The Strongtalk Type System for Smalltalk

Gilad Bracha

August 4, 1996

1 Introduction

The benefits of static typechecking in software development are widely recognized. It is now feasible to introduce static typechecking into Smalltalk in a manner that does not compromise the flexibility of the language or the programming environment. This document describes *Strongtalk,* a type system well suited to this purpose.

The Strongtalk type system is designed to ease the development process by improving the reliability and readability of programs during development, maintenance and use. The type system provides an enforceable formalism for describing the interfaces of Smalltalk libraries. This formalism helps designers communicate to themselves and to others the interfaces of classes. In addition, Strongtalk helps catch errors earlier in the development process.

The type system is *not* concerned with improving execution performance, since it is based on interface types. Optimization requires concrete implementation type information. Agesen [Age96] has demonstrated that extracting concrete type information in Smalltalk-like languages does not require linguistic extensions. Therefore concrete types are not considered here.

Since Strongtalk provides a language for describing interfaces to humans and machines, it is by definition a language extension. However, use of the type system is completely optional.

The following section briefly describes the specifics of the type system. The later sections discuss the type system's interaction with other language features, its implementation and experience in its use.

2 The Type System

This document is too brief to give a complete specification of the Strongtalk type system. Here we can only provide an overview of Strongtalk's salient characteristics and a summary of its features. For a more detailed description (though somewhat out of date) see [BG93].

The key characteristics of the Strongtalk type system are :

- The type system is *expressive* enough to allow natural Smalltalk idioms to be typechecked. To accomplish this, Strongtalk:
  1. Separates the subtype and subclass lattices.
  2. Includes parameterized types and classes.
  3. Supports parametrically polymorphic messages, with a flexible mechanism for automatically inferring actual type parameters.
  4. Supports both subtyping and type matching.
  5. Preserves the subtype relations between classes defined by the Smalltalk metaclass hierarchy and relates them to the types of their instances.
  6. Provides facilities for dynamic typing.

class protocol. In type expressions, one can always elide the word protocol, and refer to the protocol of the metaclass as C class.

In a protocol, the type variable Self refers to the type of the receiver. Self class then represents the type of the class of the receiver.

In the case of class protocols, the receiver is a class, and we may wish to refer not only to the type of the receiver and the type of the receiver's class, but also to the type of the receiver's instances. This can be done using the type variable Instance.

Every protocol definition induces a class protocol representing the type of the class of objects which support the protocol. The class protocol hierarchy parallels the protocol hierarchy, just as the metaclass hierarchy parallels the class hierarchy. Thus, all class protocols inherit from Object class protocol, just as all metaclasses inherit from Object class.

2.1.2 Typed Methods and Expressions

Smalltalk is extended syntactically so that all variable declarations may be annotated with a type declaration. This includes temporaries, block and method arguments, instance, class, pool and global variables.

All methods have an associated return type. If no return type is explicitly declared then Self, the type of the receiver, is assumed. Methods may be declared to take type arguments. These are never passed in explicitly, but are inferred.

The visibility of a method may be either public (the norm) or private (in which case the method is only visible within the class and its subclasses). A private method does not appear in the classes' protocol.

Classes may include message declarations, indicating messages that must be supported but are not implemented. In practice, subclass responsibility methods may be used for this purpose. Classes may be declared *abstract,* meaning that they include message declarations. The system can then ensure that classes that are not abstract implement all their message declarations.

2.1.3 Genericity

Definitions such as classes and protocols can be declared to be *generic.* A generic abstracts over one or more types. Typical examples are the classes in the Collection hierarchy:

```
Collection[T] {Object subclass ...}
Set[E] {Collection[E] subclass ...}
```

A generic definition introduces one or more formal type arguments. These variables can be used in the scope of the definition. To be used in a type safe manner, a generic must be invoked with actual type arguments (e.g., Collection[Character], Set[Boolean], Dictionary[Symbol, Integer]). Occurrences of the formal type arguments are replaced with the corresponding actual type arguments. A bound can be specified for each formal type argument T. If no bound is given, T is assumed to be a subtype of Object.

2.1.4 Block Types

A special syntax is provided for the types of blocks. [^Symbol] is the type of a block with no arguments that returns a symbol. [Character, ^ Integer] is the type of a block that takes a single character argument and returns an integer. [Boolean, Integer, ^Boolean] is the type of a block taking two arguments, a boolean and an integer and returning a boolean result.

[X, ^Object] can be written more concisely as [X]. Similarly [Boolean, Integer, ^ Object] = [Boolean, Integer]. [^Object] can be contracted to [].

2.1.5 Union Types

One may specify the type of an object as the union of several object types, (e.g. Symbol | UndefinedObject).

2.1.6 Parametric Polymorphism

In some cases the type of a method's result may depend on the type of its arguments. To accurately express the signatures of such methods one may use *parametric polymorphism*. A parametrically polymorphic method signature introduces one or more formal type arguments which can be used throughout the signature and the method body. When the message is sent, the actual types will be inferred and replace the formal type arguments in the signature. As an example, consider the signature of Collection[T]>>collect: :

```
collect: blk <[T, ^X def]> ^ <Collection[X]>
```

The type term X def introduces a type argument X, which will be inferred to be the type returned by the block which must be the first argument to the method. A typical usage would be

```
| nums <Collection[Integer]> |
nums := 'Fee Fi Fo Fum' collect:[:c <Character> | c asciiValue].
```

Here X will be inferred to be Integer, and the type of the send is Collection[Integer] as expected.

2.1.7 Relationships among types

Strongtalk supports two relations on types: *subtyping* and *matching*. Subtyping implies substitutability: an element of a subtype can be safely used in wherever an element of supertype is expected. Matching implies a common pattern of self reference.

Originally, Strongtalk relied on a structural type system augmented with brands. The current version abandons brands and uses declared relations to determine subtyping and matching.

Every protocol definition specifies its supertypes. Likewise, a class definition can specify the supertypes of its protocol. By default, a protocol is declared to be a subtype of its superprotocol.

Naturally, every declared type relation is verified by means of a structural check.

Subtyping among invocations of a given generic is declared as follows. Each formal type argument has a *variance* which determines how the type of an invocation varies with the type of the actual type argument supplied. If the formal is *covariant* (the default), then

T <= S $\Longrightarrow$ G[T] <= G[S].

If it is *contravariant* then T <= S $\Longrightarrow$ G[S] <= G[T]. If it is *unrelated* then no subtype relation is declared between G[T] and G[S].

Subtyping of block types is contravariant on the argument types and covariant on the range type.

2.1.8 Essential Type rules

Message send The message selector must be defined for the type of the receiver. The types of arguments in a message send must be subtypes of the types of the formal parameters. The type of the send is the return type of the message signature being invoked.

Assignment The type of the right hand side of the assignment must be a subtype of the type of the variable being assigned. The type of the assignment is the type of its right hand side.

Return statement The type of the expression being returned must be a subtype of the declared return type of the enclosing method. The type of the return statement is DoesNotMatter, which is a subtype of any type.

Method body The return type of a method that does not execute a return must be a supertype of Self.

Subclassing A method signature in a subclass must always be a matchtype of its signature in the superclass. The types of variables cannot change. Inherited public methods cannot be privatized.

3 Interaction with Existing Language Features

Since the type system is optional, the extension does not interact with existing language features. Strongtalk is both upward compatible with Smalltalk (since any Smalltalk program is a Strongtalk program) and downward-compatible in the sense that any Strongtalk program can be trivially converted back into a Smalltalk program by stripping out the type annotations.

4 Implementation Issues

Strongtalk can be integrated into an existing implementation by modifying the programming environment so it can parse and typecheck type-annotated programs. No changes are needed to the underlying compiler or runtime because the type system is optional.

5 Implementation Status and Experience

An implementation that integrated Strongtalk into an existing browser was demonstrated at OOPSLA 93, and used in production in the financial industry. A second generation implementation has since been developed using completely new browsers that take fuller advantage of the type system.

Strongtalk has been used to write several large libraries, including a complete blue book base library implementation. The implementation has the highly desirable property that it imposes no overhead unless the typechecker is actually invoked. When the typechecker is used, memory overhead is small, and incremental response is good.

Experience using Strongtalk has supported most of the assumptions of the original type system design. In particular:

- Even if a library is not typechecked, typechecking its clients is feasible and beneficial, as long as a library interface has been defined.
- Type annotation enhances readability of code.
- Typechecking can actually ease the development process. It detects many bugs that otherwise show up during debugging, but faster. Also, the type system occasionally detects problems that lurk in working code for extended periods of time.

The most important change introduced into Strongtalk is the move from structural to declaration based subtyping. This change was made because:

- Structural type descriptors do not express programmer intent, and are difficult to read.
- Error messages produced by structural checking are poorly localized and very hard to understand. This is especially true in a large system with lots of internal cross references like Smalltalk.

References

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, January 1996. Also available as Sun Labs technical report SMLI TR-96-52.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, September 1993.