*> then the type variable <*T1*> in the expression <*True*> or: <*T1*> may take a type no 'bigger' than <*BooleanType*>. We call this kind of information flow a requirement-flow because the type variable <*T1*> is constrained by <*BooleanType*>. This kind of type inference is called backward type inference.

Forward type inference allows the type system to infer the return type of a message given the argument types. Backward type inference enables the type system to infer the types of arguments and temporaries when the user does not declare them. The latter should be thought of as a preprocessing phase where the argument types of a method are inferred. Forward type inference is then used to infer the method's return type from the argument types. Since the current type system requires the user to declare the argument types of a method, this thesis focuses on forward type inference. Backward type inference is only used to infer the types of the block arguments when the programmer did not declare them.

To support the bi-directional information flow, a type variable is said to have two bounds: an upper bound and a lower bound. An unbounded type variable has <*Anything*> as its upper bound and <*Nothing*> as its lower bound. The upper bound of a type variable is its requirement-flow information whereas the lower bound is its data-flow information. When a type variable s includes another type variable t, data-flow information flows from t to s. At the same time, requirement-flow information flows from s to t. The rest of this section describes the upper bounds and the lower bounds in greater detail.

Consider an assignment statement s $\leftarrow$ t. The type inclusion rule of assignment says that the type of s must include the type of t. Let s be the type of s and t be the type of t. There are four cases to consider:

- Both s and t are not type variables — t is a subtype of s.
- s is a type variable, t is not — t is added to the lower bound of s. If s is unbounded before the inclusion test, then the lower bound of s becomes t.
- s is not a type variable but t is — s is added to the upper bound of t. If t is unbounded before the inclusion test, then the upper bound of t is s.
- Both s and t are type variables — the upper bound of s is added to the upper bound of t, and the lower bound of t is added to the lower bound of s.

The first case amounts to a simple check for subtyping. The last three cases involve type variables and form the basis of (forward and backward) type inference. When a type variable s includes another type variable t (as in $\mathsf{s} \leftarrow \mathsf{t}$), the lower bound of s must be at least as big as the lower bound of t (the lower bound of s can be greater because s may include other type variables). Data-flow information is propagated from t to s by unioning the lower bound of t with the lower bound of s and saving the union into the lower bound of s. This guarantees that s is no smaller than t. Very often, the lower bound of a type variable is assumed to be the value of the type variable. That is, if s has a lower bound of <*SmallInteger*>, then s is assumed to be a <*SmallInteger*>. When the lower bound of s gets 'bigger' from more inclusion tests, type-checking has to be repeated.

While data-flow information propagates from t to s, requirement-flow information propagates from s to t. If t is included in s, all the restrictions placed on s must also apply to t. For this reason, the upper bound of s is intersected with and added to the upper bound of t. By doing so, t is bounded by the combined requirements. The upper bounds of type variables are similar to the subscript bounds of an array: at no time should they be violated. Sometimes, only the requirement-flow but not the data-flow information of a type variable is known. In this case, the type variable is said to have undefined type even though it has a finite upper bound.

Forward type inference deals with data-flow information, and backward type inference deals with requirement-flow information. Much of this chapter deals with forward type inference. The next section will talk about backward type inference for type-checking blocks. Subsequently, only forward type inference will be mentioned.

3.2 Backward type inference

Consider the program in Figure 3.2. If the user does not declare the type of child, then the block [:child | child foo] cannot be type-checked. Consequently, the type of [:child | child foo] is <*Block of: T1 returns: T2*> where <*T1*> and <*T2*> are unbounded type variables. Next, the following message send is type-checked.

```
self updateChildrenUsing: [ :child | child foo].
```

```
foo
        self updateChildrenUsing: [ :child | child foo ]
```

Figure 3.2: Infer block argument types.

Suppose that the declared argument type of updateChildrenUsing: is

$$< Block\ of\colon (ClassA + ClassB)\ returns\colon (ClassC + ClassD) > .$$

According to the message send type inclusion rule, the declared argument types of the called method must include the argument types of the message send. In this case, <*Block of: (ClassA + ClassB) returns: (ClassC + ClassD)* > must include <*Block of: T1 returns: T2*>. By the contravariance rule of Block type subtyping, <*T1*> must include <*ClassA+ClassB*>. That is, <*T1*> has a lower bound of <*ClassA+ClassB*>. As a result, the block [:child | child foo] is retype-checked using <*ClassA+ClassB*> as the type of child. Then, the return type of child foo must be included in <*ClassC+ClassD*>. Since no type inclusion error occurs, the type system concludes that the the method foo is type-correct. Because type information flows from the callee of a message send to its caller, the above process is backward type inference. Using backward type inference, the type system has inferred the type of the block argument to be <*ClassA + ClassB*>.

If the user also did not declare the argument type of updateChildrenUsing: then updateChildrenUsing: must be retype-checked at the call site, using <*Block of: T1 returns: T2*> as its argument type. In that case, the callees of updateChildrenUsing: may be used to infer the type of child in [:child | child foo] (the same way updateChildrenUsing: was able to type-check the block if the user declared its argument type). Furthermore, a valuation message such as value: x, may be sent to [:child | child foo]. The block is then type-checked using <*x*> as the type of child. If all the above approaches fail, then the type system will conclude that more information is required to type-check the block, and will defer type-checking the current method until it is called by another method.

```
smallestPowerOfTwoGreaterThan: arg
        | temp |
        temp ← 1.
        [temp < arg] whileTrue: [temp ← temp + temp]
        ↑temp
```

Figure 3.3: Type inference of assignments.

3.3 Forward type inference

To recap, forward type inference is a data-flow technique. To infer the type of an assignment, the type of its right-hand-side expression must be known. To infer the type of a message send, the argument types must be known. Forward type-inference is an iterative process.

3.3.1 Iteration due to assignment statements

Consider the method in Figure 3.3. If the user does not declare the type of temp, then from the statement temp ← 1 it can be inferred that the type of temp is at least a <*SmallInteger*>. Subsequently, the expression, [temp < arg] whileTrue: [temp ← temp + temp], is type-checked based on the assumption that the type of temp is <*SmallInteger*>. However, this assumption is broken after type-checking the assignment temp ← temp + temp. As a result, type-checking is repeated using the new assumed type of temp. When the type of temp remains unchanged for two consecutive iteration, the type of temp has been computed. Thus, the type of a temporary variable can be inferred by the type system using forward type inference.

3.3.2 Iteration due to message sends

Recursive message sends may also cause the type system to iterate. Consider the recursive method in Figure 3.4. If the type of arg is unknown, then the message send (arg < instVar) cannot be type-checked and the constraints flag in the method type declaration will be set to true. On the other hand, if the programmer declares the type of arg to be <*SmallInteger + LargePositiveInteger + LargeNegativeInteger*>, then the above method is type-checked for every argument type <*SmallInteger*>, <*LargePositiveInteger*>, and <*LargeNegativeInteger*>. In the process, a table mapping these argument types to their corresponding return types is

```
recursiveMethod: arg
        ↑(arg < instVar)
                ifTrue: [self recursiveMethod: (arg + arg)]
                ifFalse: [arg]
```

Figure 3.4: Type inference of recursive method.

constructed. This table reduces redundant work when the return type of the current method is needed to type-check another method. The following paragraphs assume that such a table already exists for $<$ and $+$.

First, suppose that arg is a <*LargePositiveInteger*>. The type of the expression arg + arg is, therefore, <*LargePositiveInteger*>. A recursive message send,

self recursiveMethod: <*LargePositiveInteger*>

has occurred. Since the type of the above recursive message send is unknown, it is assumed to be <*Nothing*>. When the entire method has been type-checked, it can be inferred that self recursiveMethod: <*LargePositiveInteger*> returns a <*LargePositiveInteger*> (if arg $\not<$ instVar). Hence, type-checking is repeated, assuming that

self recursiveMethod: <*LargePositiveInteger*>

returns a <*LargePositiveInteger*>. When this assumption remains unchanged for two consecutive rounds, the type checker concludes that recursiveMethod: takes a <*LargePositiveInteger*> and returns a <*LargePositiveInteger*>.

A similar process happens for the case where arg is a <*LargeNegativeInteger*>. The type checker will conclude that recursiveMethod: takes a <*LargeNegativeInteger*> and returns a <*LargeNegativeInteger*> (even though the program may go into an infinite loop).

Finally, consider the case where arg is a <*SmallInteger*>. This is more complicated than the previous two cases because arg + arg now returns a union type, <*SmallInteger + LargePos-*

$\mathit{itiveInteger} + \mathit{LargeNegativeInteger}>$. Here again, the following recursive call is encountered.

self recursiveMethod: $<\mathit{SmallInteger} + \mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$.

From above, self recursiveMethod: $<\mathit{LargePositiveInteger}>$ returns a $<\mathit{LargePositiveInteger}>$ and self recursiveMethod: $<\mathit{LargeNegativeInteger}>$ returns a $<\mathit{LargeNegativeInteger}>$ whereas by assumption, self recursiveMethod: $<\mathit{SmallInteger}>$ returns $<\mathit{Nothing}>$. Combining these results together, the type system infers that the return type of

self recursiveMethod: $<\mathit{SmallInteger} + \mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$

is $<\mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$. As a result, the method's return type is $<\mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$ when arg $<$ instVar. In addition, the method also returns a $<\mathit{SmallInteger}>$ when arg $\not<$ instVar. The combined return type of

self recursiveMethod: $<\mathit{SmallInteger}>$

is therefore $<\mathit{SmallInteger} + \mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$. So, the assumption that

self recursiveMethod: $<\mathit{SmallInteger}>$

returns $<\mathit{Nothing}>$ is incorrect, causing the type system to perform one more iteration using the new assumption. When the assumption remains unchanged, the type of applying recursiveMethod: to $<\mathit{SmallInteger}>$ is computed to be $<\mathit{LargePositiveInteger} + \mathit{SmallInteger} + \mathit{LargeNegativeInteger}>$. To summarize the whole process, the type of recursiveMethod: is:

- recursiveMethod: $<\mathit{SmallInteger}> \uparrow <\mathit{LargePositiveInteger} + \mathit{SmallInteger} + \mathit{LargeNegativeInteger}>$
- recursiveMethod: $<\mathit{LargePositiveInteger}> \uparrow <\mathit{LargePositiveInteger}>$
- recursiveMethod: $<\mathit{LargeNegativeInteger}> \uparrow <\mathit{LargeNegativeInteger}>$

In the above example, the case where arg is a $<SmallInteger>$ is computed last. As a result, type inference is simplified. If the order is reversed, the type of

$$\textsf{self recursiveMethod:}\ <LargePositiveInteger>$$

and

$$\textsf{self recursiveMethod:}\ <LargeNegativeInteger>$$

must be inferred from scratch when computing the type of

$$\textsf{self recursiveMethod:}\ <SmallInteger + LargePositiveInteger + LargeNegativeInteger>\ .$$

In any case, because there is a finite number of types and messages in the image, the above iterative process will always terminate.

3.4 Type-inference using signature types

Signature types can be used to represent the type of a method. For example, the type of SmallInteger>>+ in Figure 3.5 can be written as

$$<' + <'\ sumFromSmallInteger :< Self > \uparrow < T1 >'> \uparrow < T1 >'>$$

That is, SmallInteger>>+ can take any argument that understands the interface of sumFromSmallInteger: and returns the result of applying sumFromSmallInteger: to $<Self>$. In fact, by examining all the messages sent to an argument, the type system can always infer a signature type to describe the argument. For instance, since the argument aNumber in Figure 3.5 receives the message sumFromSmallInteger:, it has the signature type $<$*'sumFromSmallInteger: Self* $\uparrow$$<$*T1*$>$*'*$>$

Unfortunately, using signature types in this way poses two problems. First, signature types cannot handle the case analysis described in the previous chapter. Furthermore, to describe the type of a method using signature types, additional type information from the called methods may be required. As an example, consider the method in Figure 3.6. To describe the type of lo and hi adequately, the signature types must also determine what messages are sent to the

```
+ aNumber
    ↑aNumber sumFromSmallInteger: self
```

Figure 3.5: SmallInteger>>+.

```
between: lo and: hi
    ↑(self > lo) and: [self < hi]
```

Figure 3.6: Signature types need to store type information of the callees.

arguments lo and hi in the callees, > and <. In the worst case, type information in the entire call tree of between:and: must be considered and the resulting signature types may take up lots of memory. Thus, signature types are never inferred by the type system to describe the type of a method. Instead, when its argument types are undeclared, the partially type-checked method is given a constraint as its type. When the return type of an incompletely type-checked method is required to type-check another method, its constraint is type-checked to give a return type. In this sense, signature types are subsumed by constraints.

While signature types are not suitable for describing the type of a method, they should be suitable for describing subtype relationships between object types. For instance, <*Float*> can be the supertype of <*SmallInteger*> based on the interfaces of their methods. As a result, a type hierarchy (different from the class hierarchy) based on signatures could be built to offer more abstraction and encapulation. The current TS type system does not have a type hierarchy though this is planned for the near future. Since one of the goals of this thesis is to document the TS type system, signature types, which are supported to some extent, are also described in the implementation chapter.

4 Implementation

The implementation of the type system consists of three kinds of classes. First, there are classes for objects that represent types. Most of these are subclasses of Type. Second, there are classes of objects that represent the parse tree during type-checking. Type-checking is done on a simplified version of the parse tree called the type tree. The nodes of the type tree consist of instances of subclasses of SPNode. The rest of the type system are stored in category TS-TypeSystem and help represent and solve systems of inclusion relations between type variables. The most important of these classes is TypeEquationSolver, which represents the inclusion relationships between type variables with a directed graph, where the nodes of the graph are type variables and an edge from u to v means that $u \sqsubseteq v$. The TypeEquationSolver is embedded in a TypeEnvironments, which is passed around during type inference.

Type inference builds a type tree and traverses it in postorder. For each node encountered, the type inclusion rules described in Chapter 2 are used to type-check the expression associated with the node. Type variables present in the expression types result in inclusion relations between the type variables. For instance, the assignment `x` $\leftarrow$ `y` requires the type of `x` to include the type of `y`. If either `x` or `y` has a type variable in its type then the inclusion relations are updated to reflect the inclusion relationship between the type variables. If the new type information is inconsistent with the previous information then a type error has occured. If the type of `x` and `y` are not type variables then the simple subtyping rules in Chapter 2 are used to verify type inclusion.

When a message node is encountered, the type system needs to identify the type of the receiver so that the correct method's declared type can be obtained to type-check the message send. If the receiver type is a type variable then its lower bound is assumed to be its type. If the type variable is free then the message send cannot be type-checked. As a result, type-checking of the current method is deferred (i.e. a constraint is produced).

After type-checking all the expressions in a method, the type system checks if any of the previous assumptions have changed. If so, another iteration is performed. After type-checking,

MethodType
TypedMessage
    TypedSuperMessage
Type
    BasicType
        ObjectType
        BlockType
    PseudoType
        SelfType
        SelfClassType
        SelfMetaType
    SignatureType
    TypeVariable
    UnionType
    UniqueType
        Anything
        MySelfInstanceType
        MySelfMetaType
        MySelfType
        Nothing

Figure 4.1: The class hierarchy of Type

if the argument types are declared then a table of argument types to return type is computed and stored.

The next section will describe the classes that implement types. Section 4.2 will describe type trees. Section 4.3 will describe the classes that maintain type information during type inference.

4.1 Types

The classes in the TS-Type category are listed in Figure 4.1. The MethodType class implements the table of argument types to return types. Since MethodType is defined on top of other classes in TS-Types, it is described at the end of this section. Every type described in Chapter 1 is implemented by a subclass of class Type. The main operations of these classes are: intersect:in:, union:in:, include:in:, matches:in:, and simulateMessageSend:in:. They are collected in the type operations protocol.

The above operations take a TypeEnvironment as their second argument. The intersect:in: method adds requirement-flow information to the upper bound of a type variable. The union:in: method adds data-flow information to the lower bound of a type variable. The include:in: method asserts that the receiver includes the first argument. The matches:in: asserts that the receiver is equivalent (two-way inclusion) to the first argument. The simulateMessageSend:in: method allows the receiver type of a message send to type-check the message send. With the exception of matches:in:, these operations are defined in the Table 4.1, Table 4.2, Table 4.3, and Table 4.4. A table for matches:in: is not given because it is defined as a two-way inclusion. Since the union and intersection operations are commutative, only half of their tables are defined. The empty portion can be filled in by doing a mirror image along the main diagonal of the table.

4.1.1 Anything and Nothing

By definition, <*Anything*> includes all the possible types in TS. It does not understand any typed messages. On the contrary, <*Nothing*> understands all the typed messages and is included by all the types. <*Anything*> can be the type of an argument that is not used by the context method. <*Nothing*> can be the return type of a method that never returns to the caller.

4.1.2 BasicType

ObjectType implements both parameterized and non-parameterized object types. It has an instance variable, argList, that contains a possibly empty list of types. BlockType differs from ObjectType in that it has a return type. BasicType is created to capture the common implementation of both BlockType and ObjectType. A BasicType remembers its associated class by its name. For instance, the associated class of <*SmallInteger*> is SmallInteger, the associated class of <*Dictionary of: KeyType of: ValueType*> is Dictionary.

A BasicType type-checks a message send by looking up the appropriate declared type based on its class name and the message selector. As an example, 'sending' a typed message <*'+ <T1>* ↑*<T2>'*> to <*SmallInteger*> will cause the latter to lookup the method type for SmallInteger>>+.

	BasicType	UnionType	SignatureType	Anything	Nothing	PseudoType
BasicType	Form a union type of the two basic types.	Add the basic type to the union type.	If the signa-type includes the basic type, then answer the sign-ature type. Otherwise, union both types together.	Answer Anything.	Answer the basic type.	Union the basic type with the actual type of the pseudo-type.
UnionType		Combine the two union types together to form a new one.	If the signa-type includes the union type, then answer the sign-ature type. Otherwise, union the types which are not included by the signature type with the signature type.	Answer Anything.	Answer the basic type.	Union the union type with the actual type of the pseudo-type.
SignatureType			Form a union type of the two signature types.	Answer Anything.	Answer the signature type.	Union the signature type with the of the pseudo-type.
Anything				Answer Anything.	Answer Anything.	Answer Anything.
Nothing					Answer Nothing.	Answer the pseudo-type.
PseudoTypes						Union the actual type of the pseudo-types.

Table 4.1: Union type operations

	BasicType	UnionType	SignatureType	Anything	Nothing	PseudoType
BasicType	If the basic types do not have the same name, then answer Nothing else answer the basic type.	Intersect the basic type with the elements of the union type, and returns the union of each of the intersection.	Check whether the basic type understand every typed messages in the signature type. Answer the basic type.	Answer the basic type.	Answer Nothing.	Intersect the union type with the actual type of the pseudo-type.
UnionType		Answer the basic types present in both the union types.	Check whether all the basic types in the union type understand every typed messages in the signature type. Answer the union type.	Answer the union type.	Answer Nothing.	Intersect the union type with the actual type of the pseudo-type.
SignatureType			Concatenate the typed messages in both the signature types.	Answer the signature type.	Answer Nothing.	Intersect the signature type with the actual type of the pseudo-type.
Anything				Answer Anything.	Answer Nothing.	Answer Nothing.
Nothing					Answer Nothing.	Answer Nothing.
PseudoTypes						Intersect the actual type of the pseudo-types.

Table 4.2: Intersect type operations

	BasicType	UnionType	SignatureType	Anything	Nothing	PseudoType
BasicType	Use the definition of basic type subtyping to check for inclusion.	Type error.	Type error.	Type error.	Answer the basic type.	Answer the result of the basic type including the actual type.
UnionType	Check whether the basic type is in the union type.	Check if the argument union type is a subset of the receiver union type.	Type error.	Type error.	Answer the union type.	Answer the result of the union type including the actual type.
SignatureType	Check whether the basic type understand all the messages in the signature type.	Check whether all the elements in the union type understands all the messages in the signature type.	Check whether every messages in the receiver sigature type is in the argument signature type.	Type error.	Answer the signature type.	Answer the result of the signature type including the actual type.
Anything	Answer Anything.	Answer Anything.	Answer Anything.	Answer Anything.	Answer Anything.	Answer Anything.
Nothing	Type error.	Type error.	Type error.	Type error.	Answer Nothing.	Type error.
PseudoTypes	Answer the result of the actual type including the basic type.	Type error.	Type error.	Type error.	Answer the pseudotype.	Answer the result of the actual type of the receiver including the actual type of the argument.

Table 4.3: Inclusion type operations

Receiver type of message send	Action
BasicType	Lookup the declared method type of the called method and apply the message send type inclusion rule.
PseudoType	Delegate type-checking to the actual type of the pseudo-type (which is a BasicType).
UnionType	Send simulateMessageSend:in: to individual types in the union type.
SignatureType	Select the appropriate typed message from the signature type and type-check the actual message send using the selected typed message. If the signature type does not understand the actual message send, then a type error has occured.
Anything	Type error. <*Anything*> does not understand any typed messages.
Nothing	Does nothing. <*Nothing*> understands all typed messages.

Table 4.4: simulateMessageSend:in:.

```
simulateMessageSend: aMsgType in: sendEnv
    | formalMethodType |
    (formalMethodType ← aMsgType declaredTypeForReceiver: self) isNil
    ifTrue: [self abort: typeGenerator, '>>' aMsgType selector, ' is untyped or undefined'].
    ↑formalMethodType
        typeCheckAgainst: (aMsgType receiverType: self)
        in: sendEnv
        using: [ :type | type removePseudoTypes ].
```

Figure 4.2: BasicType>>simulateMessageSend:in:.

```
simulateMessageSend: aMsgType in: sendEnv
    ↑aMsgType isValuationMessage
        ifTrue: [self evaluateBlockUsing: aMsgType in: sendEnv]
        ifFalse: [super simulateMessageSend: aMsgType in: sendEnv]
```

Figure 4.3: BlockType>>simulateMessageSend:in:.

In the actual code (see Figure 4.2), method type lookup is implemented by declaredTypeForReceiver of class TypedMessage and TypedSuperMessage. The former looks up the method type starting from the class of the receiver type. The latter looks up the method type starting from the superclass of the receiver class. When the formal method type is undefined, an abort error is signalled. Otherwise, it is used to type-check against the actual message send. The second argument sendEnv is the caller's TypeEnvironment of the message send. The block, [:type | type removePseudoTypes], is given to the declared method type to convert Self, Self class, and Self new to the actual class of the receiver, its metaclass, and its instance class respectively.

4.1.3 BlockType

Due to unknown argument types, a block may be type-checked completely. In that case, the partially type-checked block is stored in the constraints instance variable of its associated BlockType. Like constraints for methods, this block will be retype-checked later given the types of its arguments. Hence, the incompletely type-checked block can also be treated as a constraint. When a BlockType with a constraint is included by another BlockType, the latter's argument types are used to type-check the former's constraint. Similarly, when a BlockType with a constraint receives a valuation message, the constraint is type-checked by adding the argument type(s) of the valuation message to the constraint's TypeEnvironment. This is different from a regular message send where the actual argument type(s) are added to the called method's TypeEnvironment for type-checking.

In the actual code (see Figure 4.3), no method look up is simulated for valuation messages. Instead, when a valuation message is sent to a BlockType, evaluateBlockUsing:in: adds the argument types of the valuation message to its constraint's TypeEnvironment for type-checking.

```
simulateMessageSend: aMsgType in: sendEnv
    | formalMethodType |
    (formalMethodType ← aMsgType declaredTypeForReceiver: actualType) isNil
        ifTrue: [self abort: typeGenerator, '>>',
            aMsgType selector, ' is untyped or undefined'].
    ↑methodType
        typeCheckAgainst: (aMsgType receiverType: actualType)
        in: sendEnv
        using: [ :type | type convertPseudoTypeWithRespectTo: self ].
```

Figure 4.4: PseudoType>>simulateMessageSend:in:.

message type-checks a block or constraint in After the constraint is type-checked, its return type is included by aMsgType's return type in sendEnv.

4.1.4 PseudoType

PseudoType implements the types <*Self*>, <*Self new*> and <*Self class*>. Since a PseudoType always delegates type operations to the BasicType it represents, it stores the associated BasicType in the instance variable actualType. A PseudoType behaves just like the BasicType it represents except that a PseudoType also knows how to convert itself in a different class. Before type-checking a message send, the pseudo-types in the callee's method type must be converted to the appropriate types in the caller's environment. The receiver type of the message send determines how the pseudo-types should be converted. The rules for performing the conversion are listed in Table 2.3. The actual code for handling a message sent to a PseudoType is shown in Figure 4.4.

The main difference between PseudoType>>simulateMessageSend:in: and BasicType>>simulateMessageSend:in: is the block [:type | type convertPseudoTypeWithRespectTo: self], which performs the pseudo-type conversion. After converting the pseudo types, the type system will type-check the message send using the type inclusion rules for type-checking message sends, which brings us to TypedMessage.

```
typeCheckMessageSendIn: sendEnv
    argumentTypes do: [ :argType | argType isTypeVariable ifTrue:
        [sendEnv constraints: true.
        ↑self]].
    receiverType simulateMessageSend: self in: sendEnv.
```

Figure 4.5: TypedMessage>>typeCheckMessageSendIn:.

4.1.5 TypedMessage and TypedSuperMsg

A TypedMessage is created whenever a message send is type-checked. The Smalltalk expression: x msg: y is type-checked by creating the TypedMessage <*'<x> msg: <y>'*> and 'sending it' to the receiver type <*x*> (see Figure 4.5). BasicType, PseudoType, UnionType, Signature types, Anything, and Nothing all know how to type-check a message send. However, only the first two classes need to look up the declared method type to type-check the message send. In the actual code, if one of the argument types of a message send is unknown then forward type inference fails. In this case, the caller is said to have a constraint. Otherwise, the message send is type-checked as described above.

Recall that when a method is completely type-checked, it is represented as a table of argument types to return types. This table is implemented as a collection of TypedMessages. So, TypedMesssage also understands the message, selectReturnTypeBasedOn:in:, which implements the message send type inclusion rule. If the message send inclusion test is successful then selectReturnTypeBasedOn:in: returns true. Otherwise, it returns false.

However, if the called method is not completely type-checked then its return type is inferred given the argument types of the message send. Essentially, the called method is type-checked at the call site. In doing so, the type system may need to type-check a cycle of message sends if one of the called methods is recursive. Since recursive message sends can occur only if a called method needs to be type-checked at the call site, type-checking a recursive call is implemented in the root of the type tree, MethodSPN, which will be described later. None of the classes in TS-Types 'knows' about recursive call.

```
selectReturnTypeBasedOn: actualTypedMsg in: sendEnv
        | env |
        env ← sendEnv typeCopy.
        actualTypedMsg receiverType matches: receiverType in: env.
        argumentTypes with: actualTypedMsg argumentTypes do: [ :formalArg :actualArg |
                formalArg includes: actualArg in: env ifFail: [ :ex | ↑false]].
        actualTypedMsg returnType includes: returnType in: env.
        typeEnv replaceTypeInfoFrom: env.
        ↑true
```

Figure 4.6: TypedMessage>>selectReturnTypeBasedOn:in:.

4.1.6 SignatureType

A SignatureType is a collection of TypedMessages. It represents a set of object types that understand the TypedMessages it contains. Consequently, it knows how to union and intersect with other types. An empty signature type is equivalent to <*Anything*>. The more typed messages in a SignatureType, the smaller the represented set of object types becomes. The types <*MySelfInstanceType*>, <*MySelfMetaType*> and <*MySelfType*> are used to define a SignatureType. <*MySelfType*> refers to the SignatureType being defined, <*MySelfInstanceType*> the instance type of the SignatureType, and <*MySelfMetaType*> the class type of the signature type. Thus we may write a signature type as <*'id ↑<MySelf>; new ↑<MySelfInstance>; class ↑<MySelfClass>'*>. A SignatureType includes a BasicType, if the BasicType 'understands' all the TypedMessages in the SignatureType.

When the receiver type of a message send is a signature type, the appropriate typed message is picked from the signature type. Then the message send type inclusion rule is applied to the the typed message in the signature type and the actual typed message.

4.1.7 TypeVariable

A type variable is a label representing an unknown type. Like TypedMessages, TypeVariables do not understand the union and intersection operations. In fact, these operations are used to add type information to the upper and lower bound of a type variable. When a TypedMessage is 'sent' to a TypeVariable representing an unknown type, the message send cannot be type-

Operations	Actions
$<T1>$ includes: aType	Union the lower bound of $<T1>$ with aType.
aType includes: $<T1>$	Intersect the upper bound of $<T1>$ with aType.
$<T1>$ includes: $<T2>$	Union the lower bound of $<T2>$ with the lower bound of $<T1>$ (save the result in the lower bound of $<T1>$) and intersect the upper bound of $<T1>$ with the upper bound of $<T2>$ (save the result in the upper bound of $<T2>$).

Table 4.5: TypeVariable operations

checked, so the constraints flag of the context method will be set to true. This means that the type of the method will be represented by a constraint. The lower and upper bounds of a TypeVariable are stored in the TypeEnvironment. Whenever an inclusion operation is applied to any type containing a type variable, type information is added to the TypeEnvironment. Table 4.5 defines the inclusion operation for TypeVariables.

4.1.8 MethodType

MethodType describes the type of a completely type-checked method. It is implemented as a collection of TypedMessages. The only important operation on a MethodType is typeCheck-Against:in:using (see Figure 4.7), which is invoked by simulateMessageSend:in:. As usual, the pseudo-types are converted and the type variables are renamed. Then, the appropriate Type-dMessages are chosen from the MethodType based on the argument types of the actual message send. If no suitable TypedMessage can be found, a type error is reported. If a method is not completely type-checked, it is represented by a type tree when called by another method. The following section will describe the type tree.

4.2 Type Trees

The type tree is a simplified version of the parse tree for type-checking purposes. Figure 4.8 shows the classes in TS-TypeTrees. The interesting classes in the SPNode hierarchy are MessageSPN and MethodSPN. MessageSPN implements case analysis and type-checking message sends whereas MethodSPN implements type-checking recursive methods.

```
    typeCheckAgainst: aMsgType in: typeEnv using: aBlock
        self convertPseudoTypesUsing: aBlock.
        self renameTypeVariablesUsing: typeEnv typeVariableGenerator.
        aMsgType forAllArgumentTypes do:
            [ :actualTypedMsg |
            self selectReturnTypeBasedOn: actualTypedMsg in: typeEnv ]


selectReturnTypeBasedOn: actualTypedMsg in: typeEnv using: aBlock
    typedMessages do: [ :formalTypedMsg |
        (formalTypedMsg selectReturnTypeBasedOn: actualTypedMsg in: typeEnv)
            ifTrue: [↑self]].
    self fail: actualTypedMsg printString, 'is an illegal message send'.
```

Figure 4.7: MethodType>>typeCheckAgainst:in:using:.

```
SPNode
    LeafSPN
        LiteralSPN
        VariableSPN
        InstVarSPN
        GlobalVarSPN
    AssignmentSPN
    MessageSPN
    CoercionSPN
    CascadeSPN
    ReturnSPN
    StmtListSPN
    BlockSPN
        MethodSPN
```

Figure 4.8: The class hierarchy of SPNode

```
decorateWithTypesIn: typeEnv
      typeEnv initTypeOf: self to: (typeEnv typeOf: value)


typeCheckIn: typeEnv
      (typeEnv typeOf: variable)
            includes: (typeEnv typeOf: value)
            in: typeEnv
            ifFail: [ :ex | ex parameter location: self. ex reject]
```

Figure 4.9: AssignmentSPN>>decorateWithTypesIn: and AssignmentSPN>>typeCheckIn:.

The most important protocol of SPNode is type checking, which consists of the methods: decorateWithTypesIn:, prepareForTypeCheckingIn:, and typeCheckIn:. The decorateWithTypesIn: method initializes the type of all the nodes in a type tree (see Table 4.6). The typeCheckIn: method implements the type-inclusion rules described in Chapter 2 (see Table 4.7). The prepareForTypeCheckingIn: method implements part of case analysis. All three take a TypeEnvironment as argument. Like the type operations, these three methods can be described easily using a table. The methods AssignmentSPN>>decorateWithTypesIn: and AssignmentSPN>>typeCheckIn: in Figure 4.9 show how Tables 4.6 and 4.7 are translated to Smalltalk code. AssignmentSPN does not implement prepareForTypeCheckingIn:.

4.2.1 MessageSPN

The most important responsibility of MessageSPN is case analysis. Consider the method and its type tree in Figure 4.10. If temp has a union type then case analysis will be performed on the entire subtree rooted at ifTrue:ifFalse:. Note that this decision is made before type-checking any children of ifTrue:ifFalse:. That is, before type-checking temp isClassA, [temp], and [temp bar: arg], the algorithm must check to see if case analysis is required on the temporary variable, temp. Thus, the detection of case analysis has to occur in preorder. Case analysis is performed on a message send by type-checking the subtree rooted at the message send several times, once for each case. Hence, the type system needs to be able to traverse a subtree several times. The method preorderDo:stoppingAt:postorderDo: is created for this purpose. The preBlock is executed in preorder on every node. The stopBlock determines if the traversal should descend

Type of nodes	Expression type
AssignmentSPN	The type of the right hand side expression.
BlockSPN	<*Block of: arg1 of: arg2 ... of: argN*> <*returns: stmtList*> where arg_i is the type of the i^{th} argument, and stmtList is the type of the statement list in the Block.
CascadeSPN	The type of the last message send.
CoercionSPN	The user-specified coercion type is stored in the instance variable, `type`.
GlobalVarSPN	Look up the type of the global variable in the pool dictionaries of the superclasses, and finally the system dictionary.
InstVarSPN	Look up the type of the instance variables
LiteralSPN	Delegate to the appropriate class (eg. SmallInteger, Character, Array, String, ...) to generate a literal type.
MessageSPN	Tell the TypeEnvironment to create a new type variable. The type variable will be bound to the return type of the message send after type-checking the method.
MethodSPN	The method return type, <*T0*>.
PrimitiveSPN	The user-specified primitive return type (stored in the instance variable returnType).
ReturnSPN	<*Nothing*>.
StmtListSPN	The type of the last statement.
SuperMsgSPN	Same as MessageSPN.
VariableSPN (for arguments, temporaries, block arguments)	If the user specified a type, then return the user-defined type. Otherwise, create a type variable.

Table 4.6: Types of type tree nodes

Type of nodes	Action
AssignmentSPN	Check whether the type of the variable includes the type of the right-hand-side expression.
MessageSPN/SuperMsgSPN	Create a TypedMessage. Look up the declared method type. Type-check the formal method type against the actual Typed-Message created.
PrimitiveSPN	Check whether the method return type includes the user-specified primitive return type.
ReturnSPN	Check whether the method return type includes the type of the returned expression.

Table 4.7: Implementing type inclusion rules

```
foo: arg
    | temp |
    temp ← self process: arg.
    temp isClassA ifTrue: [temp] ifFalse: [temp bar: arg].
```

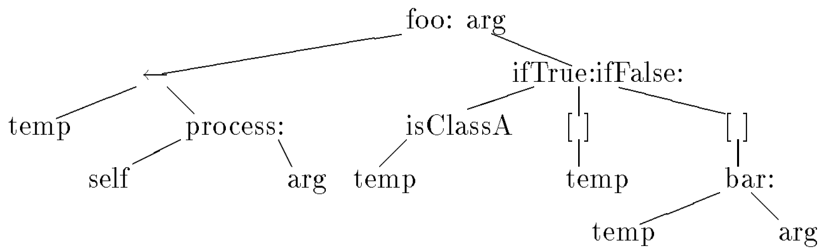

Figure 4.10: Structure of a sample method

```
preorderDo: preBlock stoppingAt: stopBlock updateUsing: postBlock
      preBlock value: self
      (stopBlock value: self) ifFalse: [self updateChildrenUsing: [ :child |
            child preorderDo: preBlock stoppingAt: stopBlock updateUsing: postBlock ]].
      postBlock value: self.

typeCheckTreeIn: typeInfo
      self
            preorderDo: [ :child | child prepareForTypeCheckingIn: typeInfo]
            stoppingAt: [ :child | child stopForCaseAnalysis]
            postorderDo: [ :child | child typeCheckIn: typeInfo ]
```

Figure 4.11: Type-checking a type tree.

to the children of the current node. Finally, when a leaf node is reached, or when the stopBlock prevents the traversal from descending to a lower level, the postBlock is executed.

A method is type-checked by sending the typeCheckTreeIn: message to the root of the type tree (see Figure 4.11). The variable typeInfo is a TypeEnvironment. With the exception of MessageSPN, all the type tree nodes do nothing for prepareForTypeCheckingIn:, and return false for stopForCaseAnalysis.

MessageSPN>>prepareForTypeCheckingIn collects all the variables with union types in both the receiver and the argument subtrees (of the MessageSPN) into its instance variable caseVars. Since the type system does not perform case analysis on assigned variables, it has to remove any variables that are assigned in the message send.

MessageSPN>>stopForCaseAnalysis tests if there are variables on which case analysis can be performed. If this is so, MessageSPN>>typeCheckIn: is executed on the current MessageSPN. Otherwise, the traversal method moves on to the next type tree node.

If case analysis is required on a message send, the variable being analyzed is specialized to one of its basic types and the sub-expressions are type-checked. In rare cases where the type system needs to case analyze several variables at the same time, it generates all the possible combinations of types for the variables involved. For instance, if the variables x *<SmallInteger + UndefinedObject>* and y *<Character + UndefinedObject>* require case analysis then the type system will generate the list { (x *<SmallInteger>*, y *<Character>*), (x *<SmallInteger>*,

```
prepareForTypeCheckingIn: typeInfo
    "set caseVars to the set of variables with a union type
    that are both in the receiver expression and in one of
    the argument expressions."
    | rcvrVars argVars |
    rcvrVars ← Set new.
    receiver postorderDo: [ :child | child addUnionTypeLeafIn: typeEnv to: rcvrVars ].
    argVars ← Set new.
    arguments do: [ :arg |
        arg postorderDo: [ :child | child addUnionTypeLeafIn: typeEnv to: argVars]].
    caseVars ←rcvrVars select: [ :rcvr | argVars includes: rcvr ].

stopForCaseAnalysis
    ↑caseVars isEmpty not

typeCheckIn: typeInfo
    caseVars isEmpty
        ifTrue: [self typeCheckMessageSendIn: typeInfo]
        ifFalse: [(self generateAllPossibleCasesFrom: caseVars) do:
            [ :ext |
            typeInfo extendedby: ext.
            self caseAnalysesMessageSendIn: typeInfo.
            typeInfo removeExtension]].
```

Figure 4.12: Implementation of case analysis.

```
caseAnalysesMessageSendIn: typeEnv
    self childrenDo: [ :child |
        child decorateTreeWithTypesIn: typeEnv.
        child typeCheckTreeIn: typeEnv]
    self typeCheckMessageSendIn: typeEnv
```

Figure 4.13: More code on case analysis.

y <*UndefinedObject*>), (x <*UndefinedObject*>, y <*Character*>), (x <*UndefinedObject*>, y <*UndefinedObject*>) }. Type-checking of the sub-expressions is then repeated for every possible case. Finally, the return type of the message send is the union of all the return types obtained by type-checking the individual cases.

Suppose no case analysis is performed at the current message node. By the time the message node receives the typeCheckIn: message, all its children have already been type-checked (type-checking occurs in postorder). At this point, the type system has already computed the receiver type and the argument type(s) of the message send. So the next step is to type-check the message send.

Now, let's take a closer look at the method caseAnalysesMessageSendIn: (see Figure 4.13). During case analysis, a node may hold contradicting type information. Again, consider the method in Figure 4.10 where case analysis occurs at the message node, ifTrue:ifFalse:. The type of temp isClassA may be <*True*> in one case and <*False*> in another. To prevent type errors due to the contradicting type information, each child of ifTrue:ifFalse: has to be type-checked in a different TypeEnvironment. Since type-checking using several TypeEnvironments is expensive, a different but equivalent approach is used. Before type-checking the children of ifTrue:ifFalse:, decorateTreeWithTypesIn: replaces all the type variables in the children of ifTrue:ifFalse: with new ones. Because different type variables are used, any contradicting type information from the previous case analysis will not affect the current one.

4.2.2 MethodSPN

MethodSPN is the root of the type tree. Its two main responsibilities are type-checking a method and inferring the return type of a message send given the types of its arguments. A method is type-checked by sending a typeCheck message to the MethodSPN. The return type of a called method is inferred by sending typeCheckAgainst:in:using: to the associated MethodSPN. The typeCheck method is also used to implement the typeCheckAgainst:in:using: method.

The typeCheckAtCallSiteAgainst:in:using: method infers the return type of a called method (see Figure 4.14). Before doing so, the TypedMessage associated with the called method is checked to see if a cycle of message sends has occurred. If so, the TypedMessage is said to be recursive and the return type of the called method has to be assumed, instead of inferred, to prevent an infinite loop. Note that if the current TypedMessage has union-type arguments, then it is broken down into several TypedMessages with basic-type arguments. This is mainly because the called method may be recursive for some argument types, but not for other argument types.

The isRecursiveMessageSendIn: method detects a recursive message send by matching the current TypedMessage with a sequence of TypedMessages (called the callChain) which are being type-checked. If a match occurs then the matching TypedMessage in the callChain is returned, indicating that the current TypedMessage is recursive. The main difference between the 'regular' TypedMessage used to type-check a message send and a TypedMessage in the callChain is their return types. The former's return type is unknown until the called method is type-checked. On the other hand, the latter has well-defined return type. Just before type-checking a method (as a caller or callee), a TypedMessage with the return type $<$*Nothing*$>$ is deposited into the callChain. If a second iteration is needed to infer the return type of the method, the corresponding TypedMessage in the callChain is replaced by a new one (with more accurate return type obtained from the first iteration of type-checking). Therefore, when isRecursiveMessageSendIn find a recursive message send, assumedReturnType:using: assumes that the 'regular' TypedMessage has the same return type as the corresponding TypedMessage in the callChain. This computes the least fixed point of the message send's return type.

In typeCheckAtCallSiteAgainst:in:using:, all the pseudo-types are converted to the correct form using aBlock. Then all type variables (in the callee's environment) are renamed. At the same time, the caller's typeVariableGenerator is copied to the callee's TypeEnvironment so that any new type variables generated by the callee will not conflict with the old type variables. In addition, the callee also shares the same callChain with the caller so that recursive messge sends that cross several method boundaries can be detected. The argument types of the message send are then added to the called method's TypeEnvironment to type-check the called method. The

```
MethodSPN>>typeCheckAgainst: aMsgType in: typeEnv using: aBlock
    | assumedType |
    aMsgType forAllArgumentTypes do: [ :typedMsg |
    assumedType ← typedMsg isRecursiveMessageSendIn: typeEnv.
    assumedType isNil
        ifTrue: [self structureCopy
            typeCheckAtCallSiteAgainst: aMsgType in: typeEnv using: aBlock]
        ifFalse: [typeEnv assumeReturnTypeFor: aMsgType
            using: assumedType]].


MethodSPN>>typeCheckAtCallSiteAgainst: aMsgType in: typeEnv using: aBlock
    | retType |
    type convertPseudoTypesUsing: aBlock.
    typeInfo renameTypeVariablesUsing: typeEnv typeVariableGenerator.
    typeInfo typeVariableGenerator: typeEnv typeVariableGenerator.
    typeInfo callChain: typeEnv callChain.

    typeEnv addTypeVariablesFrom: receiverType.
    receiverType matches: aMsgType receiverType in: typeEnv.

    typeInfo addTypeVariablesFrom: aMsgType receiverType.
    aMsgType receiverType matches: self receiverType in: typeInfo.
    self argumentTypes with: aMsgType argumentTypes do:
        [ :formalArg :actualArg |
            typeInfo addTypeVariablesFrom: actualArg.
            formalArg includes: actualArg in: typeInfo].

    self typeCheck
    typeInfo constraints ifTrue: [typeEnv constraints: true].
    typeEnv addTypeVariablesFrom: (retType ←typeInfo evaluateReturnType).
    aMsgType returnType includes: retType in: typeEnv.


TypeEnvironment>>assumeReturnTypeFor: aMsgType using: assumedType
    callChain at: assumedType put: true
    aMsgType returnType includes: assumedType returnType in: self
```

Figure 4.14: MethodSPN>>typeCheckAgainst:in:using:.

```
isRecursiveMessageSendIn: sendEnv
	sendEnv callChain keys reverseDo: [ :assumedType |
		(self isTheSameMessageSendAs: assumedType in: sendEnv) ifTrue:
			[↑assumedType]].
	↑nil
```

Figure 4.15: TypedMessage>>isRecursiveMessageSendIn: sendEnv.

```
typeCheck
	| assumedType env |
	[typeInfo constraints: false.
	typeInfo doAgain: false.
	typeInfo callChain at: (assumedType ← self asApproximatedType) put: false.
	self typeCheckTreeIn: typeInfo.
	typeInfo callChain last ifTrue:
		[env ←typeInfo typeCopy.
		assumedType returnType matches: (self asApproximatedType returnType) in: env
			ifFail: [ :ex | typeInfo doAgain: true]].
	typeInfo callChain removeLast.
	typeInfo doAgain] whileTrue.
```

Figure 4.16: MethodSPN>>typeCheck.

resulting return type is included in the return type of the message send, aMsgType. If the callee cannot be fully type-checked then the constraints flag of the caller is set to 'true'.

The typeCheck method is shown in Figure 4.16. It is the top-level method for type-checking a method (and inferring its return type). In the beginning, a method is assumed to have no constraint, and requires one iteration to type-check. In addition, the asApproximatedType method generates the TypedMessage assumedType to be deposited into the callChain. As mentioned above, the assumedType of a method is used to type-check a recursive message send. If the current method is recursive then at the end of one iteration, its return type is compared with that of the associated TypedMessage in the callChain. If the assumed return type of the recursive method before and after type-checking is different, then another iteration is required.

TypeDeclarationDescriptor
TypeEnvironment
TypeEquationSolver
TypeVariableGenerator
EquivalenceClassBuilder
TypeTable
TypeInclusionNode

Figure 4.17: Classes related to the TypeEnvironment

4.3 Type System

The TS-TypeSystem category consists of classes that make up the type environment. The classes in TS-TypeSystem are listed in Figure 4.17. The type environment serves as a context for type inference and performs the following functions:

- Keeping track of inclusion relationships between type variables
- Storing and printing type declaration data such as class type parameter and local type variables
- Maintaining type information for type trees
- Keeping track of control variables for type checking

A type environment is implemented by the class TypeEnvironment, which encapsulates the following major components:

- TypeDeclarationDescriptor
- TypeVariableGenerator
- TypeTable
- TypeEquationSolver

The TypeEnvironment delegates most of its operations to its components.

4.3.1 TypeDeclarationDescriptor

```
{ classTypeParameters: <KeyType> <ValueType>
  localTypeVariables: T1 <'at: <SmallInteger> ↑ T2'>
  arguments: anObject <T1>
  temporaries: temp <Character>
  blockArguments: each <ValueType>
  returnType: <T2>
  constraints: true
  onlyForTS: false }
```

Figure 4.18: A sample method type declaration.

The TypeDeclarationDescriptor stores all the information declared in the method type declaration. A method type declaration is shown in Figure 4.18. For book-keeping purposes, the TypeDeclarationDescriptor needs to maintain several dictionaries. The typeDeclForTypeVars dictionary maps type variables to their declared upperbounds. The typeVarEncoder maps type variable names to the TypeVariables, preventing the user from declaring duplicated type variables. Finally, typeDeclForVarNodes maps variable nodes to the user-declared types. For the above type declaration, <*T1*> is mapped to the signature type <*'at:* <*SmallInteger*> ↑ <*T2*>*'*> in typeDeclForTypeVars, and typeVarEncoder stores the mapping for <*T1*>, <*T2*>, <*KeyType*>, and <*ValueType*>. Last but not least, typeDeclForVarNodes associates anObject with <*T1*> and each with <*ValueType*>.

The TypeDeclarationDescriptor is created by the compiler to map type variable names to TypeVariable objects. Since signature types can be recursive (eg. T1 <*'+* <*T1*> ↑ <*T2*>*'*>), the parser needs two passes to parse a method type declaration correctly. The first pass collects all the user-declared type variable names into the typeVarEncoder (without looking at their declared ranges), while the second pass parses the ranges of the type variables and the rest of the user-specified type information. When parsing is completed, the TypeDeclarationDescriptor is used to create and initialize the TypeEnvironment. After type-checking, the TypeDeclarationDescriptor will be asked to print a new method type declaration.

4.3.2 TypeVariableGenerator

The TypeVariableGenerator creates new type variables with unique names on request. New type variables are created to represent the type of a message node. The instance variable forbiddenNames is a dictionary of user-specified type variable names. Whenever a new type variable is generated, the TypeVariableGenerator makes sure that the new name does not conflict with the user-declared ones in forbiddenNames. All the type variables generated by the TypeVariableGenerator are of the form 'A#' where A is a pre-selected letter and # is an integer (counter). The counter is incremented every time a new type variable is created. Hence, the type variables generated by the same TypeEnvironment will not conflict with each other.

On the other hand, when type-checking a message send, type variables from the callee's method may conflict with the ones in the caller's TypeEnvironment. The TypeVariableGenerator renames all the type variables in callee's TypeEnvironment, and relies on the message send inclusion rule to match equivalent type variables.

For example, consider the callee's declared type

$$<'< Array\ of: ElementType > at:< SmallInteger > \uparrow\ < ElementType >'>$$

and the actual message send

$$<'< Array\ of: T1 > at:< SmallInteger > \uparrow < T2 >'> .$$

Since there is no pseudo-type in the formal interface, no pseudo-type conversion is necessary. The type variable $<$*ElementType*$>$ is renamed to an unused name, say $<$*T10*$>$. By the message send inclusion rule, $<$*Array of: T10*$>$ must match $<$*Array of: T1*$>$, which means that $<$*T10*$>$ is equivalent to $<$*T1*$>$.

Because only type variables in the callee's TypeEnvironment are renamed, the TypeVariableGenerator never renames a type variable created by its own TypeEnvironment. Also, a type variable is never renamed more than once. On the other hand, all type variables in the callee's environment are always renamed.

```
typeCheckIn: typeEnv
    (self generateAllPossibleCasesFrom: caseVars) do:
        [ :ext |
            typeInfo extendedBy: ext.
            self caseAnalysesMessageSendIn: typeInfo.
            typeInfo removeExtension]
```

Figure 4.19: MessageSPN>>typeCheckIn:.

4.3.3 TypeTable

The TypeTable maps each node in the type tree to its type. Its type accessing protocol consists of two methods: initTypeOf: and typeOf:, which store and return the type of a type tree node respectively. In addition, the TypeTable supports case analysis using two methods in the type-checking protocol: extendedBy: and removeExtension. Suppose the type system decides to perform case analysis on the variables x <*SmallInteger + UndefinedObject*> and y <*SmallInteger + Character*>. The TypeTable needs to extend itself four times, each time with a different case (e.g. { (x, <*SmallInteger*>), (y, <*SmallInteger*>) }, { (x, <*SmallInteger*>), (y, <*Character*>) }, ...). When the case analysis is completed, the TypeTable reverts to the original types for x and y. The code in Figure 4.19 performs the above process.

Because of the need to push and pop the type environment extensions, the TypeTable is implemented as a stack of Dictionaries. During case analysis, a dictionary containing the 'new' types of the variables is pushed onto the TypeTable. This is done by calling the method extendedBy. When the TypeTable needs to access the type of a variable, it searches down the stack of dictionaries so that the 'new' types are encountered first. After a case analysis, the 'new' types are popped from the stack, revealing the original types. The removal of extensions is implemented by removeExtension.

4.3.4 TypeEquationSolver

The TypeEquationSolver maintains inclusion relationships between type variables and their ranges (upper and lower bounds). The TypeEquationSolver is essentially a directed graph of type variables. The directed graph is stored in the instance variable typeVariableDictionary, which is a Dictionary from TypeVariables to TypeInclusionNodes. A TypeInclusionNode acts as a graph node and a record for its associated TypeVariable. Each TypeInclusionNode has an upper bound and a lower bound, a successor and a predecessor. The upper bound represents the requirement-flow information while the lower bound holds the data-flow information of the type variable. When a type variable t includes a type variable s, s becomes the successor of t and t becomes the predecessor of s in the directed graph. At the same time, the lower bound of s is propagated to all its predecessors while the upper bound of t is propagated to its successors. The TypeEquationSolver has the invariant that the upper bounds of all type variables must include their corresponding lower bounds.

The TypeEquationSolver is only visible by the TypeEnvironment. The client of the TypeEquationSolver is usually a TypeVariable which needs to add type information to its TypeInclusionNode. The added type information will propagate to the other TypeInclusionNodes via the edges in the directed graph. In general, all requests to the TypeEquationSolver are made via the TypeEnvironment, which is passed around during type-checking. The important protocols of TypeEquationSolver are updating, query, pruning, and type checking. The updating protocol consists of the following methods: addTypeVariable:, let:beIncludedIn:, let:include:, let:match:, and let:alias:. Their functions are described in Table 4.8.

The query protocol allows the client to ask questions about the current type graph. For example, the evaluate:in: method answers the 'value' of a type variable. The lower bound of a type variable is assumed to be the actual type represented by the type variable. If the type variable does not have a well-defined lower bound, then the type variable itself is returned. The other three methods isAlias, unAlias, and selectRepresentativeOf: are used to support the equivalence class computation. Equivalence class computation occurs at two places. The first is when two type variables match each other (e.g. when type-checking a message send, the receiver type must match), and the second is just before the method type declaration is printed.

When an equivalence class is identified, a representative is selected from the equivalence class by calling the method selectRepresentativeOf:. Given a set of type variables, this method chooses the class type parameter or any user-declared type (in that order) to be a representative. If none is found, any type variable can be a representative. Once a representative is selected, the other type variables in the equivalence class become aliases to the representative. Suppose $<\mathit{Rep}>$ is selected to be a representative for $<\mathit{T1}>$. At this point, all the type information

Selector	Action
addTypeVariable:	Add a new type variable created by the TypeVariableGenerator or from a callee's TypeEnvironment to my typeVariableDictionary
let:include:	If the second argument is a type variable, then add the upperbound of the first argument to the upperbound of the second argument and its successors. At the same time, add the lowerbound of the second argument to the lowerbound of the first argument and its predeccessors. If the second argument is not a type variable, then add the second argument to the upperbound of the first argument.
let:beIncludedIn:	Same as above, except that the roles of the first and second arguments are reversed.
let:match:	This is defined as a two-way inclusion. This method is used when the first argument is a type variable but the second argument is not.
let:alias:	This method is for matching two type variables. This is separated from the let:match: implementation because equivalence class computation of type variables may be necessary.

Table 4.8: The updating protocol

in the TypeInclusionNode of <*T1*> is added to that of <*Rep*>. Then the TypeInclusionNode of <*T1*> is replaced by the TypeInclusionNode of <*Rep*>. From now on, whenever the type inference algorithm reads or writes information to the TypeInclusionNode of <*T1*>, the TypeInclusionNode of <*Rep*> is used.

Because of aliasing, a TypeInclusionNode has an instance variable typeVar that contains its TypeVariable. Recall that the graph of type variables is implemented by a Dictionary of TypeVariables to TypeInclusionNodes. If a type variable is an alias then the typeVar instance variable in its TypeInclusionNode will be different from the one used to access the TypeInclusionNode. The method isAlias answers whether a type variable is an alias. The method unAlias answers the representative for a TypeVariable. An alternative way to implement the aliasing is to rename the aliased type variables to their representative's name. This is not done for the following reason. Very often, backtracking is used to test for an inclusion. For example,

```
aSignatureType includes: anObjectType in: (testEnv ← typeEnv typeCopy) ifFail: [ :ex | ...]
```

If type variables are renamed, then when the inclusion check fails, the type variables have to be renamed back. Using the above scheme, only type information in typeEnv has changed. As a result, the copy of TypeEnvironment may be discarded. The method typeCopy makes a copy of the TypeEnvironment's type graph (instead of the whole TypeEnvironment). It is implemented for efficiency only. The method replaceTypeInfoFrom: can be used to replace the type graph of a TypeEnvironment with the type graph of another TypeEnvironment.

The pruning protocol consists of the type graph pruning method cleanUp, which removes all aliases and other useless type variables (e.g. the type variables used for case analysis) from the type graph. The remaining type information is associated with the the temporaries, the arguments, the block arguments, and the return type. This information is needed for printing the method type declaration.

4.4 Error Handling

Error handling is an important part of the type system implementation. Shown below is a list of errors that may occur during type-checking:

- method not defined — the called method does not exist.
- method not typed — the called method is defined, but is not type-checked.
- type inclusion error — a wrong argument has been passed to a method, the return type of a message send is inconsistent with the user-declared return type, or a variable is assigned the wrong type.
- user-defined error method — the called method should not be invoked at all. For example,

```
foo: arg
    ↑self shouldNotImplement

foo: arg
    ↑self subclassResponsibility

foo: arg
    self error: 'Not used anymore'
```

```
test
        | temp |
        temp ← Array new.
        1 to: 100 do: [temp ← Array with: temp].
```

Figure 4.20: Recursive type error.

Whenever the above methods are called, we can say that a type error has occurred. These methods are created in an untyped environment to capture type errors. Hence, it is only reasonable that the type system take advantage of this fact to signal a warning.

- recursive type error — a recursive type has occured causing the type system to iterate infinitely. Consider the method in Figure 4.20. Because the lower bound of temp always changes after type-checking the assignment temp ← Array with: temp, an additional iteration is always required. As a result, the type system will loop forever, trying to type-check the above method using new assumptions from the previous iteration. Currently, an explicit check is performed to detect and report such condition.

The Signal and Exception classes are used to implement the error handling part of the type system. Two kinds of signals, abortSignal and failSignal, are used. AbortSignal handles the method not defined and method not typed errors while failSignal handles the user-defined error, the type inclusion error and the recursive type error. In general, if failSignal raises an exception then type-checking has failed completely. An error notifier will appear on the screen and the method will be 'rejected' by the Smalltalk compiler. On the other hand, if abortSignal raises an exception while filing-in a Smalltalk method then type-checking is aborted, but might complete successfully if retried later. An error message is printed in the Transcript and the method being type-checked has its constraints flag set to true, indicating that it is not completely type-checked. The compiler proceeds to accept the method. By doing so, mutually recursive methods can be accepted and type-checked (the first method has to be filed-in again). If abortSignal raises an exception while accepting a method in the system browser, a notifier will appear on the screen (instead of an error message in the Transcript). As before, the current method will be accepted and the constraints flag of the current method will be set to true.

```
preorderDo: preBlock stoppingAt: stopBlock updateUsing: postBlock
	| handlers |
	handlers ← HandlerCollection new.
	handlers on: self failSignal handle: [ :ex |
		typeInfo initTypeOf: self to:
			(IllegalBlockType error: ex parameter signal: self failSignal)
	handlers on: self abortSignal handle: [ :ex |
		typeInfo initTypeOf: self to:
			(IllegalBlockType error: ex parameter signal: self abortSignal)
	handlers handleDo:
		[↑super preorderDo: preBlock stoppingAt: stopBlock updateUsing: postBlock]
```

Figure 4.21: Exception handling for IllegalBlock.

To enable both the type trees and the types to signal type errors, the signals are stored as class variables in the superclass of the Type classes and the SPNode classes. When the exception handler receives an exception, it creates an Error object that stores the error message and the type tree node causing the error. The type tree node can be an assignment node, a message node, or a return node. After the handler stores the Error object in the exception, it rejects the exception, causing the system to look for another handler higher up the type tree. If a type error occurs within the sub-tree rooted at a Block node, then the type of the Block node should be <*IllegalBlock*>. Thus, a handler is needed at the Block node to intercept any type error exception. This means that an exception handler must be installed before type-checking the children of a Block node. To achieve this, the method preorderDo:stoppingAt:postorderDo: in BlockSPN is redefined to use the do:handle: method of class Signal (see Figure 4.21).

When a Block node captures an exception, the Error object and the Signal are stored in an IllegalBlock object. The IllegalBlock object will stay dormant in the type tree until it encounters a type inclusion check where another exception is raised. This time, the violating error condition is still the original Error object (contained in the IllegalBlock). The type system continues to look for a suitable handler. Finally, at the top level type-checking method, all exceptions are trapped and reported to the user accordingly.

The signal package is also used to implement backtracking. When a type error is anticipated, a special form of type operation is used (see Figure 4.22). The ifFail: extension handles exception

```
intersectObjectType: anObjectType in: typeEnv
    | env |
    self matches: anObjectType in: (env ← typeEnv typeCopy)
        ifFail: [ :ex | ↑Nothing uniqueType].
    typeEnv replaceTypeInfoFrom: env
```

Figure 4.22: Example of backtracking.

raised by the abortSignal. The replaceTypeInfoFrom: method replaces all the type inclusion relations of a TypeEnvironment by those in another TypeEnvironment. The ifFail: extension is also available to other type operations such as includes:in:. These special operations are collected in the type operations with backtracking protocol in class Type.

5 Conclusion

The TS type system makes a distinction between subclassing and subtyping. The conflicts between subclassing and subtyping are resolved by focusing on subclassing only. Because no assumption (regarding specifications) can be made about a Smalltalk class and its subclasses, no subtyping relationship is defined for the object types. In addition, code that performs explicit class checking can be type-checked automatically by using constraints and case analysis. This allows the type system to type-check as many unmodified Smalltalk programs as possible. Finally, given the argument type(s) of a method, the TS type system is able to infer a specific return type. By using this information, the compiler can produce better code.

Type-checking Smalltalk programs is a hard problem. Many areas still remain unsolved. For instance, the current TS type system needs a flow analyzer to type-check programs which 'nil-out' temporary and global variables. Otherwise, class UndefinedObject may need to understand all messages in the image. Currently, the TS type system also does not keep track of the dependencies between typed methods. When the interface of a method is changed, there is no mechanism to propagate the changes to all its callers. So the user need to keep track of these dependencies manually. Furthermore, the current definition of object types forces the user to use explicit union types (e.g. $<\mathit{SmallInteger} + \mathit{LargePositiveInteger} + \mathit{LargeNegativeInteger}>$) or to rely on constraints. By introducing subtyping between object types, the type system can offer more encapsulation and abstraction. Finally, when the user does not declare the argument types of a method, backward type inference can be used to infer them. By doing so, the programmer can be freed from declaring type information for the methods. All these problems have been identified and more work will be done on them.

BIBLIOGRAPHY

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA '90, Object-Oriented Programming Systems, Languages and Applications*, pages 161–168, 1990.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Functional Programming Languages and Computer Architectures '89*, pages 37–44, 1989.

[GJ90] Justin Graver and Ralph Johnson. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

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

[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).

[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).

[Mey87] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1987.

[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).

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