A Type System for Smalltalk

Justin O. Graver
University of Florida

Ralph E. Johnson
University of Illinois at Urbana-Champaign

Abstract

This paper describes a type system for Smalltalk that is type-safe, that allows most Smalltalk programs to be type-checked, and that can be used as the basis of an optimizing compiler.

1 Introduction

There has been a lot of interest recently in type systems for object-oriented programming languages [CW85, DT88]. Since Smalltalk was one of the earliest object-oriented languages, it is not surprising that there have been several attempts to provide a type system for it [Suz81, BI82]. Unfortunately, none of the attempts have been completely successful [Joh86]. In particular, none of the proposed type systems are both type-safe and capable of type-checking most common Smalltalk programs. Smalltalk violates many of the assumptions on which most object-oriented type systems are based, and a successful type system for Smalltalk is necessarily different from those for other languages.

We have designed a new type system for Smalltalk. The biggest difference between our type system and others is that most type systems for object-oriented programming languages equate classes with types and subclassing with subtyping [Suz81, BI82, SCB*86, Str86, Mey88]. In our type system, types are based on classes (i.e. each class defines a type or a family of types) but subclassing has no relationship to subtyping. This is because Smalltalk classes inherit implementation and not specification.

Our type system uses discriminated union types and signature types to describe inclusion polymorphism and describes functional polymorphism (sometimes called parametric polymorphism, see [DT88]) using bounded universal quantification. It has the following features:

- automatic case analysis of union types,
- effective treatment of side-effects by basing the definition of the subtype relation on equality of parameterized types, and
- type-safety, i.e. a variable's value always conforms to its type.

Because the type system uses class information, it can be used for optimization. It has been implemented as part of the TS (Typed Smalltalk) optimizing compiler [JGZ88].

2 Background

Smalltalk [GR83] is a pure object-oriented programming language in that everything, from integers to text windows to the execution state, is an object. In particular, classes are objects. Since everything is an object, the only operation needed in Smalltalk is message sending. Smalltalk uses run-time type-checking. Every message send is dynamically bound to an implementation depending on the class of the receiver. This unified view of the universe makes Smalltalk a compact yet powerful language.

Smalltalk is extremely extensible. Most operations are built out of a small set of powerful primitives. For example, control structures are all implemented in terms of blocks, which are the Smalltalk

Authors' address, telephone, and e-mail:
Department of Computer and Information Sciences, E301 CSE, Gainesville, FL 32611, (904) 392-1507, graver@cis.ufl.edu
Department of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801, (217) 244-0093, johnson@cs.uiuc.edu

equivalent of closures. Smalltalk's equivalent to if-then-else is the ifTrue:ifFalse: message, which is sent to a boolean object with two block arguments. If the receiver is an instance of class True then the "true block" is evaluated. Similarly, if the receiver is an instance of class False then the "false block" is evaluated. Looping is implemented in a similar way using recursion. These primitives can be used to implement case statements, generators, exceptions, and coroutines[Deu81].

Smalltalk has been used mostly for prototyping and exploratory development. It is ideal for these purposes because Smalltalk programs are easy to change and reuse, and Smalltalk comes with a powerful programming environment and large library of generally useful components. However, it has not been used as much for production programming. This is partly because it is hard to use for multi-person projects, partly because it is hard to deliver application programs without the large programming environment, and partly because it is not as efficient as languages like C. In spite of these problems, the development and maintenance of Smalltalk programs is so easy that more and more companies are developing applications with it.

A type system can help solve Smalltalk's problems. Type information makes programs easier to understand and can be used to automatically check interface consistency, which makes Smalltalk better suited for multiperson projects. However, our main motivation for type-checking Smalltalk is to provide information needed by an optimizing compiler. Smalltalk methods (procedures) tend to be very small, making aggressive inline substitution necessary to achieve good performance. Type information is required to bind methods at compile-time. Although some type information can be acquired by dataflow analysis of individual methods [CUL89], explicit type declarations produce better results and also make programs more reliable and easier to understand. From this point of view, the type system has been quite successful, because the TS compiler can make Smalltalk programs run nearly as fast as C programs.

It is important that a type system for Smalltalk not hurt Smalltalk's good qualities. In particular, it must not hinder the rapid-prototyping style often used by Smalltalk programmers. Exploratory programmers try to build programs as quickly as possible, making localized changes instead of global reorganization. They often use explicit run-time checks for particular classes (evidence that code is in the wrong class) and create new subclasses to reuse code rather than to organize abstraction (evidence that a new abstract class is needed) [JF88]. Although conscientious programmers remove these improprieties from finished programs, we wanted a type system that would allow them, since they are often important intermediate steps in the development process. Thus, a static type system for Smalltalk must be as flexible as the traditional dynamic type-checking.

```
class Change:
    values
        ↑Array with: self class with: self parameters
    parameters
        self subclassResponsibility

class ClassRelatedChange:
    parameters
        ↑className

class MethodChange:
    parameters
        ↑Array with: className with: selector

class OtherChange:
    parameters
        ↑self text
```

Figure 1: Change and its subclasses.

In an untyped object-oriented language like Smalltalk, classes inherit only the implementation of their superclasses. In contrast, most type systems for object-oriented programming languages require classes to inherit the specification of their superclasses [BI82, Car84, CW85, SCB*86, Str86, Mey88], not just the implementation [Suz81, Joh86]. A class specification can be an explicit signature [BHJL86], the implicit signature implied by the name of a class [SCB*86], or a signature combined with method pre- and post-conditions, class invariants, etc. [Mey88]. Inheriting specification means that the type of methods in subclasses must be subtypes of their specifications in superclasses.

Since Smalltalk is untyped, only implementation is inherited. This makes retrofitting a type system difficult. Since specification inheritance is a logical organization, many parts of the Smalltalk inheritance hierarchy conform to specification inheritance, but there is nothing in the language that requires or enforces this. Thus, it is common to find classes in the Smalltalk-80 class library that inherit implementation and ignore specification.

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

Abstract classes, a commonly used Smalltalk programming technique, provide other examples where classes inherit implementation but not specification. Consider the method definitions shown in Figure 1. These (and other) classes are used to track changes made to the system. The abstract class Change defines the values method in terms of the parameters method. The implementation of the values method is inherited by the subclasses of Change. The result of sending the values message depends on the result of sending the parameters message, which is different for each class in Figure 1. Hence, the specification of the values method is also different for each of the classes.

A class-based type system with explicit union types works for an implementation inheritance hierarchy and greatly simplifies optimization, but is not very flexible when it comes to adding new classes to the system. To regain some of the flexibility we incorporate type-inference and signatures. This gives us both flexibility and the ability to optimize.

A type system suitable for an optimizing Smalltalk compiler must also have parameterized types. It is difficult to imagine a (non-trivial) Smalltalk application that does not use Collections. The compiler must be able to associate the type of objects being added to a Collection with the type of objects being removed from it. Without parameterized types, all Collections are reduced to homogeneous groups of generic objects. Furthermore, due to the potentially imperative side-effects of any operation, a special imperative definition of subtyping for parameterized types is used (see Section 3.1).

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

For a detailed discussion of these issues see [Gra89].

3 Types

The abstract and concrete syntax for type expressions is shown in Figure 2 (abstract syntax on the left and concrete syntax on the right). In the concrete syntax grammar terminals are underlined, (something)* represents zero or more repetitions of the something, and (something)$^+$ represents one or more repetitions of the something. Each of the type forms will be described in detail in the following sections.

We use the abstract syntax in inference rules and the concrete syntax in examples. Type expressions are denoted by s, t, and u. Type variables are denoted by p (abstract syntax) or by P (concrete syntax). Lists (tuples) of types are denoted by $\vec{t} = < t_1, t_2, \ldots, t_n >$. The empty list is denoted by $<>$ and $t \cdot \vec{t}$ is the list $< t, t_1, t_2, \ldots, t_n >$. Similar notation is used for lists of type variables. C denotes a class name and m denotes a message selector.

Strictly speaking, a type variable is not a type; it is a place holder (or representative) for a type. We assume that all free type variables are (implicitly) universally quantified at an appropriate level. For example, the local type variables of a method type are assured to be universally quantified over the type of the method; the class type parameters of a class are assumed to be universally quantified over all type definitions in the class. We also assume that all type variables are unique (i.e. have unique names) and are implicitly renamed whenever a type containing type variables is conceptually instantiated. For example, the local type variables of a method type are renamed each time the method type is "used" at a call site. We do use explicit range declarations for type variables to achieve bounded universal quantification (see Sections 3.5 and 3.6). Given these assumptions, we treat type variables as types.

3.1 Subtyping

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

The following are basic inference rules.

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

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

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

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

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

	t ::=	$Type$ ::=
(object type)	$C(\vec{t})$	$ClassName\ (\underline{\mathsf{of:}}\ Type)^*$
(block type)	$\mid \vec{t} \to t$	$\mid \underline{\mathsf{Block}}\ (\underline{\mathsf{of:}}\ Type)^*\ \underline{\mathsf{returns:}}\ Type$
(union type)	$\mid t + t$	$\mid Type\ \underline{+}\ Type$
	$\mid (t)$	$\mid \underline{(}Type\underline{)}$
(signature type)	$\mid\ < m, \vec{t} \to t >^*$	$\mid \underline{\mathsf{understands:}}\ \underline{(}(MsgName\ (\underline{\mathsf{of:}}\ Type)^{\underline{+}}\underline{\mathsf{returns:}}\ Type)^*\underline{)}$
	$\mid (t)$	$\mid \underline{(}Type\underline{)}$
(type variable)	$\mid p$	$\mid P$
($\top$)	$\mid Anything$	$\mid \underline{\mathsf{Anything}}$
($\bot$)	$\mid Nothing$	$\mid \underline{\mathsf{Nothing}}$

Figure 2: The abstract and concrete syntax of types.

The following rules are used for the type parameter lists of object types (see Section 3.2) and for the argument type lists for block types (see Section 3.4).

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

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

Subtyping is reflexive, transitive, and antisymmetric.

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

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

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

3.2 Object Types

Some variables contain only one kind of object. For example, all Characters[1] have an instance variable that contains a SmallInteger ASCII value. Other variables can contain different kinds of objects. For example, a Set can contain SmallIntegers, Characters, Arrays, Sets, and so on. Each class has a set of zero or more type variables called *class type parameters* that can be used in type expressions to specify the types of instance variables. These are the only type variables that can appear in type declarations for instance variables.

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

Due to the imperative nature of Smalltalk [GR83, Joh86] and because of the antimonotonic ordering of function types [DT88] we define subtyping for object types as equality, i.e. one object type is included in another if and only if they are equal (Cardelli also takes this approach for "updatable" objects in [Car85]). (Recall that $\mathcal{C}$ is a set of inclusion constraints and C is a class name.)

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

3.3 Union Types

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

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

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

Our type system does not use the subclass relation on classes to induce the subtype relation on types. Instead, type inclusion is based on discriminated union

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

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

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

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

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

Some examples are:

Character $\sqsubseteq$ *Character* + *SmallInteger*
Array of: SmallInteger $\sqsubseteq$
(Array of: SmallInteger) + *(Array of: Character)*
Array of: SmallInteger $\not\sqsubseteq$
Array of: (SmallInteger + *Character)*

Note that our type system does not reflect class inheritance. Even though Integer is a subclass of Number, *Integer* is not a subtype of *Number*. However, we can specify the type of all subclasses of Number by listing them, i.e. *Integer* + *Float* + *Fraction*.[2]

3.4 Block Types

Blocks (function abstractions) are treated differently from other objects. A *block type* consists of the name *Block*, a possibly empty list of types for block arguments, and a return type. For example,

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

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

A block type u is included in a block type u' only if the return type of u is included in the return type of u' and if each argument type of u' is included in the corresponding argument type of u (this is the standard antimonotonic subtype relation for function types [MS82]).

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

For example,

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

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

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

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

Figure 3: Class definition for OrderedCollection.

3.5 Typed Class Definitions

All instance variables and class variables must have their types declared. Each class defines a set of type variables called class type parameters, which may be used to specify the types of instance and class variables.

Figure 3 shows the definition of OrderedCollection, which is a subclass of SequenceableCollection. Note that the type of each instance variable is declared when the variable is declared and that angle brackets are used to set off type expressions from the regular Smalltalk code.

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

OrderedCollection of: ElementType.

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

The *range* (upper bound for type instantiation) of a class type parameter may be restricted by including an optional range type declaration. A class type parameter may only be instantiated to a type that is a subtype of its range type. The

```
{ localTypeVariables: <P1> <P2>
  receiverType: <Self>
  arguments: arg1 <arg1Type> arg2 <arg2Type>
  temporaries: temp <tempType>
  blockArguments: blkArg <blkArgType>
  returnType: <Self> }

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

      | temporary variable names |
      statements
```

Figure 4: Template for typed methods.

range declaration of a class type parameter is syntactically identical to the type declaration for a variable. In the above example, if we wished to restrict the elements of OrderedCollections to be Characters, the typeParameters: field would be declared as 'ElementType <*Character*>'. If the range declaration of a class type parameter is omitted it is assumed to be *Anything*.

3.6 Typed Method Definitions

The types of method arguments, temporary variables, and block arguments can be explicitly declared or inferred by the TS type-inference mechanism [Gra89]. The only difference between a typed method and an untyped method is that the typed method has type declarations. Type declarations must precede any part of the method definition. A typed method template is shown in Figure 4. The arguments for the fields arguments:, temporaries:, and blockArguments: are lists of identifier <*type*> declaration pairs. The arguments for the receiverType: and returnType: fields are single type declarations. The arguments for the localTypeVariables: field are capitalized identifiers. The range of a type variable can be restricted by using the range: modifier. For example, the local type variable declaration <*P range: (Integer + Character)*> declares *P* to be a type variable that can be associated with any of the types *Integer*, *Character*, or *Integer + Character*. Local type variables are instantiated during type-checking to types determined by the types of actual arguments at a call site. Local type variables will usually, though not necessarily, correspond to the class type parameters of some class.

The do: method for SequenceableCollection is shown in Figure 5 as an example of a typed method definition. Notice that certain fields of the type declarations have been omitted. The syntax rules for type declarations are fairly liberal. Fields can occur in any order. All fields except receiverType: and returnType: can have multiple occurrences. The smallest type declaration for a method is "{}."

Self represents the type of the pseudo-variable self (and super). Unfortunately, the class of self is different in each class that inherits a method, so *Self* must differ, too. Thus, each method is type checked for each class that inherits it by replacing *Self* with the type appropriate for the inheriting class. In the absence of a receiverType: declaration (which is over 99% of the time), *Self* defaults to the object type for the class in which the method is being compiled. Otherwise, *Self* is replaced with the type given in the receiverType: declaration. A legal receiverType: must be a subtype of the default value of *Self*.

Returning to the example of Figure 5, SequenceableCollection has one class type parameter called *ElementType*, so the type substituted for *Self* will be *SequenceableCollection of: ElementType*. The argument aBlock is a block that takes one argument of type *ElementType* and returns an object of unknown type *P*. When do: is envoked in another method definition both *ElementType* and *P* will be associated with actual receiver and argument types. The temporary variables index and length have values dependent on the size of Collections. In Smalltalk implementations with 30+ bit SmallIntegers, the possible size of any Collection is well within the range of SmallIntegers.[3] There is no explicit return statement in the method so it implicitly returns self.

Type declarations and typed methods introduce the notions of message and method types. A *message type* consists of a message selector, a receiver type, a list of argument types, and a return type. A *method type* consists of a message type and a set of constraints on the type variables used in the message type. A method type denotes the type of a method relative to a specific class of receivers. Therefore, *Self* should already have been expanded and should never appear in a method type. The type of the above method is denoted by

<*SequenceableCollection of: ElementType*>
    do: <*Block of: ElementType returns: T*>
    ↑<*SequenceableCollection of: ElementType*>.

Message types use the same notation. Local range constraints on type variables are shown enclosed in

[3] Smalltalk defines infinite precision integer arithmetic using the classes LargeNegativeInteger, SmallInteger, and LargePositiveInteger (the three subclasses of class Integer). SmallIntegers are essentially machine integers; instances of the Large-Integer classes are more complex.

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

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

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

Figure 5: The do: method for SequenceableCollection.

braces following the return type of a method type. For example,

```
<Object> foo: <P> ↑<P>
{<P> ⊑ <Integer + Character>}.
```

Subtyping for message types, although complicated by type variables (see [Gra89]), is essentially the same as for block types, with the added condition that the selectors must be equal.

3.7 Metaclass Method Definitions

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

Similarly, when defining methods in a metaclass it is useful to be able to refer to the type of instances of the metaclass. This is done by using the *InstanceType* type specification. For example, Figure 6 shows the typed method definition of new: for the Set metaclass.

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

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

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

Figure 6: The new: method in Set class.

4 Type-Checking

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

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

4.1 Basic Type-Checking

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

The types of all variables are declared.

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

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

$$\vdash \underline{\mathsf{true}} : \mathit{True}$$

$$\vdash \underline{\mathsf{false}} : \mathit{False}$$

$$\vdash \underline{\mathsf{nil}} : \mathit{UndefinedObject}$$

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

$$\vdash n : \mathit{Integer}$$

$$\vdash \underline{\$}c : \mathit{Character}$$

$$\text{etc.}$$

If this type is not general enough then an explicit type declaration can be used to supply the "correct" type [Gra89].

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

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

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

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

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

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

An additional inference rule for blocks is given in Section 4.2.

A return statement $\uparrow e$ is type-correct if e is type-correct and if the type of e is a subtype of the declared return type of the method. The set of assumptions $\mathcal{A}$ contains a special variable returnValue whose type is the declared return type of the method being type-checked. The only place a return statement can appear is as the last statement in the statement list of a method or a block. The types of the top-level statements in the statement list of a method can be ignored. The type of a block will always be checked for inclusion within some block type. To afford the maximum freedom to such blocks, the type given to return statements is the special type *Nothing*, which is included in any type.

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

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

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

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

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

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

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

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

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

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

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

Message sends to receivers with union types are handled by an inference rule in Section 4.2.

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

$$<\mathit{Array\ of{:}\ P}>\ \mathsf{at{:}}\ <\mathit{SmallInteger}>\ \uparrow<P>.$$

The inference rule for message sends produces a set of inclusion equations

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

whose solution is

$$P \sqsubseteq \textit{Character}.$$

In general, there may not be a unique solution to a given set of inclusion equations. How to deal with (avoid) multiple solutions is discussed in [Gra89]. If no solution exists then there is a type error somewhere.

An inference rule is also needed to type-check method definitions. Recall that the abstract definition for a method m looks like

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

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

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

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

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

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

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

4.2 Case Analysis

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

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

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

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

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

The typed method definition of controlToNextLevel for class Controller is shown in Figure 7. The notNil message is defined to return an object of type *True* for all classes except UndefinedObject, for which it is defined to return an object of type *False*. If the type of aView is assumed to be *View* then the type of the expression aView notNil is *True*. The ifTrue: method defined for class True has a type

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

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

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

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

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

4.3 Inheritance

One way that inheritance complicates type-checking is that the type of a method for a subclass that inherits it is slightly different from its type in the class that defines it. For example, the type of the receiver is different, and the type of the returned value will be different when the receiver is returned. This problem

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

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

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

Figure 7: The controlToNextLevel method for class Controller.

is solved by referring to the type of the receiver as *Self* and expanding *Self* to the type of the receiver in each subclass. *Self* is not a type, but instead is "macro-expanded" to a type at compile-time.

Abstract classes provide another way in which the type of a method can change when it is inherited. Consider the method definitions shown in Figure 1. The values method is defined in class Change and inherited by the other classes. The return type of values depends on the return type of parameters, which is different for each class. The types of the different parameters methods are

$<$*Change*$>$ parameters $\uparrow<$*Change*$>$
$<$*ClassRelatedChange*$>$ parameters $\uparrow<$*Symbol*$>$
$<$*MethodChange*$>$ parameters $\uparrow<$*Array of: Symbol*$>$
$<$*OtherChange*$>$ parameters $\uparrow<$*String*$>$

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

Another way in which abstract classes show that Smalltalk classes inherit implementation, not specification, is that some methods cannot be executed by all of the classes that inherit them. For example, class Collection has a number of methods that use the add: message, such as addAll:, but it does not implement or inherit the add: message itself. Instead, add: is implemented by the various subclasses of Collection. Some of its subclasses, such as Array, do not implement add: and thus cannot use methods like addAll:. Smalltalk relies on run-time type-checking and the "does not understand" error to detect when an undefined message is sent to an object. Our type system detects all such cases at compile-time.

These problems are solved by retype-checking methods in each subclass that inherits them. The method definition (defined in terms of *Self*) is inherited and *Self* is expanded to refer to the current class as described in section 3.6. We assume that the hierarchy of typed class definitions H (see Section 4.1) contains not only the user declared method definitions for a class, but also any methods that can be meaningfully (i.e. type-correctly) inherited by a class. If H_0 is a partial function from $< C, m >$ pairs to method definitions representing user declared methods then H is derived by extending H_0 to include definition points for inherited methods. In other words, if $H_0 < C, m >$ is a user defined method then $H < C', m >$ has the same definition provided that C' is a subclass of C and $H_0 < C', m >$ is not defined. There are many possible H function. A type-correct H is one in which, for all method types derivable from H, the declared type of $H < C, m >$ is the same as the type inferred by type-checking.

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

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

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

The benefits of delaying type-checking of inherited methods is that a method is only retype-checked for every subclass that actually inherits it, and that this type-checking is spread out over a large amount of time. A new class does not require any of the methods that it inherits to be type-checked when it is created.

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

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

Figure 8: whileTrue: for class BlockContext.

Adding a new method to a superclass does not require type-checking it for every subclass that inherits it, nor does adding a new variable. However, changing the type of a method or variable might require a lot of computation to ensure that each of its uses is still type-correct.

4.4 Specific Receivers

Some classes have methods that can be executed by only a subset of their instances. For example, the whileTrue: message should be sent only to blocks that return Booleans. We can specify this by using the receiverType: field in the type declaration of the method definition as shown in Figure 8. Recall that if this field is omitted then *Self* defaults to the object type for the class in which the method is being compiled. Although whileTrue: is nearly the only method in the Smalltalk-80 class library that needs to declare a specific receiver, it is easy to imagine other methods that would need this feature, such as a summation method in Collection.

4.5 Signature Types

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

$$\frac{\mathcal{C},\mathcal{A} \vdash e : t \qquad \mathcal{C},\mathcal{A} \vdash \vec{e} : \vec{t} \qquad \mathcal{C},\mathcal{A} \vdash (e\ m\ \vec{e}) : u}{\mathcal{C},\mathcal{A} \vdash t \sqsubseteq << m, t \cdot \vec{t} \rightarrow u >>}$$

$$\frac{\mathcal{C},\mathcal{A} \vdash t \sqsubseteq << m, t \cdot \vec{t} \rightarrow u >> \qquad \mathcal{C},\mathcal{A} \vdash t \sqsubseteq << m', t' \cdot \vec{t}' \rightarrow u' >>}{\mathcal{C},\mathcal{A} \vdash t \sqsubseteq << m, t \cdot \vec{t} \rightarrow u >, < m', t' \cdot \vec{t}' \rightarrow u' >>}$$

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

A signature type includes object types belonging to classes not yet created. Thus, signature types contain an infinite number of object types. In fact, the type specified by an empty signature contains every possible object type, since every object type understands every message type in the signature.

$$\mathcal{C},\mathcal{A} \vdash t \sqsubseteq <>$$

Since signature types specify the largest possible types, procedures that use them exhibit the most polymorphism and so are the most flexible. However, use of signature types prevents the compiler from performing some important kinds of optimizations, since they do not provide enough information about the class of the receiver to allow compile-time binding of message sends.

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

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

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

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

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

As an example, consider the occurrencesOf: method for class Collection shown in Figure 9. The only restriction placed on the type of the argument anObject is that its corresponding class (or classes) must implement (or inherit) an = (equality) message that will take an argument of type *ElementType* and return a boolean. It is helpful to compare the use of the do: message in this example with its definition in Figure 5.

Type-checking a message send to a receiver whose type is a signature type is straightforward since all relevant type information is contained in the signature type.

$$\frac{\mathcal{C},\mathcal{A} \vdash e : <\ldots < m, \vec{t}' \rightarrow u' > \ldots > \qquad \mathcal{C},\mathcal{A} \vdash \vec{e} : \vec{t} \qquad \mathcal{C}.\vec{t} \sqsubseteq \vec{t}' \vdash u' \sqsubseteq u}{\mathcal{C},\mathcal{A} \vdash (e\ m\ \vec{e}) : u}$$

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

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

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

Figure 9: The occurrencesOf: method in class Collection.

5 Beyond Signatures

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

Although Smalltalk is a typeless language, it is possible to discriminate between classes of objects and provide a simple kind of explicit run-time type-checking. An example is the isLookupKey message whose implementation is shown in Figure 10. Messages like isLookupKey can be used in a method to "type-check" an argument before sending it a message that it might not understand.

A example of this style of programming is the = (equality) message shown in Figure 11. Equality must be defined between any two objects and should not be subject to run-time errors. This style of implementation meets both of these criteria. The problem is how to type-check such a method.

The type of this method can be roughly stated as follows. An argument must understand the isLookupKey message. If an argument responds to this message with true then it must also understand the key message and the method will (presumedly) return an object of type *True + False*, otherwise the method returns an object of type *False*.

In general, the type of a polymorphic method like = is complicated, but its type for a specific use is simple and easy to understand. Thus, type-checking a method defers some of the type-checking for a method until its invocations are type-checked. It doesn't matter when a method is type-checked, only that type-checking is completed before any code that invokes the method is executed.

```
class Object:

        isLookupKey
            ↑false


class LookupKey:

        isLookupKey
            ↑true
```

Figure 10: Implementation of isLookupKey.

When a send of = to a LookupKey needs to be type-checked, the code in Figure 11 is substituted for the call. The types of the actual arguments then replace those of the formal arguments, allowing both definition and use information to be used in type-checking. This usually permits the = method to be type-checked completely. If not, then type-checking for the method that sends = must also be deferred. This static analysis technique has proven useful for type-inference as well as for type-checking [Gra89].

6 Type Safety

A type system can only be shown correct relative to a formal definition of the language. This section outlines a proof of correctness for the type system relative to Kamin's denotational semantics of Smalltalk [Kam88].

```
= aLookupKey

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

Figure 11: = (equality) in class LookupKey.

Usually a type is described as a set of objects. Type safety then means that the value of an expression is always contained in its type. However, our types are not sets of objects, so a different definition of type safety is needed. We will define type safety by assigning an object type to each object and then showing that the value of an expression always is assigned a type contained in the expression's type.

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

If each object had a type assigned to it then we could check whether an object was consistent with its type by checking whether its class was consistent with its type and whether the types assigned to the values of its instance variables were included in the declared types of its instance variables. This assignment would be consistent if every object was consistent with the type assigned to it. Although this type-assignment might not be unique, any consistent type-assignment could be thought of as describing the types of all objects. These ideas are formalized in Definition 1.

Definition 1 (Object/type consistency) *An object o is consistent with a mapping from objects to types ξ, relative to a class hierarchy* $\mathtt{H}$, *and a state ψ, if*

1. $\xi(o) = \mathtt{C}(\vec{\mathtt{t}})$,
2. *the class of o is* $\mathtt{C}$, *and*
3. *for each instance variable x_i of o, let t_i be the declared type of x_i with type variables replaced by the types given in $\xi(o)$. Then the type of the object referred to by x_i is a subtype of* $\mathtt{t}_i$.

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

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

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

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

The second way that the state changes is when a new object is created. The type of some new objects, such as new blocks, are known in advance so the new object will always be of the correct type. Unfortunately, problems occur with the **new** primitive and when creating a new context to evaluate a method. This is because instance variables of new objects and local variables of new contexts are both initialized to nil, but the types of these variables usually do not include *UndefinedObject*. Thus, until the variables are initialized, their value is not consistent with their type. We ensure type-safety by requiring all variables whose types do not include *UndefinedObject* to be assigned before they are read, and use flow analysis to check this. Thus, we will change the statement of the proposition slightly.

Type-Safety Theorem: *If a type-correct program is in a state that is consistent with a type assignment ξ except for unassigned variables, and if a variable is always assigned before it is read, then any succeeding state will be consistent with ξ.*

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

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

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

$$\mathsf{rcv\ k1{:}\ exp_1\ k2{:}\ exp_2 \ldots\ km{:}\ exp}_m$$

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

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

The type of

$$\mathsf{rcv\ k1{:}\ exp_1\ k2{:}\ exp_2 \ldots\ km{:}\ exp}_m$$

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

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

7 Conclusion

Our type system for Smalltalk is type-safe. It has been implemented in Smalltalk and used in the TS optimizing compiler for Smalltalk [JGZ88]. It has been able to solve most type-checking problems in the standard Smalltalk-80 class hierarchy. Thus, it is correct, useful, and usable.

Our type system is also unique. It differs from other type systems for object-oriented programming languages by acknowledging that only implementation is inherited, not specification, and by handling case analysis of union types automatically.

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

Acknowlegements

This research was supported by NSF contract CCR-8715752, by the AT&T ISEP grant, and by an equipment grant from Tektronix.

References

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

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

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

[Car85] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages, Proceedings of the 13th Summer School of the LITP*, Le Val d'Ajol, Vosges (France), May 1985.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of OOPSLA '89*, pages 49–70, October 1989. printed as SIGPLAN Notices, 24(10).

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

[Deu81] L. Peter Deutsch. Building control structures in the Smalltalk-80 system. *Byte*, 6(8):322–347, August 1981.

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

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

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

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

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

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

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

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

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

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of OOPSLA '86*, 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.

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