# PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language*

*Extended Abstract*

Kim B. Bruce, Angela Schuett**, and Robert van Gent***

Williams College, Williamstown, MA 01267

**Abstract.** PolyTOIL is a new statically-typed polymorphic object-oriented programming language which is provably type-safe. By separating the definitions of subtyping and inheritance, providing a name for the type of *self*, and carefully defining the type-checking rules, we have obtained a language which is very expressive while supporting modular type-checking of classes. The *matching* relation on types, which is related to F-bounded quantification, is used both in stating type-checking rules and expressing the bounds on type parameters for polymorphism. The design of PolyTOIL is based on a careful formal definition of type-checking rules and semantics. A proof of type safety is obtained with the aid of a subject reduction theorem.

**Categories**: Type systems, design and semantics of object-oriented languages.

## 1 Introduction

Because of the complexity of object-oriented languages, it has proven to be very difficult to design type-safe statically-typed object-oriented languages which are also very expressive. At one extreme we have statically-typed languages like C++ [ES90] and Object Pascal [Tes85] which come close to type safety, but whose rigid and inflexible type systems result in the need for type casts in order to express the programmer's desires. At the other extreme are untyped or weakly typed languages like Smalltalk [GR83], which are prone to run-time errors of the form "message not understood." Somewhere between these extremes are languages like Eiffel [Mey92], which provides a conservative link-time "system validity check" to pick up errors missed by the type checker, and Beta [MMMP90], which builds in run-time checks to pick up errors which cannot be detected reliably by the type-checker.

* This research was partially supported by NSF grant CCR-9121778. The full paper is available via anonymous ftp from cs.williams.edu in pub/kim/PolyTOIL.dvi or on the world-wide web through http://cs.williams.edu/~kim.

** Current address: Department of Electrical Engineering and Computer Science, University of California at Berkeley.

*** Current address: Department of Computer Science, Stanford University.

The language PolyTOIL is the culmination of a series of design efforts to build a type-safe language based on progress in the theoretical understanding of object-oriented languages. In [Bru93], we introduced a statically-typed, functional, object-oriented language, TOOPLE. This and subsequent papers [Bru94, BCK94, BCD+93] provided typing rules and both denotational and operational semantics for the language. These papers included proofs of the relative consistency of the operational and denotational semantics, a subject-reduction theorem, the type safety of the type-checking rules, and the decidability of the type-checking problem.

Because most object-oriented programming languages are imperative, it was clearly desirable to extend this work to create an imperative object-oriented language. The language TOIL [BvG93, vG93] was designed to satisfy this goal while retaining the same nice features as TOOPLE. The extension of TOIL to PolyTOIL is obtained by adding an unusual form of bounded polymorphism which provides a very flexible yet safe language for object-oriented programming. In this paper we report on the design, type-checking rules, and semantics of PolyTOIL.

A minimal list of features that should be supported in any object-oriented programming language includes:

- **Objects**, consisting of both state and operations.
- **Classes**, to generate objects (though delegation would be a reasonable alternative).
- **Message sending**, as a way of specifying computation.
- **Instance variables**, representing state.
- **Subclasses**, to support reuse of the instance variables and methods of an existing class in defining a new class.
- **Subtypes**, depending only on object interfaces, to provide a mechanism for programmers to use an object of one type in a context which expects one of a different but related type.
- Keywords, *self* and *super*, representing the receiver of a message and its superclass, respectively.

Other desirable features supporting reuse include:

- **Information hiding**, ensuring that applications do not depend on the implementation details of objects.
- **Parameterized types and (bounded) polymorphic functions**, allowing a programmer to abstract over a type in order to define a related family of types or operations.
- **Modular type-checking**, providing greater support for the reuse of class definitions by allowing subclasses to be type-checked with knowledge only of the type of the superclass, rather than its code. This provides support for separately compiled classes and eliminates the necessity of repeatedly type-checking methods when they are inherited from superclasses.

The language PolyTOIL satisfies all of these requirements and is provably type safe. PolyTOIL also has the following important features.

PolyTOIL treats classes as first-class values, allowing them to be values of variables, to be passed as parameters, and to be returned as values of functions. This simplifies many of the problems with initializing objects and provides greater flexibility in support of such things as parameterized subclasses. We also provide more flexibility in the definition of subclasses by allowing the programmer to change the types of over-ridden methods to be subtypes of the corresponding types in the superclass.

Separating the subclass and subtype hierarchies in PolyTOIL allows greater expressibility while providing type safety. We introduce a new relation on types that we call "matching" which corresponds more closely than subtyping to the subclass hierarchy. This relation, which is weaker than subtyping on object types, is very useful both in expressing the type-checking rules for classes and in determining when messages can be sent to objects. The major difference between our earlier languages, TOIL (for Typed Object-oriented Imperative Language) and PolyTOIL is the support for polymorphic functions. We express constraints on type parameters by requiring that a type match a given object type. As we shall see in the next section, constraining types using matching is much more useful that using the subtype relation.

While we have a number of results on the type system and semantics of PolyTOIL, in this paper we focus on type safety. We include and explain the type-checking rules as well as a natural (operational) semantics. The natural semantics is environment-based and corresponds closely to an interpreter which we have implemented for the language. We prove a subject-reduction theorem, from which follows the type safety of the system. One consequence is that the computation of a well-typed term will never result in sending a message to an object that it will not understand. The proof of subject reduction (which is included in the full paper) is interesting in that it applies to an environment-based natural semantics for a polymorphic language.

In Section 2 of this paper we discuss the design of PolyTOIL. We include an extended example which illustrates the flexibility of the language. In Section 3 we provide the formal syntax and type-checking rules for PolyTOIL. In Section 4 we provide the natural semantics and discuss the subject-reduction and type-safety theorems for PolyTOIL. In Section 5 we compare our development of PolyTOIL with other similar work on statically-typed imperative object-oriented programming languages. In the final section we discuss more recent work and future plans for PolyTOIL.

## 2 Language Design for Type Safety

We presume that the reader is familiar with the basic notions of object-oriented languages, including object, class, subclass, method, and instance variable. Recall that type $\sigma$ is a subtype of type $\tau$, written $\sigma <: \tau$, if an element of type $\sigma$ can be used in any context expecting an element of type $\tau$.

The distinction between subclasses and subtyping is illustrated clearly in [CHC90]. It is possible to have subclasses which do not generate subtypes, and

subtypes which do not arise from subclasses. Thus if one is interested in designing a statically typed object-oriented language, type safety will require either restricting subclasses to be those which generate subtypes or separating the subclass and subtype hierarchies. In the interest of maximizing expressibility we have chosen to separate these hierarchies.

Types in programming languages provide interfaces which can be used to determine whether certain operations or constructs are legal. They typically do not include semantic information. Classes in object-oriented languages include the bodies for methods as well as initial values for instance variables. Since this is semantic information, we conclude that it is more appropriate to think of classes as values than as types.

Because classes and objects are used in different ways (classes can be extended or modified to form subclasses or instantiated to create objects, while messages can be sent to objects in order to perform actions), it is important to give classes and objects different types. A class type, written $\mathit{ClassType}(\sigma, \tau)$, includes information on the types of the record of instance variables, $\sigma$, and methods, $\tau$, while an object type, $\mathit{ObjectType}\ \tau$, reveals only the types of methods.

Virtually all object-oriented languages include a construct *self* (sometimes named *this* or *Current*) which is used either explicitly or implicitly within a method to refer to the object executing the method. In order to statically type check a method in which *self* occurs, we must be able to determine its type. This is harder than it might appear at first sight.

Let $c$ be a class and $\mathit{ObjectType}\ \tau$ be the type of objects generated by $c$. If *self* occurs in method $m$ in class $c$, one might expect to assign *self* the type $\mathit{ObjectType}\ \tau$. However the method $m$ might also be inherited in a subclass $c'$ of $c$. In this new context, *self* represents an object generated from $c'$, which likely has a different type. If we wish to be able to type check the method $m$ only the first time it appears, and not repeatedly type check it every time it is inherited, we must do the type checking under weaker assumptions on the type of *self*.

In order to do this, we must know what changes are allowed in the types of methods in subclasses. As explained in [Bru94], to preserve type safety the body of a method $m$ with type $\tau_m$ in a class $c$ may only be overridden in a subclass with a new method as long as the type $\tau'_m$ of the replacement method is a subtype of $\tau_m$. This motivates the definition of the *matching* relation, $<\#$, between object types. $\mathit{ObjectType}\ \tau' <\# \mathit{ObjectType}\ \tau$ if for every method name, $m$, in $\tau$ there is a corresponding method in $\tau'$, and the type of $m$ in $\tau'$ is a subtype of its type in $\tau$. (An example of two types which match, but are not subtypes is given later in this section. Formal definitions of matching and subtyping are given in the next section.)

In order to accomodate the possible changes to the types of methods in subclasses, we introduce the special type, *MyType*, to represent the external type of *self* (where instance variables are hidden). In order to ensure that methods remain type safe when inherited in subclasses, methods are type checked under

the assumption that *MyType* <# *ObjectType* $\tau$.[4] By type checking under this assumption, we need not worry about the type safety of the method when it is inherited in subclasses.

Most statically-typed object-oriented programming languages (including C++ and Object Pascal) do not include a keyword like *MyType* to express the type of *self*, and instead are forced to assume its type is the same as the objects being defined by the class. In order to preserve type safety, the language designer must then restrict the user from changing the types of methods when they are overridden in a subclass. As a result it becomes impossible to properly redefine methods which take parameters or return a result whose type should correspond to that of the receiver.

Eiffel [Mey92] rejects this rigidity and includes a construct similar to *MyType* (which is written *like Current*). Unfortunately, the adoption of a covariant rule for parameters of methods and the identification of subclasses with subtypes leads to type insecurities [Coo89], which must be compensated for with a link-time "system validity check."

We avoid these problems while preserving much of the expressibility of Eiffel by separating the class and subtype hierarchies, providing more careful type-checking rules, and providing support for bounded polymorphism using matching. The result is a language that is provably safe, while being significantly more flexible than such statically-typed object-oriented languages as C++ and Object Pascal.

Before providing a complete example of a PolyTOIL program to illustrate the language features, we mention a few issues that arise when imperative features are combined with subtyping.

According to the definition of subtyping, it should be possible to assign to a variable an expression whose type is a subtype of that of the variable. That is, if $T <: U$, $e_T$ is an expression of type $T$, and $x_U$ is a variable of type $U$, one would like to allow the assignment $x_U := e_T$. As usual, we accomodate this in the run-time system by requiring that all objects be implemented as references. A variable of object type either contains a reference to the instance variables and methods of an object or contains a nil reference, which is denoted by the constant *nil*. *Nil* is considered an element of each object type, including *MyType*.

If $x$ is a variable holding values of type $\tau$, the type checker treats $x$ as having type *ref* $\tau$. Because these reference types are both receivers and producers of values, they have no non-trivial subtypes (see [Rey80]). Similar problems obstruct the use of subtyping with call-by-reference parameters. For instance, a procedure might take a formal parameter of type $T$ and assign it a new object of that type. If an actual parameter, $u$, of type $U <: T$ is passed in to the procedure, it will

[4] We also introduce a special type *SelfType* to represent the internal type of *self*. In this internal view, *self* has access to its instance variables. A built-in function, *close*, converts *self* from the internal to the external view. The implemented language PolyTOIL hides this distinction by automatically inserting instances of *self* and *close* where needed so that the programmer need not be concerned with this distinction or even know about *SelfType*.

be assigned a value of type $T$. This would create a hole in the type system which would show up if a message $m$ belonging to $U$ but not $T$ were sent to $u$ after the procedure returns.

To avoid adding special rules, PolyTOIL only supports call-by-constant-value parameters. It is illegal to assign to a formal parameter in a procedure or function. Of course, it is legal to send a message to such a parameter, requesting it to perform an action (which may result in a change to that object's instance variables). As a result, this mechanism, similar to that used in Eiffel, is more flexible than it might first appear.

PolyTOIL supports the definitions of parameterized types (functions from types to types) and polymorphic functions (functions from types to values). In order to provide finer control over the types which may be passed in as parameters, the language allows the programmers to constrain a type parameter to match a given object type.

To illustrate the expressiveness of PolyTOIL, a relatively sophisticated sample PolyTOIL program which defines and uses singly and doubly-linked lists is given in Appendix A. We point out a few interesting features below.

Parameterized types are represented in PolyTOIL as functions from types to types. For instance, *OrdList_type* and *OrdList_class_type* are functions which take a type, $U$, matching *Node_type*, and return an object and class type, respectively. The language also supports bounded polymorphic functions which return values. *OrdList_class* is a function which takes a type, $U$, matching *Node_type* and returns a class with type *OrdList_class_type[U]*. Bounded matching can be shown to be equivalent to F-bounded polymorphism [CCH+89], but is expressed in a simpler and more readily understood form.

As illustrated in the definition of *Node_class*, a common way of specifying initial values of instance variables is to write a function which takes the initial values as parameters and returns a class. Other languages (such as C++, Eiffel, and Turbo Pascal's object extensions) often require the introduction of a new language mechanism (*e.g.*, constructors or creation routines) to initialize objects. We accomplish this by the combination of "new" and functions returning classes. This is just one of several ways in which having classes as first-class values is very helpful.

The use of *MyType* permits the implicit recursive definition of object types. For instance, the type *Node_class_type* in the Appendix has an instance variable of type *MyType*, while methods *get_next* and *attach_right* respectively return a value of type *MyType* and take a parameter of type *MyType*. When a message is sent to an object, all occurrences of *MyType* in the method body and its type are replaced by the type of the object. Thus if $o$ is an object of type *Node_type*, the type of *o.get_next* is *func():Node_type*, while the type of *o.attach_right* is *proc(Node_type)*. Because the instance variable *next* in *Node_class_type* has type *MyType*, each object generated from a class with that type also contains a (hidden) instance variable of type *Node_type*.

In the example, *DNode_class*, a class generating doubly-linked nodes, is de-

fined via inheritance from *Node_class*.[5] This is accomplished by adding a new instance variable and methods as well as modifying *attach_right*. All of the instance variables and methods of *Node_class(v)* are inherited by *DNode_class(v)*. Because the types of methods and instance variables are specified using *MyType*, the meaning of the types changes automatically. (This would not happen if the type *Node_type* were written with all occurrences of *MyType* replaced by the name *Node_type*.) Thus, if *node* is of type *Node_type* and *dble_node* is of type *DNode_type*, the value returned from *node.getnext()* will have type *Node_type*, while the value returned from *dble_node.getnext()* will have type *DNode_type*. This makes it possible to polymorphically define homogeneous data structures.

New objects are created by applying *new* to a class. The instance variables of the new object are initialized to the values specified in the class. Classes may be returned from functions in PolyTOIL. *Node_class* is a function which takes a value of type *Num_type* and returns a class, while *OrdList_class* takes a type which matches *Node_class* and returns a class for a linked list composed of that kind of node. *OrdList_class* can be applied to either *Node_type* or *DNode_type* to create a linked list since *DNode_type* <# *Node_type*. In the first case we end up with a singly-linked list, while in the second case we have a doubly-linked list. This is a very powerful and flexible mechanism for creating new objects in a parametric way.

It is worth noting here that one cannot write the program in this way in either C++ or Object Pascal, because the types of the methods and instance variables of *DNode_class* are all different from the corresponding types in the superclass, *Node_class*, because of the use of *MyType*.

We note here that while *DNode_type* <# *Node_type*, it is *not* a subtype. Suppose we write the following procedure:

```
break_it = procedure(sn1, sn2: Node_type)
              begin
                 sn1.attach_right(sn2)
              end;
```

Because Eiffel assumes subclasses generate subtypes, it would allow the call of `break_it(dnode,lnode)` for *dnode* of type *DNode_type* and *lnode* of type *Node_type*. If allowed, this call would result in a run-time error when *lnode* is sent the *set_prev* message from within *attach_right*. PolyTOIL would not allow one to deduce that *DNode_type* <: *Node_type*, and hence would not allow the use of actual parameter *dnode* for *sn1*.

As in the definition of *OrdList_class*, it is often more important to know what messages can be safely sent to an object than to know whether or not it is a subtype of some other type. Thus we choose to support a form of bounded polymorphism where the bound is expressed in terms of matching.

A further example of the usefulness of matching is obtained by noting that the sample program can be further parameterized to make it more flexible. Define

[5] Technically, we define a function which returns a subclass of the class returned by evaluating *Node_class(v)*.

```
Comparable = ObjectType {ge:func(MyType):bool;
                         eq:func(MyType):bool}
```

Note that *Num_type* <# *Comparable*, though again it will *not* be a subtype. Now we can parameterize each of *Node_type, Node_class_type, DNode_type, DNode_class_type, OrdList_type*, and *OrdList_class_type* to take a parameter $T$ <# *Comparable* within *TFunc* headers. All occurrences of *Num_type* within the bodies of the associated methods will now be changed to $T$. The result is a collection of classes which can be used to generate either singly or doubly-linked lists with elements of any type supporting appropriate *ge* and *eq* methods.

# 3 A Formal Definition of PolyTOIL Syntax

In this section we present the formal definitions of types and terms, and provide type-checking rules for PolyTOIL. The language is presented here with an abstract syntax which differs in inessential ways from that used in the earlier examples.

**Definition 1.** Let $\mathcal{V}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and $\mathcal{C}$ be a collection of type constants which includes at least the type constants *Bool, Num, COMMAND, UNIT*, $\bot$, and $\top$. The simple pre-type expressions of PolyTOIL with respect to $\mathcal{V}$. $\mathcal{L}$, and $\mathcal{C}$ are given by the following context-free grammar. We assume $t \in \mathcal{V}$, $c \in \mathcal{C}$, and $m_i \in \mathcal{L}$ in the following.

$$\tau ::= t \mid c \mid \mathit{ref}\ \tau \mid \mathit{func}(\tau_1){:}\,\tau_2 \mid \mathit{func}(\mathit{atype}\ t){:}\,\tau \mid \mathit{func}(t <\!\!\#\ \tau_1){:}\,\tau_2 \mid \{m_1{:}\,\tau_1, \ldots, m_n{:}\,\tau_n\} \mid \mathit{ClassType}(\tau_1, \tau_2) \mid \mathit{VisObjType}(\tau_1, \tau_2) \mid \mathit{ObjectType}\ \tau$$

The *kinds* of the language are collections of constructors. The simplest kinds are the collection of all types and collections of types which match an object type. The higher-order kinds are collections of functions whose domains are restricted to be one of the simple kinds.

$$K ::= \mathit{TYPE} \mid \mathit{TYPE}^{\#\,\tau} \mid \mathit{TYPE} \Rightarrow K \mid \mathit{TYPE}^{\#\,\tau} \Rightarrow K$$

The pre-constructors represent elements of the various kinds. They are types, higher-order functions with types as parameters, or applications of these functions to types.

$$\kappa ::= \tau \mid \mathit{TFunc}[T{:}\,\mathit{atype}]\kappa \mid \mathit{TFunc}[T <\!\!\#\ \tau]\kappa \mid F[\tau]$$

The type constant *COMMAND* is the type of an imperative command expression. *UNIT* is a type whose only value is written () and is used when typing parameterless functions. The type $\bot$ is a subtype of all object types. It contains the element *nil* which is used in our sample program. The type $\top$ is a supertype of all object types, and (in our current implementation) contains built-in *clone*

and *deep_clone* methods which are (implicitly) inherited by all object types. Reference types are the types of variables. The three function types represent the types of normal functions, polymorphic functions, and bounded polymorphic functions.

*VisObjType*$(\sigma, \tau)$ represents the type of the internal view of objects generated from a class with type *ClassType*$(\sigma, \tau)$. The external view represented by the type *ObjectType* $\tau$ does not expose the instance variables. Both $\sigma$ and $\tau$ in these types must be record types.

Because programs in the implemented language are not allowed to use *ref*, record, or *VisObjType* types or the type constants *COMMAND*, *UNIT*, $\top$, or $\bot$, these types do not complicate the language seen by programmers. However, they are useful in writing the type-checking rules for the language.

The kinds give us a way to classify higher-order functions from types to types. For simplicity, we restrict the domains of these functions to either be *TYPE* or $TYPE^{\not\#\tau}$ for some type $\tau$. *TYPE* represents the collection of all types, while $TYPE^{\not\#\tau}$ represents the collection of all (object) types which match $\tau$.

The constructors include all types, functions from types (or bounded collections of types) to constructors, and the application of constructors to types. We omit the axioms and rules for determining which are the legal type and constructor expressions in this conference paper.

The axioms and rules for $<:$ , $\not\#$, and type-checking are given with respect to a set, $C$, of simple type constraints, which provide information about type variables. These definitions are mutually recursive.

**Definition 2.** Relations of the form $t$: *TYPE*, $t \leq \tau$, and $t \not\# \tau$, where $t$ is a type variable and $\tau$ is a type expression, are said to be *simple type constraints*. A *type constraint system* is defined as follows:

1. The empty set, $\epsilon$, is a type constraint system.
2. If $C$ is a type constraint system and $t$ is a type variable which does not appear in $C$, then $C \cup \{t: TYPE\}$ is a type constraint system.
3. If $C$ is a type constraint system such that $C \vdash \tau$: *TYPE*, and $t$ is a type variables which does not appear in $C$ or $\tau$, then $C \cup \{t \leq \tau\}$ is a type constraint system.
4. If $C$ is a type constraint system such that $C \vdash \tau \not\# \top$, and $t$ is a type variable which does not appear in $C$ or $\tau$, then $C \cup \{t \not\# \tau\}$ is a type constraint system.

The restriction that $C \vdash \tau \not\# \top$ in the last clause ensures that $\tau$ will be an object type.

The axioms and rules for $\leq$ and $\not\#$ are similar to those for TOOPLE and can be found in Appendix B. Neither class nor reference types have non-trivial subtypes. The subtyping rule for object types is adapted from the rule for determining subtypes of recursive types in [AC93]. It is more difficult to satisfy than the corresponding matching rule for objects. In particular, if an object type has a method with parameter of type *MyType*, then it cannot have any non-trivial

subtypes. On the other hand, a type formed by adding new methods will always "match" the original. Note that only object types are related by "matching."

Applications of *TFunc* terms in type expressions are evaluated before type checking by explicitly substituting actual parameters for formal parameters in the bodies. We do not include subtyping rules for higher-order constructors (*e.g.*, $C \vdash F \leq G$ iff $C \cup \{t\colon TYPE\} \vdash F(t) \leq G(t)$). These would not be difficult to include, but are omitted from the language for simplicity. Because we evaluate *TFunc* applications before type checking, they are also less important than they might otherwise be.

In the examples presented earlier, we used a concrete syntax which is understood by our interpreter. Here we use a de-sugared abstract syntax which is closer to the way expressions are held internally. In particular, the key words *var* and *method* separating instance variable and method definitions are dropped in favor of a pair of record expressions, and *val* must be applied to a variable to obtain its value. Procedures are modeled by functions which return the unique value, *command*, of type *COMMAND*. All functions and procedures are written in Curried form.

We distinguish between instance variable access and message sending in the abstract syntax by writing $o \Leftarrow m$ for message sending and $o.v$ for accessing instance variables. In the concrete syntax, an unaccompanied message name, $m$, or instance variable, $v$, denoted sending a message to *self* or extracting the instance variable of *self*. However, in the abstract syntax we require that these be written explicitly as $self \Leftarrow m$ or $self.v$.

In this conference paper we will omit the discussion of declarations in PolyTOIL. See the full paper [BSvG94] for details. The syntax of blocks, expressions and commands of the language are given as follows:

**Definition 3.** The set of pre-terms of PolyTOIL over a set C of constants is given by the following context-free grammar:

$$
\begin{aligned}
&\mathit{Block} ::= \textit{begin } S \textit{ return } E \textit{ end}\\
&E ::= x \mid c \mid \textit{val } E \mid \textit{function}(v\colon\sigma)\colon\tau\ \mathit{Block} \mid \textit{function}(t\colon \mathit{atype})\colon\tau\ \mathit{Block} \mid\\
&\qquad \textit{function}(t <\!\!\# \sigma)\colon\tau\ \mathit{Block} \mid E(E') \mid E[\tau] \mid \{m_1 = E_1, \ldots, m_n = E_n\} \mid\\
&\qquad E.m_i \mid \textit{class}(E_1, E_2) \mid \textit{new } E \mid E \Leftarrow m_i \mid E.v_i \mid\\
&\qquad \textit{class inherit } E \textit{ modifying } x_1, \ldots, x_n\\
&\qquad\qquad (\{v_{i_1} = E_1, \ldots, v_{i_k} = E_k\}, \{m_{j_1} = E'_1, \ldots, m_{j_l} = E'_l\})\\
&S ::= x := E \mid \textit{if } E \textit{ then } E_1 \textit{ else } E_2 \mid \textit{while } E \textit{ do } S \mid S; S'
\end{aligned}
$$

The intended meaning of these terms should be clear from our previous discussion. The type-checking axioms and rules for PolyTOIL are given in terms of a type constraint system, $C$, as defined earlier, and a *variable type assignment*, $E$, which assigns types to free variables.

**Definition 4.** A variable type assignment, $E$, is a finite set of associations between variables and type expressions of the form $x\colon\tau$, where each $x$ is unique in $E$. If the relation $x\colon\tau \in E$, then we write $E(x) = \tau$.

The collection of terms of PolyTOIL with respect to $C$, $E$ is the set of pre-terms which can be assigned types with respect to the type-assignment axioms and rules. The most interesting of these are included in Appendix D.

The type-assignment rules for the commands and non-object-oriented expressions should be relatively clear, while those for the object-oriented expressions may require some extra explanation.

The type-assignment rule, *Class*, involves several interesting features. As suggested earlier, the reserved word *self* may be used inside the methods of a class to refer to the object which is executing the method (though *self* may not be used in the initial values of instance variables). Because instance variables are not visible outside of an object's methods, it is convenient to assign *self* two types, an internal type, *SelfType*, in which the instance variables are visible, and an external type, *MyType*, which hides the instance variables. The reserved word *self* represents an internal view of the object, and has type *SelfType*. The reserved word *close*: *SelfType* $\rightarrow$ *MyType* is used to convert a term (usually *self*) from the internal type to the external type of the receiver. (Again, this coercion is inserted automatically in the interpreter for the language, so a programmer need not be aware of it.) Since *SelfType* is the internal type of the object, it cannot appear in the type of any instance variable or method of the object.

Recall that the class contains initial values for the instance variables, while the objects contain locations at which those initial values are stored. The function $RecToMem(\{v_1{:}\sigma_1; \ldots; v_n{:}\sigma_n\}) = \{v_1{:}\, ref\ \sigma_1; \ldots; v_n{:}\, ref\ \sigma_n\}$, is used to convert a record type into one in which the type of each field is a reference to the type in the original.

Finally, recall that values of instance variables and method bodies should be type checked so that they are guaranteed to be type correct when inherited in a subclass. Thus we must type check these terms only under relatively weak assumptions on the possible values of *SelfType* and *MyType*.

With this background we can now understand the type-assignment rule, *Class*, as indicating that to type check a class term one must ensure that the records of initial values of instance variables and of methods have the desired types. This type checking is done assuming that *SelfType* $<:$ $VisObjType(\sigma, \tau)$ and *MyType* $<\#$ *ObjectType* $\tau$, since any subclass of the class being type checked must satisfy these constraints. We also assume that *self* and *close* have the appropriate types in type checking methods. By the subtyping rule for *VisObjType*, which converts types of instance variables using *RecToMem*, the subtype may not change the type of any instance variable, but may only add new ones.

The type-assignment rules for subclasses (classes with *inherit* clauses) are similar to *Class*, but require type checking the new or modified components. It is not necessary to type check inherited methods or instance variables, since they were already type checked under assumptions which are still valid in the subclass. (In fact, even stronger assumptions hold in the subclass.) Unlike C++ or Object Pascal, one may replace a method in a class by one in the subclass whose type is a subtype of the original.

The type-checking rule for message sending is interesting in that it uses the matching relation. A simpler version of the rule given in the appendix would be:

$$Msg' \quad \frac{C, E \vdash o\colon \mathit{ObjectType}\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}}{C, E \vdash o \Leftarrow m_i\colon \tau_i[\mathit{ObjectType}\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}/\mathit{MyType}]}$$

Unfortunately, this simple rule is not sufficient to handle the case in which a message is sent to *self*. That case would be written as $\mathit{close}(\mathit{self}) \Leftarrow m$, where $\mathit{close}(\mathit{self})$ has type $\mathit{MyType} <\!\!\# \mathit{ObjectType}\ \tau$. Since we cannot write the type of $\mathit{close}(\mathit{self})$ in the form $\mathit{ObjectType}\ \tau'$, the simple rule, (*Msg'*), is not applicable, and we must use the more complex rule given in the appendix.

We can get by with a simpler rule for instance variables. Since we assume $\mathit{SelfType} <: \mathit{VisObjType}(\sigma, \tau)$, we can deduce $\mathit{self}\colon \mathit{VisObjType}(\sigma, \tau)$ by subsumption. We do need to remember, however, that the instance variable actually has type $\mathit{ref}\ \sigma_i$, since it is a variable.

## 4 An Operational Semantics for PolyTOIL

In this section we sketch a natural (operational) semantics for PolyTOIL. This semantics is similar to that given in [BCK94] for TOOPLE. We also state a subject-reduction theorem and show how it can be used to prove that our type-assignment rules are safe.

The natural semantics for PolyTOIL provides a description of the reduction rules for a simple interpreter for the language. An environment keeps track of the current values of identifiers, while the store keeps track of what values are stored in currently active locations in memory. The semantic rules reduce a term, $M$, with associated environment, $\rho$, and current store, $s$, to a pair consisting of an irreducible value, $V$, and an updated store, $s'$. We write this as $(M, \rho, s) \downarrow (V, s')$.

There are a number of semantic decisions involving binding time that must be made carefully in order to provide a useful language. For instance, in order for *new* to create new objects each time it is invoked, it is important that locations for instance variables are not allocated when their initial values are declared in a class. Instead, they can only be allocated when a new object is created.

Suppose the initial values of instance variables or method bodies depend on global variables. When should they be evaluated? In TOOPLE, we waited until new objects were created, but, because it was a functional language, it made no real difference. In PolyTOIL, we employ eager evaluation in computing the initial values of instance variables specified in classes, while method bodies (since they are protected by function or procedure abstraction) do not evaluate variables until they are actually invoked by a message.

Another complication is that methods may refer to *self* and *super*. Yet the values of these reserved words are not known at the time a method body is provided in a class definition. Thus the instantiation of the corresponding values must be postponed until the new object is created.

The *irreducible values* in PolyTOIL are elements of base types, locations, functions (represented as *closures*, which are pairs of terms and environments),

and records, classes, and object values with irreducible components. To assist in proving the subject-reduction theorem we presume that each location in the store is associated with a fixed type. We write $Loc_\tau$ for the collection of locations holding values of type $\tau$, and interpret the type *ref* $\tau$ as $Loc_\tau$.[6] *ObjectType* $\tau$ will be interpreted as $\bigcup_\sigma Loc_{VisObjType(\sigma,\tau)} \cup \{nil\}$, where $\sigma$ ranges over record types. (We ignore error terms resulting from sending messages to *nil* in this conference paper.) If we think of terms of the form $obj(a, e)$ as representing objects whose instance variables are visible, objects with hidden instance variables are interpreted as references to those with visible instance variables. (See the full paper for more information on "obj" terms and closures.)

The store is infinitely extensible. The function *GetNewLoc* takes a store, $s$, a type, $\tau$, and a value $V$ of type $\tau$, and returns an unused location $l \in Loc_\tau$, along with a new store, $s'$. The new store is the same as $s$, except that the location $l$ holds the value $V$. The notation $s[l \mapsto V]$ is used to denote the updated store.

Because PolyTOIL supports the application of polymorphic functions to type expressions, and because some other constructs (such as records and regular function definitions) include type expressions as part of the terms, types will necessarily appear in our reduction rules. However, we also include some other type information in the rules in order to make it simpler to prove soundness via the subject-reduction theorem. Aside from indicating the amount of memory necessary to be allocated for a new variable, this extra typing information is inessential to the evaluation of terms.

Because the language is polymorphic, the types of terms can involve type variables. The environment keeps track of the values of these type variables. Because the result of evaluating a term is a pair of an irreducible value and a state, and because our terms involve type information, we will substitute in the values of type variables from the environment as part of the computation. We use the notation $\tau_\rho$ to stand for the result of replacing free variables in a type expression, $\tau$, by the values assigned to them by the environment, $\rho$.

As stated earlier, the natural semantics rules (a selection of which can be found in Appendix E), reduce a term with associated environment and store to a pair consisting of an irreducible value and an updated store. An environment, $\rho$, is a (finite) mapping of names to irreducible values, while a store, $s$, is a (finite) mapping of locations to irreducible values. A state is consistent iff for each type $\tau$ and location $l \in Loc_\tau$, $s(l)$ has type $\tau$. Because we wish to maintain a consistent state, allocation of new memory via *GetNewLoc* (*e.g.*, in expressions of the form *new* $c$) requires the type of memory being allocated and an initial value of that type.

Function expressions reduce to closures. Regular function applications proceed as expected by first reducing the function to a closure and the argument to a value. The body of the function is then reduced using the closure's environment, which is updated to interpret the formal parameter as the actual parameter's value.

---

[6] It would be possible instead to have a single collection of locations and simply keep a "store environment" which keeps track of the intended types of allocated locations, but the approach used here seemed a bit simpler.

Polymorphic functions are applied to type expressions. While there are no reduction rules for type expressions, we must handle properly any type variables contained in the type parameter. As a result, we replace the type variables in the parameter by the values assigned by the current environment. We then update the closure's environment with this modified type expression before evaluating the body of the function.

Class expressions are reduced by evaluating the initialization code for instance variables and method expressions in an environment which does not include interpretations of *self*, *close*, *SelfType*, or *MyType*.

The most complex rule is for *new c*. In *new c*, the expression $c$ is first reduced to an irreducible value of the form $\mathit{class}(\mathit{IV}, \mathit{Methods})$. New locations are allocated for each of the instance variables of the object and the initial values of the instance variables are stored in those locations. Then a new location is allocated to hold the object. The value returned is the new location, into which is stored a term of the form $\mathit{obj}(\mathit{newIV}, \mathit{newMethods})$. $\mathit{NewIV}$ is defined to be the record of locations of the new instance variables, $\mathit{newMethods}$ is obtained from $\mathit{Methods}$ by replacing all free occurrences of *self* by $\mathit{obj}(\mathit{newIV}, \mathit{Methods})$, *close* by the appropriate specialized $\mathit{close}_{(\sigma,\tau)}$, and the types *SelfType* and *MyType* by the types $\mathit{VisObjType}(\sigma, \tau)$ and *ObjectType* $\tau$. Notice that we must be careful to allocate memory corresponding to types in which the free type variables have been replaced by the values assigned by the environment. The rule for $\mathit{close}_{(\sigma,\tau)}$ $o$ is not shown, but is interpreted similarly.

Since *new c* and $\mathit{close}_{(\sigma,\tau)}$ $o$ are the only ways of constructing terms of type *ObjectType* $\tau$, no irreducible values of these types will have free occurrences of *self* or *close*. As a result, the natural semantics of sending a message to an object is trivial. Just look up the value stored in the location corresponding to the object and extract the appropriate method value.

Extracting the value of instance variables from (visible) objects, like sending messages, simply involves extracting the location from the record of instance variables.

Rather than attempting to write one all-encompassing rule for *inherit* terms, we include only one representative case. The semantics of inheritance is the obvious one. The new or modified values are added to or replace the corresponding values in the original class. *Super*, when applied to *self*, returns the record of methods of the superclass.

With some additional definitions and technical results about closures and substitutions, it is possible to state and prove a subject-reduction theorem which ties together the type-checking rules with the natural semantics. The subject-reduction theorem states that if we evaluate a term in a "consistent" environment and state, we obtain a pair consisting of an irreducible term and a consistent state, such that the irreducible term has the same type as the original term. (A careful statement of the theorem can be found in the full paper.) The proof proceeds by induction on the length of a computation. The general approach is similar to that taken in [BCD+93], but with extra complications due to the fact

that class definitions do not contain explicit names for *self* and *MyType*. (This explicit annotation of classes was omitted in PolyTOIL because of the rarity of occurrences of nested classes.) The greatest new difficulties to overcome in this imperative language include handling of environments (*e.g.*, in closures) and ensuring that the state is always typed correctly.

By adding semantic rules which generate a value *stuck* when none of the original rules apply, a simple proof by induction using the subject-reduction theorem shows that no computation gets "stuck." That is, all computations either result in an irreducible value of the appropriate type or loop forever. (We note that sending a message to the *nil* object generates an explicit error value rather than generating "stuck." One must clearly be very careful in inserting "error" results versus "stuck". See the full paper for details.) This theorem ensures the type safety of the language.

## 5 Comparison with Other Work

PolyTOIL is a very expressive, yet type-safe, statically-typed object-oriented programming language. Its type system is more powerful and flexible than other statically-typed languages like C++ and Object Pascal, while avoiding the type-checking problems of Eiffel.

An important difference between PolyTOIL and these other typed object-oriented languages is that the subtype and subclass hierarchies are no longer identified. If they are identified then one is either left with an unsafe language (either by design, or with conventions that require many type casts – essentially by-passing the type system), a language with limited expressibility, or a language which requires extra run-time or link-time checks in order to preserve type safety.

In PolyTOIL the types of instance variables and methods may be given in terms of *MyType*. Thus when a subclass is defined, the types of the instance variables and methods automatically change to reflect that of the new object type.

The combination of the use of *MyType* and bounded polymorphism using matching provides much of the flexibility found in languages using the (unsafe) "covariant" rule for changing types of parameters in subclasses. A "covariant" type system, such as that found in Eiffel, would consider *DNode_type* to be a subtype of *Node_type* in the example in Appendix A. This leads to type errors unless a link-time global analysis of a program is added to identify those places where type-related errors could arise. PolyTOIL does not need this kind of global analysis in order to ensure type safety.

We see PolyTOIL as providing a sound semantic basis for a new generation of object-oriented programming languages which offer both increased expressibility and a safe type system. An example of this impact is the language Strongtalk [BG93], which has adopted essentially the typing rules for TOIL (along with a few extensions) in order to type check a subset of Smalltalk.

The language Theta [DGLM94] was developed independently of this work, but shares many of the features of PolyTOIL. In particular it supports a mech-

anism for constraining polymorphism which is equivalent to our use of bounded matching. The authors also argue that this mechanism is more useful than bounds on type parameters based on subtyping. Unfortunately, Theta does not appear to include a *MyType* construct, and it appears that inherited methods must be type checked again in the context of the subclass. The paper claims that Theta is type safe but provides no supporting evidence. We expect that the results of this paper can be used to add flexibility to their type system, while providing a proof of its type safety.

The Emerald language [BH91] seems to have been the first to explicitly identify the relation of matching as distinct from subtyping. While Emerald does not contain classes, it supports the use of bounded matching to restrict polymorphic functions.

The theoretical work most similar to that described here is [ESTZ94]. That paper presents an analysis of a statically-typed object-oriented language, LOOP, which is similar to PolyTOIL, although it lacks support for polymorphic functions. Results include proofs of type safety as well as the decidability of type checking.

There are a few important differences between PolyTOIL and LOOP. LOOP allows the use of *self* in the initial values of instance variables (though at the cost of a substantially more complex semantics for object creation), and supports multiple inheritance. Its subtyping rules for object types differ substantially from those for PolyTOIL. It includes folding and unfolding rules for object types (similar to those typically used with recursive types), which allow one to replace *MyType* by the type it represents. On the other hand, LOOP does not include our rule for subtyping object types. As a result, object types which involve *MyType* generally do not have subtypes.

LOOP does not explicitly identify the concept of "matching", though it appears that the concept is implicitly supported in the type-checking rules for methods. The typing rules for subclasses indicate that inherited methods should be type checked in the context of the subclass definition, but this does not seem to be necessary. It would be interesting to design a language which combined the strengths of LOOP and PolyTOIL.

The proof of type safety for LOOP differs from that sketched here in that LOOP is first given a somewhat complicated translation into an imperative language, SOOP, which supports F-bounded polymorphism, but has no object-oriented features. The proof of type-soundness for SOOP can then be lifted to LOOP. The operational semantics of LOOP is given via this translation into SOOP, while we provide a direct natural semantics for PolyTOIL.

Other researchers have performed interesting investigations of imperative object-oriented languages by looking at translations into simpler calculi. Pierce, [Pie93], has extended his encoding of object-oriented languages in higher-order bounded lambda calculi [PT94] to imperative languages. The semantics is somewhat simpler than the semantics of PolyTOIL, requiring one fewer fixed-point operator in creating types of objects. However, there is a corresponding loss of expressibility in that methods like the `attach_right` method of `node_class_type`

in Appendix A, which have parameters of type *MyType*, may not be written as methods of the class.

Abadi and Cardelli have written a series of papers [AC94c, AC94b, AC94a] investigating a series of low-level object calculi designed to serve as a foundation for higher-level object-oriented programming languages. Interesting new unpublished results by the same authors show that one of these calculi can be used to model TOOPLE and PolyTOIL. Such a foundational object calculus should make it easier to explore new language features and prove type safety.

# 6 Further Results and Extensions to PolyTOIL

A proof of the decidability of a type-checking algorithm for TOIL appears in [vG93]. It is very similar to the one described in [BCD+93] for TOOPLE. It is easy to extend this to PolyTOIL because of the simplicity of our subtyping rule for polymorphic types. (It omits the problematic rule for bounded polymorphic types which leads to undecidability.) We have built a prototype interpreter for PolyTOIL which is based on the natural semantics given here.

Though not included here, there is also a denotational semantics for TOIL (which should be easily extensible to PolyTOIL), along with a proof of the soundness of type-checking with respect to the denotational semantics (similar to that in [Bru93] and [Bru94]). See [vG93] for details.

Recent work with Leaf Petersen and Jasper Rosenberg at Williams College involved adding new features to the language, improving the interpreter, and improving the readability of the concrete syntax. New features added include new control constructs, recursive types, arrays and other base types, a shorthand type-inclusion notation (similar to that of Rapide [KLM94]), programmer control over visibility of methods, We also added a new subtyping rule which allows us to deduce more subtype relations. The new rule states that if $\sigma <\# \tau$ and all occurrences of *MyType* in $\tau$ are positive then $\sigma$ is a subtype of $\tau$. We have also written and run a large number of PolyTOIL programs which have led us to these changes and have provided us with greater confidence in the strength and flexibility of the language. A reference and users' manual for PolyTOIL are currently being written.

The type system for PolyTOIL allows the programmer greater flexibility than most statically type languages, while providing assurance that the static type checking rules guarantee type safety. We are somewhat concerned, however, at the added complexity for the programmer in needing to keep track of two related, yet different, orderings on types: subtyping and matching. In writing a number of PolyTOIL programs we were somewhat surprised to find that we relied on matching quite heavily, but rarely used subtyping. Current research is involved in exploring the possibility of designing an object-oriented language which supports matching, but not subtyping, while still providing sufficient expressibility for programmers.

# References


[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC94a] Martin Abadi and Luca Cardelli. An imperative object calculus. Manuscript, 1994.

[AC94b] Martin Abadi and Luca Cardelli. A theory of primitive objects: second-order systems. In *Proc. ESOP '94*. Springer-Verlag, 1994. to appear.

[AC94c] Martin Abadi and Luca Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag, 1994. to appear.

[BCD+93] K. Bruce, J. Crabtree, A. Dimock, R. Muller, T. Murtagh, and R. van Gent. Safe and decidable type checking in an object-oriented language. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 29–46, 1993.

[BCK94] K. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 603–626. LNCS 802, Springer-Verlag, 1994.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 215–230, 1993.

[BH91] A. Black and N. Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1 (Revised), DEC Cambridge Research Lab, 1991.

[Bru93] K. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 285–298, 1993.

[Bru94] K. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[BSvG94] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. Technical report, Williams College, 1994.

[BvG93] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. Technical report, Williams College, 1993.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[Coo89] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[DGLM94] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Abstraction mechanisms in Theta. Technical report, MIT Laboratory for Computer Science, 1994.

[ES90] Margaret A. Ellis and Bjarne Stroustrop. *The annotated C++ reference manual.* Addison-Wesley, 1990.

[ESTZ94] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *Proceedings of OOPSLA '94*, pages 16–30, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *21st ACM Symp. Principles of Programming Languages*, pages 138–150, 1994.

[Mey92] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[MMMP90] O. Madsen, B. Magnusson, and B. Moller-Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA-ECOOP '90 Proceedings*, pages 140–150. ACM SIGPLAN Notices,25(10), October 1990.

[Pie93] Benjamin C. Pierce. Mutable objects. Technical report, University of Edinburgh, 1993.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of functional programming*, 4:207–247, 1994. An earlier version appeared in Proc. of POPL '93, pp. 299-312.

[Rey80] J.C. Reynolds. Using category theory to design implicit conversions and generic operators. In N.D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–2580. Springer-Verlag Lecture Notes in Computer Science, Vol. 94, 1980.

[Tes85] L. Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.

[vG93] Robert van Gent. *TOIL: An imperative type-safe object-oriented language.* Williams College Senior Honors Thesis, 1993.

# Appendices

## A Sample PolyTOIL Program

```
program linkedlist;
type
   Num_type = ObjectType {ge: func(MyType):bool;
                          eq: func(MyType):bool;
                          get_val: func():num;
                          set_val: proc(num)};
   Num_class_type = ClassType ({val: num},
                               {ge:...; set_val: proc(num)})

   Node_type = ObjectType {get_next: func():MyType;
                           get_val: func():Num_type;
                           set_next: proc(MyType);
                           set_val: proc(Num_type);
                           attach_right: proc(MyType)};
```

```
   Node_class_type = ClassType ({val: Num_type ; next: MyType},
                                {get_next: ...});

   DNode_type = ObjectType {get_next: func():MyType;
                            set_next: proc(MyType);
                            get_val: func():Num_type;
                            set_val: proc(Num_type);
                            get_prev: func():MyType;
                            set_prev: proc(MyType);
                            attach_right: proc(MyType)};
   DNode_class_type = ClassType ({val: Num_type ; next: MyType},
                                 {...});

   OrdList_type = TFunc[U <# Node_type]
                          ObjectType {find: func(Num_type):bool;
                                      add: proc(U)};
   OrdList_class_type = TFunc[U <# Node_type]
                          ClassType ({head: U}, {find:...});

const
   Num_class = class
               var val = 0: num;
               methods
                  ge = function(other: MyType): bool
                     begin return val >= other.get_val(); end;
                  ... : Num_class_type;
   Node_class = function(v: Num_type): Node_class_type
      begin
         return class
            var val = v: Num_type;
                next = nil: MyType;
            methods
               get_next = function(): MyType
                  begin   return next;  end;
               set_next = procedure(nxt: MyType)
                  begin  next := nxt;  end;
               get_val = ...; set_val = ...;
               attach_right = procedure(nxt: MyType)
                  begin  set_next(nxt);  end;  ...
      end;
    DNode_class = function(v: Num_type): DNode_class_type
      begin
         return inherit Node_class(v) modifying attach_right
            var  prev = nil: MyType;
            methods
```

```
            get_prev = ...;  set_prev = ...;
            attach_right = procedure(nxt: MyType)
               begin set_next(nxt); nxt.set_prev(self) end;
         end;
   OrdList_class=function(U <# Node_type): OrdList_class_type[U]
      begin
         return class
            var  head = nil: U;
            methods
               find = function(match: Num_type): bool ... end;
               add = procedure(new_node:U)
                  var  prev: U;  current: U;
                  begin
                     if head = nil then
                        head := new_node;
                        new_node.set_next(nil);
                     else if head.get_val().ge(new_node.get_val())
                        then
                           new_node.attach_right(head);
                           head := new_node;
                     else ...
      end;  -- function
var num_obj: Num_type;
    lnode: Node_type;                  -- singly linked node
    dnode: DNode_type;                 -- doubly linked node
    slist: OrdList_type[Node_type];    -- singly linked list
    dlist: OrdList_type[DNode_type];   -- doubly linked list
begin
    num_obj := new (Num_class);
    num_obj.set_val(1);
         -- create singly linked list
    slist := new (OrdList_class(Node_type));
         -- create doubly linked list
    dlist := new (OrdList_class(DNode_type));
    dnode := new (DNode_class(num_obj));
    dlist.add(dnode);
    printBool (dlist.find (num_obj));
end.
```

## B Selected Subtyping Rules

We include only the rules for selected type constructors.

$$Bound\text{–}Bound(<:) \qquad \frac{C \cup \{u <\# \gamma\} \vdash \tau[u/t] <: \tau'[u/t']}{C \vdash (Func(t <\# \gamma):\ \tau) <:\ (Func(t' <\# \gamma):\ \tau')},$$

where u does not occur in $C$, $\tau$ or $\tau'$.

$$Func(<:) \qquad \frac{C \vdash \sigma' <: \sigma,\ C \vdash \tau <: \tau'}{C \vdash Func(\sigma):\ \tau\ <:\ Func(\sigma'):\ \tau'},$$

$$Rec(<:) \qquad \frac{C \vdash \sigma_j <: \tau_j,\ \text{for } 1 \le j \le k,\ \ C \vdash \sigma_j : TYPE,\ \ \text{for } k \le j \le n}{C \vdash \{m_1 : \sigma_1; \ldots; m_k : \sigma_k; \ldots; m_n : \sigma_n\} <: \{m_1 : \tau_1; \ldots; m_k : \tau_k\}},$$

$$VisObj(<:) \qquad \frac{C \vdash RecToMem(\sigma) <: RecToMem(\sigma'),\ C \vdash \tau <: \tau'}{C \vdash VisObjType(\sigma, \tau) <: VisObjType(\sigma', \tau')},$$

$$Obj(<:), \qquad \frac{C \cup \{t: TYPE, s <: t\} \vdash \tau[s/Mytype] <: \tau'[t/Mytype]}{C \vdash ObjectType\ \tau <: ObjectType\ \tau'}.$$

where neither $s$ nor $t$ may occur free in $C$, $\tau$, or $\tau'$.

## C Selected Matching Rules

Matching is a preorder on object types with $\top$ as maximum element. The most important structural rule is:

$$ObjectType(<\#) \ \frac{C \cup \{Mytype <\# ObjectType\ \sigma\} \vdash \sigma <: \tau}{C \vdash ObjectType\ \sigma <\# ObjectType\ \tau},$$

## D Selected Type-Checking Rules

**Type Assignment Rules (Commands):**

$$Assn \qquad \frac{C, E \vdash X : ref\ \tau,\ \ C, E \vdash M : \tau}{C, E \vdash X := M : \text{COMMAND}}$$

$$Cond \ \frac{C, E \vdash B : Bool,\ C, E \vdash S : \text{COMMAND},\ \ C, E \vdash T : \text{COMMAND}}{C, E \vdash if\ B\ then\ S\ else\ T\ end : \text{COMMAND}}$$

Similar rules exist for *while* loops and sequencing.

**Type Assignment Rules (Expressions):**

$$Function \qquad \frac{C, E \cup \{v : \sigma\} \vdash Block : \tau}{C, E \vdash function\ (v : \sigma)\ Block:\ (func\ (\sigma) : \tau)}$$

$$BdPolyFunc \qquad \frac{C \cup \{t <\# \gamma\}, E\ \vdash Block : \tau}{C, E \vdash function\ (t <\# \gamma)\ Block:\ (func\ (t <\# \gamma) : \tau)}$$

*FuncAppl*
$$\frac{C, E \vdash f: func(\sigma): \tau, \quad C, E \vdash M: \sigma}{C, E \vdash f(M): \tau}$$

*BoundFuncAppl*
$$\frac{C, E \vdash f: func(t <\# \gamma): \tau, \quad C \vdash \sigma <\# \gamma}{C, E \vdash f(\sigma): \tau[\sigma/t]}$$

*Class*
$$\frac{C^{IV}, E \vdash e_1: \sigma, \quad C^{METH}, E^{METH} \vdash e_2: \tau}{C, E \vdash class(e_1, e_2): ClassType(\sigma, \tau)}$$

where $C^{IV} = C \cup \{MyType <\# ObjectType\ \tau\}$,
$C^{METH} = C^{IV} \cup \{SelfType <: VisObjType(\sigma, \tau)\}$,
$E^{METH} = E \cup \{self: SelfType, close: func(SelfType): MyType\}$
Neither *MyType* nor *SelfType* may occur free in $C$ or $E$.
$\tau$ must be the type of a record of functions.

*Object*
$$\frac{C, E \vdash c: ClassType(\sigma, \tau)}{C, E \vdash new\ c: ObjectType\ \tau}$$

*Msg*
$$\frac{C \vdash \gamma <\# ObjectType\{m: \tau\}, C, E \vdash o: \gamma}{C, E \vdash o \Leftarrow m: \tau[\gamma/MyType]}$$

*InstVar*
$$\frac{C, E \vdash o: VisObjType(\{v_1: \sigma_1; \ldots; v_n: \sigma_n\}, \tau)}{C, E \vdash o.v_i: ref\ \sigma_i}$$

*Inherits*
$$\frac{\begin{array}{c} C, E \vdash c: ClassType(\{v_1: \sigma_1; \ldots; v_m: \sigma_m\}, \{m_1: \tau_1; \ldots; m_n: \tau_n\}), \\ C^{IV} \vdash \tau_1' <: \tau_1, \quad C^{IV}, E \vdash a_{m+1}: \sigma_{m+1}, \quad C^{IV}, E \vdash a_1': \sigma_1, \\ C^{METH}, E^{METH} \vdash e_{n+1}: \tau_{n+1}, \quad C^{METH}, E^{METH} \vdash e_1': \tau_1' \end{array}}{\begin{array}{l} C, E \vdash inherit\ c\ modifying\ v_1, m_1 \\ \qquad (\{v_1 = a_1': \sigma_1, v_{m+1} = a_{m+1}: \sigma_{m+1}\}, \\ \qquad\ \{m_1 = e_1': \tau_1', m_{n+1} = e_{n+1}: \tau_{n+1}\}): \\ \quad ClassType(\{v_1: \sigma_1; \ldots; v_{m+1}: \sigma_{m+1}\}, \\ \qquad\ \{m_1: \tau_1'; m_2: \tau_2; \ldots; m_{n+1}: \tau_{n+1}\}) \end{array}}$$

where $C^{IV} = C \cup \{MyType <\# ObjectType\ \{m_1: \tau_1'; \ldots; m_{n+1}: \tau_{n+1}\}\}$,
$C^{METH} = C^{IV} \cup \{SelfType <: VisObjType(\{v_1: \sigma_1; \ldots; v_{m+1}: \sigma_{m+1}\}, \{m_1: \tau_1'; \ldots; m_{n+1}: \tau_{n+1}\})\}$
$E^{METH} = E \cup \{self: SelfType, close: func(SelfType): MyType, super: SelfType \rightarrow \{m_1: \tau_1; \ldots; m_n: \tau_n\}\}$
Neither *MyType* nor *SelfType* may occur free in $C$ or $E$.

*Subsump*
$$\frac{C \vdash \sigma <: \tau, \quad C, E \vdash e: \sigma}{C, E \vdash e: \tau}$$

**Type Assignment Rules (Commands):**

$$Assn \qquad \frac{C, E \vdash x\text{: } ref\ \tau,\ \ C, E \vdash M\text{: } \tau}{C, E \vdash x := M\text{: COMMAND}}$$

$$Cond \quad \frac{C, E \vdash B\text{: } Bool,\ C, E \vdash S\text{: COMMAND},\ \ C, E \vdash T\text{: COMMAND}}{C, E \vdash \text{if } B \text{ then } S \text{ else } T \text{ end: COMMAND}}$$

$$While \qquad \frac{C, E \vdash B\text{: } Bool,\ C, E \vdash S\text{: COMMAND}}{C, E \vdash \text{while } B \text{ do } S \text{ end: COMMAND}}$$

$$StmtList \qquad \frac{C, E \vdash S\text{: COMMAND},\ C, E \vdash T\text{: COMMAND}}{C, E \vdash S;\ T\text{: COMMAND}}$$

# E The Natural Semantics of PolyTOIL

**Expressions:**

$$FuncAppl \qquad \frac{(M, \rho, s^0) \downarrow (\langle \text{function}(v\text{: } \sigma)B, \rho_f \rangle, s^1),\ (N, \rho, s^1) \downarrow (V_1, s^2),\ \ (B, \rho_f[v \mapsto V_1], s^2) \downarrow (V, s^3)}{(M(N), \rho, s^0) \downarrow (V, s^3)},$$

$$BdPolyFuncAppl \quad \frac{(M, \rho, s^0) \downarrow (\langle \text{function}(t <\!\!\# \gamma)B, \rho_f \rangle, s^1),\ (B, \rho_f[t \mapsto \tau_\rho], s^1) \downarrow (V, s^2)}{(M[\tau], \rho, s^0) \downarrow (V, s^2)},$$

$$Class \qquad \frac{(a, \rho', s) \downarrow (V_a, s'),\ \ (e, \rho', s') \downarrow (V_e, s'')}{(\text{class}(a, e), \rho, s) \downarrow (\text{class}(V_a, V_e), s'')},$$

where $\rho' = \rho \backslash \{self, close, SelfType, MyType\}$

$$New \quad \frac{\begin{array}{c}(c, \rho, s) \downarrow (\text{class}(\{iv_1 = V_1\text{: } \sigma'_1, \ldots, iv_n = V_n\text{: } \sigma'_n\}, Methods), s^0),\\ (newLoc_i, s^i) = GetNewLoc\ s^{i-1}\ \sigma''_i\ V'_i,\ \text{for } 1 \le i \le n,\\ (newLoc_{n+1}, s^{n+1}) = GetNewLoc\ s^n\ VisObjType(\sigma', \tau')\ o'\end{array}}{(\text{new } c, \rho, s) \downarrow (newLoc_{n+1}, s^{n+1})},$$

where $\sigma' = \sigma_{\rho'} = \{iv_1\text{: } \sigma'_1; \ldots; iv_n\text{: } \sigma'_n\},\ \tau' = \tau_{\rho'}$
for $\rho' = \rho \backslash \{self, close, SelfType, MyType\}$,
$\sigma''_i = \sigma'_i[MyType \mapsto ObjectType\ \tau']$
$V'_i = V_i[MyType \mapsto ObjectType\ \tau'])$,
$newIV = \{iv_1 = newLoc_1\text{: } ref\ \sigma''_1, \ldots, iv_n = newLoc_n\text{: } ref\ \sigma''_n\}$,
$o' = \text{obj}(newIV, Methods)$,
if $Methods = \{\langle f_1, \rho_1 \rangle, \ldots, \langle f_k, \rho_k \rangle\}$

$$\text{then } newMethods = \{\langle f_1, \rho_1'\rangle, \ldots, \langle f_k, \rho_k'\rangle\}$$
$$\text{where for } 1 \le i \le k,$$
$$\rho_i' = \rho_i[self \mapsto o', close \mapsto close_{(\sigma',\tau')}, SelfType \mapsto VisObjType(\sigma', \tau'), MyType \mapsto ObjectType\ \tau']$$

$$Message \quad \frac{(o, \rho, s) \downarrow (Loc, s'),\ s'(Loc) = \text{obj}(V_a, \{m_1 = V_1 : \tau_1', \ldots, m_n = V_n : \tau_n'\})}{(o \Leftarrow m_i, \rho, s) \downarrow (V_i, s')},$$

$$InstVble \quad \frac{(o, \rho, s) \downarrow (\text{obj}(\{iv_1 = V_1 : \sigma_1', \ldots, iv_n = V_n : \sigma_n'\}, V_e), s')}{(o.iv_i, \rho, s) \downarrow (V_i, s')},$$

$$Inherit \quad \frac{\begin{array}{c}(c, \rho, s) \downarrow (\text{class}(V_a, \{m_1 = V_1 : \tau_1', \ldots, m_{upd} = V_{old} : \tau_{old}', \ldots, m_n = V_n : \tau_n'\}), s'),\\ (u, \rho', s') \downarrow (V_{new}, s'')\end{array}}{\begin{array}{l}(\text{class inherit } c \text{ modifying } m_{upd};\ (\{\}, \{m_{upd} = u : \tau_{new}'\}), \rho, s) \downarrow\\ \quad (\text{class}(V_a, \{m_1 = V_1 : \tau_1', \ldots, m_{upd} = V_{new} : \tau_{new}', \ldots, m_n = V_n : \tau_n'\}), s'')\end{array}},$$

where $sup = \langle \text{function}(self : SelfType).\ \{m_1 = V_1 : \tau_1', \ldots, m_{upd} = V_{old} : \tau_{old}', \ldots, m_n = V_n : \tau_n'\}, \rho\rangle$
$\rho' = \rho[super \mapsto sup] \backslash \{self, close, SelfType, MyType\}$
$\tau_i' = (\tau_i)_{\rho'}$.

**Commands:**

$$Assign \quad \frac{(X, \rho, s) \downarrow (Loc, s'),\ (M, \rho, s') \downarrow (V, s'')}{(X := M, \rho, s) \downarrow (\text{command}, s''[Loc \mapsto V])},$$

$$Conditional_{true} \quad \frac{(B, \rho, s) \downarrow (\text{true}, s'),\ (M, \rho, s') \downarrow (\text{command}, s'')}{(\text{if } B \text{ then } M \text{ else } N \text{ end}, \rho, s) \downarrow (\text{command}, s'')},$$

$$Conditional_{false} \quad \frac{(B, \rho, s) \downarrow (\text{false}, s'),\ (N, \rho, s') \downarrow (\text{command}, s'')}{(\text{if } B \text{ then } M \text{ else } N \text{ end}, \rho, s) \downarrow (\text{command}, s'')},$$

$$while_{true} \quad \frac{(B, \rho, s) \downarrow (\text{true}, s'),\ (S, \rho, s') \downarrow (\text{command}, s''),\ (\text{while } B \text{ do } S \text{ end}, \rho, s'') \downarrow (\text{command}, s''')}{(\text{while } B \text{ do } S \text{ end}, \rho, s) \downarrow (\text{command}, s''')},$$

$$while_{false} \quad \frac{(B, \rho, s) \downarrow (\text{false}, s')}{(\text{while } B \text{ do } S \text{ end}, \rho, s) \downarrow (\text{command}, s')},$$

$$Sequence \quad \frac{(C_1, \rho, s) \downarrow (\text{command}, s'),\ (C_2, \rho, s') \downarrow (\text{command}, s'')}{(C_1;\ C_2, \rho, s) \downarrow (\text{command}, s'')}$$