Strongtalk: Typechecking Smalltalk in a Production Environment

Gilad Bracha
gilad@longview.com

David Griswold
david@longview.com

Horizon Technologies of New York, Inc.
38 W. 32nd St., Suite 402,
New York, NY 10001

Abstract

Strongtalk^{TM} is a typechecker for a downward-compatible Smalltalk dialect. It is designed for large-scale production software development, and incorporates a strong, modern structural type system. It not only separates the notions of type and class, but also deals with the more difficult issue of separating inheritance and subtyping using the notion of inherited types [CHC90, Bru93a] to preserve encapsulation. Strongtalk integrates inherited types, metaclasses, blocks and polymorphic methods into a highly usable, full-scale language.

1 Introduction

It is widely accepted that a strong, static type system for a programming language has important benefits, including increased reliability, readability, and (potentially) performance. However, an inadequate type system can constrain the flexibility of a language, and thus its expressiveness.

Static typing allows the type safety of source code to be completely determined before execution. Understanding of effective ways to adapt such typing techniques to polymorphic languages with subtyping has matured only slowly over the last decade, beginning with [Car84]. For this reason (among others), languages such as *LISP*, *Smalltalk*, and *Self* that strive for extreme code reusability through various forms of polymorphism have traditionally foregone the benefits of static typing in favor of the more arbitrary flexibility of dynamic typing (where all types are verified on-the-fly during execution).

A number of attempts [Suz81, BI82, GJ90] have been made to retrofit a static type system to *Smalltalk*. Most of these efforts have been significantly complicated by the extraordinary requirement that they typecheck large bodies of existing *Smalltalk* code that were written without any notion of static type safety in mind.

This paper describes a new typechecker for *Smalltalk*, called *Strongtalk*, that has been designed with a quite different set of goals in mind. It is the belief of the authors that the challenge of the next decade will be to adapt the high-productivity programming styles of languages like *Smalltalk* to the very different rigors of very-large-scale software development, where issues such as encapsulation become more critical.

Accordingly, *Strongtalk* is an attempt to take advantage of recent typing advances by defining a new highly-encapsulated, production-oriented dialect within the *Smalltalk* language framework.

[0] ^{TM} Strongtalk is a trademark of Horizon Technologies of New York, Inc.

Appeared in proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 1993.

Smalltalk has been chosen as the base language not only because of its widespread acceptance and the availability of mature incremental development environments, but because it possesses several crucial features: extreme simplicity, garbage-collection, and literal first-class functions, in the form of blocks.

The following sections include a discussion of some of the important issues addressed by the *Strongtalk* design, followed by an exploration of the type system itself. After the type system has been introduced, our experiences with it are presented, followed by a section relating *Strongtalk* to other work. Finally, we present our conclusions.

2 Type System Overview

Before beginning our discussion of the design issues that have shaped *Strongtalk*, we present a brief list of its major attributes for those who wish an overview. The *Strongtalk* type system

- has a purely structural (rather than name-based) form, which provides for flexible and complete separation of the subtype and subclass lattices, while using the concept of brands [Nel91] to provide a notion of type identity.
- includes parameterized types and classes (bounded and unbounded quantification).
- supports polymorphic messages (messages whose type signatures are parameterized by a type argument), with a flexible programmer-controlled mechanism for automatically inferring values for the type parameter.
- distinguishes inheritance from subtyping, which is important for a number of forms of code sharing, by incorporating a tractable form of the recent notion of inherited types [Bru93a, CHC90, CCHO89], which are especially useful for describing abstract data type (ADT) hierarchies.
- preserves the subtype relationships between classes defined by the *Smalltalk* metaclass hierarchy, and relates them to the types of their instances.
- provides facilities for dynamic typing.

3 Design Issues

> *The language designer should be familiar with many alternative features designed by others, and should have excellent judgement in choosing the best and rejecting any that are mutually inconsistent.... One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.*
> *C.A.R. Hoare*

Type systems have a deserved reputation for being notoriously difficult things to design and reason about. One of the motivations for the development of *Strongtalk* was a feeling of dissatisfaction with the usability of widely available statically-typed object-oriented languages. When coming from a dynamically-typed background, one can easily end up feeling that the "devil is in the details."

Although "checking" types may sound to the uninitiated like a straightforward task, a straightforward type system can enormously complicate such seemingly mundane tasks as passing an array of strings to a method that needs a read-only array of any type of object. Whenever you place strict limits on programmers who are accustomed to none at all, you must be very careful indeed that you have anticipated and provided for effective ways of expressing solutions to a great and subtle variety of common programming problems.

In this section we will lay the groundwork for a presentation of the *Strongtalk* type system by discussing our major design goals, and by drawing attention to a number of the not-so-obvious issues

that can profoundly affect the expressiveness of an object-oriented type system.

3.1 Compatibility

Although *Strongtalk* is not designed to typecheck existing *Smalltalk* code without modification, neither is it intended as an incompatible language extension. Our goal has been to eliminate the risk that developers have traditionally had to assume when switching to a new, untried language. We achieve this through 'downward compatibility', which means that *Strongtalk* code can always be trivially converted to *Smalltalk,* but not necessarily vice-versa. Using downward compatibility as our benchmark has the benefits of giving us some leeway to define a very clean type system, while guaranteeing adopters of *Strongtalk* that at any point they can switch back to *Smalltalk* if they so desire, without recoding. Other than three trivial translations (for manipulating uninitialized variables and dynamic typechecking), no code transformation is performed.

The *Strongtalk* preprocessor produces typesafe *Smalltalk-80* code that preserves indentation, comments, and type annotations.

3.2 Expressiveness

Smalltalk is an unusually flexible and expressive language. Any type system for *Smalltalk* should place a high priority on preserving its essential flavor.

Several of our design goals for *Strongtalk* involve providing better typing solutions for situations we have found awkward in other languages. One of these situations has to do with the fact that statically typed languages often have difficulty expressing useful return types for methods that may return values of variable type, or when several return values are necessary. A second such situation stems from the fact that often one needs to design classes that support the protocol of another class, without inheriting from it, and this can be difficult or impossible to do in *name-based* type systems if you do not control the source code for that class, or if you are working in a single-inheritance language. A third shortcoming of all widely available statically-typed OO languages is extremely awkward support for higher-order messages such as iterators and other user-defined control structures.

Strongtalk is designed to circumvent these sorts of awkward situations. In the first and third situations described above, we take advantage of the simple but powerful combination of strongly-typed blocks, and polymorphic signatures for the higher-order methods that take them as arguments. Examples of this technique will be presented in the experience section.

To deal with the second problem described above, *Strongtalk* has a *structural* type system. In *Strongtalk*, an object can be passed anywhere as long as its static type is known to support the necessary protocol, regardless of its class. A way of explicitly defining protocols, and enforcing class support of them, is also provided.

3.3 Type Inference

Some recent languages, such as *TS* [GJ90], and *ML* [MTH90] include features to automatically infer variable and return types. Such features would be convenient in a type system for a language like *Smalltalk,* which lacks any sort of type annotations. However, *Strongtalk* does not perform general type inference.

There are number of reasons for this. In a language like *ML,* which has no concept of subtyping, type inference can be well-defined for any expression. However, for object-oriented languages type inference is considerably more complicated. A well-defined language specification that includes type inference should be able to either guarantee that all types can be correctly inferred, or provide a clear statement of what expressions

can have their types inferred, and which cannot. State-of-the-art object-oriented type inference algorithms that meet these criteria either require non-local code analysis to do complete type inference [PS91a, PS91b], or are not capable of dealing with realistic programs [Hen91].

Another consideration affecting our decision to avoid type inference is that at best type inference is little more than a convenience when writing a program, and at worst a program for which many types are inferred can be harder to read, since type annotations clarify the source text. Programs are more often read than written.

A browser that can infer types and then insert them into the source text would solve this problem, but in general for structurally-typed languages the inferred types can be complex, anonymous interface signatures that are not suitable for human consumption [Cur90].

3.4 Encapsulation

One of the major reasons we have adopted a requirement of 'downward' rather than 'upward' compatibility is that it allows us to enforce a much stronger notion of encapsulation in our type system. As mentioned in the introduction, it is likely that encapsulation issues will become much more important as *Smalltalk* continues its move from the research to the production world.

By encapsulation we mean that modifications to the implementation of a class that do not affect its public or subclass-visible behavior should not cause code elsewhere to break (where behavior includes both the typed message interface and external semantics of the class). When different organizations depend heavily on one another's class libraries, this is obviously a desirable property. While semantic specification techniques are not nearly mature enough to enforce full semantic encapsulation, an appropriate type system can prevent important kinds of encapsulation violation, at the cost of imposing a more disciplined structure on inheritance hierarchies.

Although from an encapsulation perspective this is obviously desirable, all dynamically-typed object-oriented languages, and many statically-typed ones, either cannot detect such encapsulation violations, or intentionally allow them.

The reason why such code is allowed in some statically-typed languages is that there is an important and fundamental tradeoff between encapsulation and certain forms of code reuse via inheritance. Non-local code analysis can determine that some kinds of dangerous interface changes during inheritance (such as removing support for a message, or arbitrarily changing the type of a message argument) are valid as long as the current pattern of instance usage indicates that such interface changes will not create type-safety loopholes.

There are a number of problems with such non-local type analysis from an encapsulation perspective. First of all, it means that changes in the *bodies* of methods outside of a class definition can cause it to become invalid. It also means that in general the source code for the entire system must be available. This can be a major problem in commercial environments, where it is important to be able to determine the typesafety of changes to a class that is used across many organizations, before release.

In addition, non-local type analysis introduces significant engineering difficulties into the typechecker design. For example, changing the body of a method in a superclass requires re-typechecking the method in the context of every subclass that inherits the method. Such requirements complicate the task of producing an implementation responsive enough for interactive, incremental development.

From the time the encapsulation/unrestricted-inheritance tradeoff was first recognized until fairly recently it has been considered an unavoidable result of the subtle distinction between subclass and subtype relationships, because it seems to occur in situations where it is handy to define a subclass

whose instances are not 'substitutable' for superclass instances.

However, recent work beginning with [CCH+89] has shown that there are forms of non-subtype-compatible inheritance that do *not* violate encapsulation. *Strongtalk* supports a form of such inheritance, F-bounded quantification, through the mechanism of inherited types. Future work in specifications may produce other such forms of inheritance. The architecture of the *Strongtalk* typechecker, because it does not *assume* subtype compatibility between a class and its superclass, should be well-positioned to support such extensions.

To put this into perspective, Figure 1 shows where various object-oriented languages fall on the spectrum of inheritance flexibility (by inheritance flexibility we are referring specifically to the kinds of changes allowed to message signatures during inheritance). At the top of the table are languages that allow methods to be added and overridden in subclasses, but which allow no changes of any sort to a method interface in a subclass. At the bottom of the table are dynamically-typed OO languages, which allow any sort of interface change, including removal of messages. Languages in all categories except for the bottom one are typesafe languages. The boundary above the second category from the bottom represents a conceptual divide below which a typesafe language cannot go without violating encapsulation. Our goal has been to place *Strongtalk* at a point as low in this table as possible, without crossing that divide.

Although the above design issues must inevitably be assessed differently for different development needs, we feel that a type system with *Strongtalk*'s characteristics will be well-suited to a very large class of production applications.

Message signature changes allowed during Inheritance	Language
No changes are allowed	C++, Modula-3
Changes that are subtype compatible are allowed	Trellis
Changes that are inherited type compatible are allowed	Strongtalk
Future type systems ?	
Changes that can be shown to be typesafe with non-local type analysis are allowed	TS, Eiffel
All changes are allowed	Smalltalk, Lisp

Figure 1: Languages categorized by inheritance flexibility

4 The Type System

4.1 Protocols

In *Strongtalk*, the type of an object is known as a *protocol*. This term is often used informally, within the *Smalltalk* community, to denote the set of messages to which an object responds.

A *Strongtalk* protocol refines this traditional notion in several ways. In addition to the names of the messages, the types of their arguments and the types of the objects they return are included, by means of type annotations, given between angle brackets. We shall use the term *interface* to describe this collection of messages and type information. A protocol has additional structure, which we discuss in section 4.3. However, the interface of an object is always determined by its protocol. When the distinction between interface and protocol is not essential to the exposition, we will use the general term *type*.

Figure 2 shows PlanarPoint, a protocol for points in the plane. To simplify exposition, we use a form of pseudo-code, not the concrete syntax of *Strongtalk*. The protocol supports five messages; four for instance variable access, and one to take the sum of two PlanarPoints. The caret (^) is used to distinguish the declaration of the message return type. The special type identifier Self is interpreted as meaning the protocol of the receiver.

Figure 2 also shows a class, BasicPlanarPoint, that implements the PlanarPoint protocol. In fact, we did not have to define the PlanarPoint protocol explicitly. For every class, *Strongtalk* also implicitly defines a protocol which describes the instances of the class. The full name (which may be shortened) of this implicit protocol is the name of the class, followed by the word protocol; in this case BasicPlanarPoint protocol.

```
protocol PlanarPoint
    x ^<Integer>.
    x: <Integer> ^<Integer>.
    y ^<Integer>.
    y: <Integer> ^<Integer>.
    + <Self> ^<Self>.
class BasicPlanarPoint
    instance var x <Integer>.
    instance var y <Integer>.
    class methods
        new ^<Instance>
            ^super new init.
    instance methods
        init
            x := y:= 0.
        x ^<Integer>
            ^x.
        x: xval <Integer>
            x := xval.
        y ^<Integer>
            ^y.
        y: yval <Integer>
            y := yval.
        + p <Self> ^<Self>
            ^(self class new x: self x + p x)
                        y:self y + p y.
```

Figure 2: Points in the plane

```
class SpatialPoint
    subclassOf:BasicPlanarPoint
    instance var z <Integer>.
    instance methods
      z ^<Integer>
        ^z.
      z: zval <Integer>
        z := zval.
      + p <Self> ^<Self>
        ^ super + p z:self z + p z.
```

Figure 3: Points in space

4.2 Subtyping and Type Inheritance

Having introduced some basic terminology and conventions, we now move on to a short review of the notions of *subtyping* and *type inheritance.* An example will help make this clear.

Figure 3 shows SpatialPoint, a subclass of BasicPlanarPoint. SpatialPoint adds a z coordinate, and modifies its methods to reflect the semantics of addition for spatial points.

The type SpatialPoint protocol is not a subtype of BasicPlanarPoint protocol. Subtyping means that members of a subtype can always be substituted where members of a supertype are expected. If one invokes + on a BasicPlanarPoint, one can pass in another BasicPlanarPoint as the argument. By contrast, + on a SpatialPoint demands a SpatialPoint, so that the method can access the z coordinate of the incoming parameter. Therefore, one cannot use a SpatialPoint where a BasicPlanarPoint is expected. Nevertheless, the two protocols share a similar recursive structure. This relationship is known as *type inheritance*, and we say that SpatialPoint protocol is an inherited type of BasicPlanarPoint protocol. We refer the reader to the literature [Bru92, Bru93a, CHC90, CCH+89, CCHO89] for a deeper discussion of the meaning of type inheritance.

4.3 Instance, Self class and the Nature of Protocols

Smalltalk is a reflective system. One consequence of this is that classes are themselves objects, which can send and receive messages. Furthermore, users can define methods not only for the instances of classes, but for classes themselves.

The definitions of instances and their classes are always mutually recursive in *Smalltalk.* A class always supports at least one method for creating new instances. Conversely, all instances support the method class which returns their class[1]. Hence, the interfaces of instances and their classes must also be mutually recursive.

The method new in figure 2 illustrates the treatment of class methods in *Strongtalk.* The method's return type is the special type identifier Instance. Within a class method or message, Instance refers to the interface of *instances* of the receiver. This is distinct from Self, which always refers to the interface of the receiver itself; in a class method, Self is the interface of the class, not of its instances. Similarly, in the context of instance methods, the type Self refers to the interface of the receiver, while the type Self class refers to the interface of the receiver's class.

The body of the method invokes the new method inherited from the superclass, using the standard *Smalltalk* super construct. Since this method also returns Instance, we know that in fact, the returned object is an instance of the receiver. Instances of the receiver are always inherited types of BasicPlanarPoint. It is therefore safe to send the init message, which is supported by BasicPlanarPoint. The init method returns an object of type Self. Since this is an instance method, Self is the interface of the instance, which is an instance of the receiver, and therefore has type Instance. As a result, the entire method is well-typed.

During inheritance, the interface of both instance and class may change. Notice that changing the interface of one automatically induces a change in the interface of the other. This means that for purposes of inheritance, the interfaces of instance and class cannot be separated, but must be packaged together.

A protocol is in fact such a package. It represents a pair of (mutually recursive) interfaces, one for instances, and one for the class. One of these interfaces is distinguished as the primary interface. An object whose type is a protocol, responds to the messages given by the protocol's primary interface. In the protocol of a class' instances, the interface of instances is considered primary. In the protocol of the class itself, the class interface is primary.

[1] We will ignore the possibility of overriding this method to do something completely different.

Mathematically, a protocol is a generator for a pair of interfaces. This generator is then used for type inheritance, in a manner analogous to the use of generators for value inheritance [CHC90]. When used as the type of an instance, we interpret the reference to the protocol to mean selecting the first (primary) interface from the fixpoint of the protocol. The fact that types are interpreted as both generators and interfaces, according to context, has been noted in [BH91].

4.3.1 The Metaclass Hierarchy and its Typing

In this section, we briefly review the notion of metaclasses in *Smalltalk*, and present our treatment of the typing of metaclasses. Before delving into the *Smalltalk* metaclass hierarchy, readers should be aware of two facts. First, this section can be ignored without loss of continuity. Second, the topic is recognized as being difficult for those new to *Smalltalk* [BO87].

In *Smalltalk* every object is an instance of some class. Since classes are objects, it follows that each class must itself be an instance of a class. Indeed, each class is an instance of its own *metaclass.* For example, Object is an instance of the metaclass Object class. Object class is then an instance of Metaclass, which is an instance of Metaclass class, which is an instance of Metaclass. This circularity prevents an infinite regress of meta-meta-... metaclasses.

The type structure of an instance, its class and metaclass is as follows. Instances of a class X have type X protocol. The object representing the class at run time has a distinct type, since it may support specific methods defined by the user as class methods in the definition of class X. This type is X class protocol. Sending the class message to an instance of X yields an object of type X class protocol. Sending this object the message class, yields the metaclass (class of a class) X class. All metaclasses are considered to have the same type, Metaclass protocol.

4.4 The Status of nil

In *Smalltalk,* nil denotes a reference to an undefined object. The value of an uninitialized instance variable is always nil. Sending a message to nil typically results in a run-time error, and this is a common cause of program failure. Unfortunately, determining that all instance variables are properly initialized requires expensive, non-local dataflow analysis.

We consider statically detecting invalid access to nil in a language with pervasive aliasing such as *Smalltalk* as beyond the purview of the type system. If desired, a separate dataflow analysis tool (e.g., [PS91b]) may be used for such purposes.

4.5 Brands

A disadvantage of structural typing is the risk of semantically incompatible objects sharing a syntactic type, and not being distinguished by the type system. To rectify this situation, we wish to support a notion of type identity, within the framework of structural typing. This can be accommodated using the concept of *brands*, as introduced in *Modula-3* [Nel91].

Brands are merely tags added to types to distinguish them from one another. To prevent accidental matching, brands are supplied only by the system (unlike *Modula-3*).

To create multiple implementations of a branded protocol users may declare that a class supports that particular protocol. This has the effect of introducing the protocol's brand into the type of the class' instances. In addition, the system will verify that the class does indeed provide methods that implement the messages specified by the protocol, with the correct type signatures.

```
generic protocol List[T]
    add:<T>.
    head ^<T | Nil>.
    tail ^<Self | Nil>.
    map:<Block[T,^S]> ^<List[S]>
        where S :: (actual arg:1) returnType
```

Figure 4: A generic class.

4.6 Blocks

Blocks are objects, like everything else in *Smalltalk*. They respond to messages such as value, which causes a block with no arguments to be evaluated. The precise protocol for a block depends upon the number of arguments it takes, their types $A_1, \ldots, A_n$, and the return type, R. In *Strongtalk* the type of such a block is written $Block[A_1, \ldots, A_n, \hat{}R]$.

Blocks are used as the basis of all control structures in *Smalltalk*. One peculiarity of blocks is that blocks that take no arguments and return a boolean support a special protocol. This allows their use as a basis for control structures like while loops. *TS* [GJ90] uses a special mechanism, *specific receivers,* to handle such cases. We prefer to provide a special type rule for such blocks. This solution is possible, because, unlike *TS,* we do not need to typecheck the class that defines blocks in the Smalltalk library (see section 5.4).

4.7 Generics

A *generic* is an abstraction over types, such as the generic protocol shown in figure 4. List is a *generic protocol*. It describes the type of linked lists with elements of any type T. The head message is intended to return the first element of a list. If the list is empty, its first element is undefined, and so the type returned is a *union* of the element type T and Nil, the type of the undefined object. This will force users to test for an undefined object dynamically, as described below. Similar comments apply to tail.

Strongtalk provides a typecase construct for dynamic typechecking. As noted in [ACPP89], the ability to dynamically determine the type of an object is essential in many circumstances, but in a structural type system this can be expensive. In practice, most protocols are branded, and the handling of these can be speeded up significantly.

The message map: utilizes parametric polymorphism, and is described in section 4.8.

A generic may be invoked by passing it *actual type parameters*, e.g., List[Integer]. The actual parameters replace the formal ones (T in the example) in the body of the generic. An invocation of a generic protocol yields a protocol equivalent to the definition of the generic, substituting the actual parameters for the formal ones. Special care must be taken to prevent recursive generics from leading to infinite expansion. The algorithm used is essentially the one developed for POOL [Ame90].

Strongtalk also supports the definition of generic classes. Many of the key classes in the *Smalltalk* library, such as Array, are generic. As noted in section 4.3 above, classes are objects in *Smalltalk*. Every object has a type, which raises the question 'what is the type of a generic class, such as Array?'.

Formally, the type of Array is a universally quantified protocol. We do not presently have type expressions for such a type in the language. We feel our users may have difficulty grasping such types. Instead, we require that all references to a generic class be invocations (e.g., Array[String]). Such an invocation yields a class. In fact, all invocations of Array yield the very same object. However, the type of a generic class invocation is an ordinary protocol. Cases where this rule is too restrictive almost never arise in application programming.

4.8 Parametric Polymorphism

Object-oriented programming typically relies on *inclusion polymorphism*. In addition to inclusion polymorphism, *Strongtalk* supports a form of *parametric polymorphism* similar to that of [CW85].

Parametric polymorphism is useful in cases where the type returned by a message send is dependent on the type of the actual arguments, a common occurrence in *Smalltalk*. The map: message in figure 4 demonstrates the use of parametric polymorphism. The map: message takes an argument which is a block. The block is to be invoked successively on each element of the list, and a list of the results is to be returned. The block must take as input an element of same type as the elements of the list. The type of the elements of the list being returned depends upon the return type of the block. The type declaration for map: specifies this dependency in terms of a type variable, S. The ordinary message declaration is suffixed by a where clause that states that S is to be derived from the return type of actual argument number 1. A special syntax is used, allowing precise specification of how to infer the type arguments from the types of the value arguments. Map: can be invoked as an ordinary method, and there is no need to pass a type argument explicitly at call sites. This syntactic treatment of explicitly polymorphic methods is somewhat novel. Its advantage is that it allows us to infer actual type arguments without implementing a sophisticated type inference mechanism. While some may find the syntax awkward, it places the syntactic burden on the (relatively infrequent) message declaration, rather than the more commonly used message send construct.

5 Experience

A *Strongtalk* implementation has been completed that incorporates all the features mentioned in this paper. A significant body of code has been written using it, including the *Strongtalk* implementation itself, and several other sizeable libraries.

The usability of a language or type system is always difficult to assess before its implementation is completed. Following are some of the interesting issues that have emerged from our use of *Strongtalk*.

5.1 The value of Blocks

Proponents of functional programming styles have long argued that the combination of higher-order functions and literal first-class functions is enormously expressive. Block objects are *Smalltalk*'s equivalent construct. Blocks are used for many things in *Smalltalk*: defining extensible control structures, callbacks, simple exception handling (e.g., Dictionary at:ifAbsent:), etc.

We have encountered a surprisingly handy interaction between blocks and static typing that mitigates much of the awkwardness we have encountered when using other statically-typed pure object-oriented languages. Blocks, in conjunction with *Strongtalk*'s polymorphic messages, provide a number of convenient idioms (shown in figure 5) that *Strongtalk* can easily type.

All of these things can be done easily in *Smalltalk*, but can be difficult to do cleanly or efficiently in statically-typed languages without blocks. The first two examples are easy to type in *Strongtalk*; the second two require polymorphic messages for full type safety. Figure 6 shows the signature for the at:ifAbsent: message from the Dictionary class.

The return type of the at:ifAbsent: message is a union type <VALUE | X>, which means that the method returns either a value from the dictionary, or a value of type <X> which is the return type of whatever block the caller passes in for blk. In this example, the block never yields a value, since it forces its containing method to return; so the return type of the block is the invisible type DoesntMatter that *Strongtalk* gives to expressions that can never yield a value, which allows our example to type correctly.

```
* Alternative or multiple return types

   dataset computeStats:
       [ :median <Double> :avg ¡Double>
                 :size ¡Int> |
         Transcript
             show: median printString;
             show: avg    printString;
             show: size   printString;
             cr.
       ]

* Blocks instead of perform: for callbacks

   button whenPressedDo:
[Transcript show: 'Yow!'; cr].

* Control structures as value-yielding expressions

   sky := daytime
       ifTrue: [#light]
       ifFalse: [#dark].

* Simple exception handling

   Transcript show: (myDictionary
       at: key
ifAbsent: [^self]  ).
```

Figure 5: Block idioms

```
       at: key <KEY>
       ifAbsent: blk <Block[^X]>
       ^<VALUE | X>
             where X :: (actual arg: 2) returnType
```

Figure 6: Dictionary[KEY,VALUE] at:ifAbsent: signature

It should be emphasized that we are not simply saying that blocks are nice. The primary point is that having well-typed blocks in a strongly-typed language turns out to provide a very convenient way of expressing solutions to problems that can otherwise be complicated by a type system. For example, without such a facility, a method that wants to return either a number or an error symbol, might otherwise require a clumsy and less reliable dynamic type check.

5.2 Naming issues with separate types and classes

One of the problems with separate type and class hierarchies has to do with the potential for a significantly larger resulting namespace, because of separate names for classes and their protocols. In practice, the namespace is indeed somewhat larger, but we have found that not only is the increase easily manageable, it is very desirable, since the new names tend to be for important interfaces that are only implicit in dynamically typed hierarchies, and which deserve far more explicit attention during both the design and library browsing processes.

We have developed a few rules of thumb that seem to work well as guidelines for assigning names for those protocols that need to be distinct. First of all, having a pithy name for the protocol is more important, since it will be widely used in declarations, whereas the class name is generally used only at the instantiation point, which may well be hidden behind encapsulation barriers from the user of the instance.

It is perfectly acceptable to simply use the protocol automatically defined by a class if it seems unlikely that the protocol will be supported by other (non sub) classes. In *Strongtalk*, this does not *prevent* other classes from supporting that protocol, because of structural typing. It can, however, cause some confusion if the protocol will have many alternative implementations, so if it appears at all likely that other implementations will exist, pro-

vision for a separate name for the protocol is encouraged. This can be accomplished by declaring a separate protocol, or simply by creating a type alias for the automatic protocol.

5.3 Prototyping with types

A commonly heard objection to strongly-typed object-oriented languages is that having to worry about the types of things gets in the way during the prototyping process. Most of these opinions are based on little or no information, since very few people have ever had a chance to use a statically-typed language in an incremental programming environment. Although such matters are difficult to analyze precisely, in our experience the opposite has appeared to be true.

Even a prototype must function in some fashion, and this requires design and debugging work just as with any program. Although some of the errors a typechecker will find are ones that would be encountered during testing, catching even these types of errors at compile time can save quite a bit of time. For example, if a method is called with an argument of the wrong type, the object is often passed around to a remote part of the system before a message that is inappropriate for its type is actually sent to it. Fixing this during testing requires a tedious backtracking effort to find out where the original error occured. A typechecker detects these errors at the place they occur.

At a higher level, focusing on typing issues even at the earliest stages of development can help enormously in clarifying the interfaces and responsibilities which drive the design process. The larger the project, the greater the benefit.

We have also encountered a somewhat more subtle benefit of strong typing and stronger encapsulation that may well in the long run turn out to have an even greater impact. It has become well accepted that one of the most difficult parts of the object-oriented learning curve to climb is understanding large class libraries. Furthermore, this problem reoccurs whenever a new library must be learned.

In the authors' personal experience with the standard *Smalltalk* class hierarchy, one of the largest impediments to understanding is the difficulty in finding out exactly what argument values methods allow, and what values they return. Comments are not enough! Typing information provides an invaluable form of machine-verifiable documentation. Browsers that understand the type system could also include powerful features that would allow a programmer to point at any variable or expression and ask 'Show me what I can do with this object at this point.'

An even more important point is that although the inheritance discipline imposed by the Strongtalk type system can be viewed as restrictive, it is restrictive in a way that contributes greatly to the comprehensibility of a class hierarchy. Strict interface inheritance relationships are a critical first step towards the larger goal of strictly enforced semantic relationships. Hierarchy designers who adhere to these stronger disciplines will earn the heartfelt thanks of those who must understand and reuse their code.

5.4 Typing the standard *Smalltalk* library

Obviously, if *Strongtalk* is to be compatible with standard *Smalltalk* environments, we must have a way of dealing with the types of objects in the existing libraries. Because of its strict encapsulation, *Strongtalk* does not need access to the implementations of those libraries; we need only declare their protocols. Cook has pointed out [Coo92] that although there are a number of areas where the *Smalltalk* class hierarchy does not have subtype relationships, it in fact implicitly contains the incomplete skeleton of a clean subtype lattice. Our original protocols for the *Strongtalk* libraries were somewhat similar. Subsequently, we have adopted the protocols from Cook's corrected Collection hierarchy because of their stronger property of semantic

substitutibility.

An interesting benefit of the fact that Strongtalk can typecheck based on purely local information is that it significantly eases integration with existing Smalltalk classes. A class declaration is specified, and this supplies all the information needed by Strongtalk. Typecheckers based on non-local information are faced with a much more difficult task when trying to integrate with non-typechecked classes (and this is necessary, since no typechecker has ever succeeded in typechecking all the classes in the existing Smalltalk hierarchy). Without the ability to typecheck all superclass source code, non-local typecheckers cannot safely allow the very flexibility they were designed to preserve.

6 Related Work

6.1 Typechecking Smalltalk

There have been several efforts to introduce static typechecking (or type inference) to *Smalltalk*. These efforts have had differing objectives, resulting in different approaches to the problem.

Early efforts met with limited success [Suz81, BI82]. *TS* [JGZ88, Gra89, GJ90] is geared toward typechecking *Smalltalk* in its entirety, and toward optimization. The scope of *Strongtalk* is less ambitious in this respect. Certain *Smalltalk* programs that are accepted by *TS* will not be accepted by *Strongtalk*. The advantage of our approach is the preservation of encapsulation. In *TS,* inherited code is retypechecked in the context of the inheriting class, with all the concommitant advantages and disadvantages outlined in section 3.4.

Palsberg and Schwartzbach [PS90, PS91c, PS91a, PS91b] developed a system that infers types for *Smalltalk* without relying on type annotations at all. The system is useful for analyzing a completed application prior to final release, to shake out remaining type errors, and can provide useful information for optimization. The system relies on global analysis, however, and is not well suited to incremental development.

6.2 Inherited Types

The notion of inherited types has appeared in various related forms in several object-oriented languages. Emerald [BH91] incorporates a similar notion called *type matching*. Emerald does not support class inheritance however. *POOL* [Ame90] is the imperative language most similar to *Strongtalk* in its type system.

The theoretical foundations for inherited types were first given in [CCH+89] and [CHC90]. The language *ABEL* [CCHO89] incorporated these ideas in a functional framework. Our type system is mainly based on the type rules given by Bruce for the language TOOPL [Bru93a]. The differences between the two systems reflect their different purposes. TOOPL is a theoretical language designed to facilitate fundamental understanding of object-oriented languages and their typing. Its type system has been proven sound (and for TOOPLE [Bru93b], a close relative, typechecking has been proven decidable). We have not formally proven such properties for our system. In contrast, we have integrated the rules into a realistic, imperative language (TOOPL models instance variables functionally). We have added support for abstract classes, generics, dynamic typechecking, multiple levels of visibility, and explicit polymorphism.

7 Conclusions

We have described *Strongtalk*, a strongly-typed dialect of *Smalltalk*. This dialect is essentially compatible with the standard *Smalltalk-80* language. The type system differs from other efforts to typecheck *Smalltalk* in that it preserves encapsulation. Encapsulation is crucial in large scale applications, and also helps the system perform incremental typechecking quickly. As a result, the advantages

of the incremental programming environment may be more easily combined with those of static typechecking.

The type system demonstrates the practical utility of recent advances in the type theory of object-oriented programming languages. The notion of inherited types has been integrated into a full-fledged imperative language, including generics, polymorphic methods, a practical approach to typing metaclasses, and dynamic typechecking. This spectrum of language features has not been combined before.

There is a tradeoff between encapsulation and the flexibility of the type system. *Strongtalk* imposes a stronger discipline than other *Smalltalk* type systems. Nevertheless, *Strongtalk* supports a rich style of programming difficult to obtain in other languages with significant static typechecking. The tradeoff is valuable because it allows, for the first time, practical, production use of a strongly-typed *Smalltalk* system.

Acknowledgements

We would like to thank Pierre America for sharing his experience with generic types in *POOL*. Peter Deutsch offered valuable comments and encouragement. Finally, we wish to thank Ralph Johnson for his valuable comments on this work. Of course, the authors are solely responsible for the shortcomings of *Strongtalk*.

References

[ACPP89] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 213–227, January 1989.

[Ame90] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, pages 161–168, October 1990.

[BH91] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL91/1 (Revised), DEC Cambridge Research Lab, July 1991.

[BI82] Alan H. Borning and D. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 133–141. Association for Computing Machinery, 1982.

[BO87] Alan H. Borning and Tim O'Shea. Deltatalk: An empirically and aesthetically motivated simplification of the Smalltalk-80 language. In *European Conference on Object-Oriented Programming*, pages 1–10, 1987.

[Bru92] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, January 1992.

[Bru93a] Kim Bruce. Safe type checking in a statically typed object oriented programming language. In *Proc. of the ACM Symp. on Principles of Programming Languages*, January 1993.

[Bru93b] Kim Bruce. Typechecking in TOOPLE is decidable. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, September 1993.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture*

Notes in Computer Science, pages 51–68. Springer-Verlag, 1984.

[CCH+89] Peter Canning, William Cook, Walt Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[CCHO89] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 457–467, 1989.

[CHC90] William Cook, Walt Hill, and Peter Canning. Inheritance is not subtyping. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.

[Coo92] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 1–15, October 1992.

[Cur90] Pavel Curtis. *Constraint Quantification in Polymorphic Type Analysis*. PhD thesis, Cornell University, Department of Computer Science, 1990. Also available as Xerox PARC Technical Report CSL-90-1.

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

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 136–150, January 1990.

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

[Hen91] Andreas V. Hense. Type inference for O'Small. Technical Report A 06/91, Fachbereich Informatik, Universitaet des Saarlandes, October 1991.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Laurance W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 18–25, November 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[PS90] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[PS91a] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1991.

[PS91b] Jens Palsberg and Michael I. Schwartzbach. Static typing

for object-oriented programming. Technical Report DAIMI PB-355, Aarhus University, Computer Science Department, June 1991.

[PS91c] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse ? In *European Conference on Object-Oriented Programming*, July 1991.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 187–199. Association for Computing Machinery, 1981.