PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language

KIM B. BRUCE, ANGELA SCHUETT and ROBERT VAN GENT
Williams College
and
ADRIAN FIECH
Memorial University of Newfoundland

PolyTOIL is a new statically typed polymorphic object-oriented programming language that is provably typesafe. 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 that 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 and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, constraints, inheritance, polymorphism*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*objecct-oriented languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*object-oriented constructs*

General Terms: Design, Language, Theory

Additional Key Words and Phrases: Matching, hash type

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 that are also very expressive. At one extreme we have statically typed languages like

The research of K. B. Bruce, A. Schuett, and R. van Gent was partially supported by NSF grants CCR-9121778, CCR-9424123, and CCR-9870253. A Fiech's research was partially supported by NSERC grant OGP0170497. The results in this article are based in part on the Williams College senior honors theses of van Gent and Schuett. An extended abstract of this article appeared in the proceedings of ECOOP '95.
Author's addresses: K. B. Bruce, Department of Computer Science, Williams College, Williamstown, MA 01267; email: kim@cs.williams.edu, A. Schuett, Department of Electrical Engineering and Computer Science, University of California at Berkeley; R. van Gent, Extempo Systems, Inc.; A. Fiech, Memorial University of Newfoundland.

C++ [Ellis and Stroustrop 1990], Java [Arnold and Gosling 1996], and Object Pascal [Tesler 1985], 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 [Goldberg and Robson 1983], which are prone to runtime errors of the form "message not understood." Somewhere between these extremes are languages like Beta [Madsen et al. 1990] and Eiffel [Meyer 1992], which require either runtime checks or more elaborate linktime checks to pick up errors that cannot be detected reliably by a static type-checker.

The language PolyTOIL is the result of a series of design efforts to build a type-safe language based on progress in the theoretical understanding of object-oriented languages (see Cardelli [1988], Cardelli and Wegner [1985], Cook et al. [1990], Canning et al. [1989], Pierce and Turner [1993], and Abadi and Cardelli [1994 a, b, 1995], for example, for work on modeling object-oriented features). In Bruce [1993], we introduced a statically typed, functional, object-oriented language, TOOPLE. This and subsequent papers [Bruce 1994; Bruce et al. 1994, 1993] 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 transitional language TOIL [Bruce and van Gent 1993; van Gent 1993] 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 that provides a very flexible yet safe language for object-oriented programming. In this article 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, represented by instance variables, and operations, represented by methods;
—**Classes**, to generate objects (although delegation would be a reasonable alternative);
—**Message sending**, as a way of specifying computation;
—**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 that expects one of a different but related type; and
—**Keywords**, `self` and `super`, representing the receiver of a message and the set of methods from 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 classes and methods**, allowing a programmer to abstract over a type in order to define a related family of types or operations; and
—**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. We list below several other important features of PolyTOIL.

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. As we show below, 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 overridden methods to be subtypes of the corresponding types in the superclass.

PolyTOIL supports a new keyword, `MyType`, which represents the type of `self`. The use of this keyword allows more accurate typing of methods that have parameters or return values whose type is desired to be the same as `self`. An interesting feature of the use of `MyType` is that subclasses of classes whose method types involve `MyType` need not always give rise to subtypes. Although we could, like Trellis/Owl [Schaffert et al. 1986], restrict legal subclasses to those that generate subtypes, we find it more useful to separate the subclass and subtype hierarchies in PolyTOIL.

Separating these hierarchies in PolyTOIL allows greater expressiveness while providing type safety. We introduce a new relation on types called *matching* that corresponds more closely than subtyping to the subclass hierarchy in the presence of `MyType`. This relation, which is more general 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 language, 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 show in the next section, constraining types using matching is much more useful than using the subtype relation.

We have a number of results on the type system and semantics of PolyTOIL, however, in this article 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 that 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 is interesting

in that it applies to an environment-based natural semantics for a polymorphic language.

In Section 2 of this article we discuss the design of PolyTOIL. We include an extended example that illustrates the flexibility of the language. In Section 3 we provide the formal syntax and type-checking rules for PolyTOIL. In Sections 4 and 5 we provide the natural semantics and present the subject-reduction and type-safety theorems. In Section 6 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 on PolyTOIL and its successors.

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. A simple example of a PolyTOIL program is given in Figure 1.[1] In the following subsections we describe the more innovative features of PolyTOIL by referencing this and other examples.

2.1 Classes Are Not Types

Types in programming languages provide interfaces that 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.

Java [Arnold and Gosling 1996] allows programmers to define interfaces as well as classes, and several classes may implement the same interface. Both classes and interfaces may be used as types in Java. We go further in PolyTOIL by completely separating classes from object types. Classes may not be used as types. Instead classes are "first class" values in PolyTOIL. They may be used as parameters in functions and may be returned from functions.

The program in Figure 1 defines a class `HelloClass` with instance variable `happy` of type `boolean`, and methods `setMood` and `printMood`. All instance variables are accessible only within the class and its subclasses. This is similar to C++ and Java's `protected` feature, with the exception that a method can access only variables of the receiver. (In Java a method can access the private variables of other accessible objects of the same class.) Methods are always accessible outside of the class (like Java's *public* methods).[2] As usual the class includes the method bodies and initial values for instance variables.

The type `HelloType` is the type of *objects* generated from `HelloClass`. The types of objects begin with the keyword `ObjectType` and include only method

[1]We have changed some features of the syntax of the language in inessential ways to obtain a more compact representation of code for the purposes of this article. The type `A → B` represents a function type with domain `A` and range `B`. If the range type is `void` then it is the type of a procedure.
[2]The implemented PolyTOIL language includes facilities to restrict the visibility of methods. We have omitted these here to reduce the complexity of our semantics.

```
program easy;

type

   HelloClassType = ClassType (
        { happy: bool },          -- types of instance variables
        { setMood: bool -> void;  -- types of methods
          printMood: void -> void });

   HelloType = ObjectType       -- method types only
        { setMood: bool -> void;
          printMood: void -> void };

const                           -- classes are considered constants

   helloClass = class
      var
         happy = true: bool;
      methods
         setMood = procedure(theMood: bool)
            begin happy := theMood; end
         printMood = procedure()
            begin
               if (happy) then
                  printString("Have a wonderful day!");
                  newline(1);
               else
                  printString("Go away!");
               end;
            end
   end: HelloClassType;

var myMood: HelloType;

begin
   myMood := new(HelloClass);
   myMood.printMood();
   myMood.setMood(false);
end
```

Fig. 1. A simple class and types in PolyTOIL.

signatures. Thus `HelloType` includes the names and signatures of both `setMood` and `printMood`.

In contrast, the type `HelloClassType` is the type of the *class* `HelloClass`. The types of classes begin with the keyword `ClassType`, and include the signatures of all methods and instance variables in the class, but no initial values or method bodies. This more complete information is necessary because we might write a function that takes a class parameter and creates a subclass. Because instance variables and all methods from the superclass are available in the subclass, it is important that the type of the class include all of that information (although note that it is *not* important that the bodies of the methods be known at compile time—only their signatures).

Thus classes in PolyTOIL are typically associated with two types: a `ClassType` that describes the class itself, and an `ObjectType` that describes the objects generated by the class. Because these types are only interfaces, distinct classes may have the same associated object and class types (the latter, however, would require the classes to be remarkably similar).

Just as we distinguish classes from types, we can also distinguish subclasses from subtypes. A type σ is a *subtype* of type τ, written $\sigma <: \tau$, if a value of type σ can be used in any context expecting a value of type τ. In most popular object-oriented languages (e.g., C++, Java, Object Pascal, and Eiffel), subtypes may only arise from the creation of subclasses, but there is no need for this restriction. As described in Cook et al. [1990], it is possible to have subclasses that do not generate subtypes, and subtypes that 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 that generate subtypes (as was done in Trellis/Owl [Schaffert et al. 1986]) or separating the subclass and subtype hierarchies. In the interest of maximizing expressiveness we have chosen to separate these hierarchies.

In Java, classes `C` and `D` may implement interfaces (object types) `IC` and `ID` where `IC` extends `ID`, but `C` does not extend `D`. In our terminology, object type `IC` could be a subtype of `ID`, but class `C` is not a subclass of `D`. However, it is not possible in Java to have a class `C` extend class `D` without the first being able to be used as a subtype of the second. (If `D` implements interface `ID`, and `C` extends `D`, then `C` automatically also implements interface `ID`.)

In the next subsection we see how the introduction of a new type expression standing for the type of `self` can result in subclasses that do not give rise to subtypes.

2.2 `Self` and `MyType`

Virtually all object-oriented languages include a construct `self` (sometimes named `this` or `Current`) that can be 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 `COType` 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 `COType`. However, the method `m` might also be inherited in a subclass `D` of `C`. In this new context, `self` represents an object generated from `D`, 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 assumptions on the type of `self` that will also hold in every subclass.

To do this, we must first know what changes are allowed in the types of methods in subclasses. As explained in Bruce [1994], to preserve type safety the body of a method `m` with type τ_m in a class `C` may only be overridden in a subclass `D` with a new method as long as the type τ'_m of the replacement method is a subtype of τ_m.

This motivates the definition of the *matching* relation <# between object types. ObjectType τ' <# ObjectType τ if and only if for every method name m in τ there is a corresponding method name m in τ', and the type of m in τ' is a subtype of its type in τ. To preserve type safety, we restrict the definition of subclass so that when D is a subclass of C, the corresponding object types (i.e., types of objects generated by the two classes) will match. (An example of two types that match, but are not subtypes, is given later in this section. Formal definitions of matching and subtyping are postponed to the next section.)

In order to accomodate the possible changes to the type of self in subclasses, and to increase the expressiveness of the language, 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 τ. Thus any object of type MyType will be guaranteed to have all of the methods occurring in ObjectType τ, and the types of the methods will be guaranteed to be subtypes of those given in τ. By type-checking under this assumption, we need not worry about the type safety of the method when it is inherited in subclasses. It is safe because the types of objects generated from subclasses will be guaranteed to match ObjectType τ.

Figure 2 contains a simple example of the use of MyType in the definition of a class that generates objects that can be used as the nodes of linked lists. NodeClass is defined as a function that takes a parameter of type NumType, returning a class that uses the parameter as the initial value of the val field (the next field is initially nil, a keyword with semantics similar to Java and C++'s null). It is convenient to think of NodeClass simply as a parameterized class, even though technically it is a function that returns a class.

In the definition of NodeClass, the type of instance variable next is given as MyType. Because getNext and setNext return the value of next and update the value of next, their return type and parameter type, respectively, have type MyType.

As mentioned earlier, the type of a class begins with the keyword ClassType. NodeClassType, the return type of NodeClass, contains the types for all of the instance variables and methods of the class. Types of objects begin with the keyword ObjectType. NodeClass will generate objects of type NodeType, as the methods and their types in NodeClassType and NodeType are the same. In particular, the method types of NodeType also involve MyType.

Although the formal type-checking rules are given in the next section, we provide some intuition here. Let nd be an object of type NodeType. Because MyType stands for the type of self (the object executing the method), nd.getNext() returns an object of type NodeType, the type of the receiver. Similarly, nd.setNext(otherNd) is well typed only if otherNd has type NodeType. Intuitively, if you send a message to an object with static type T, then the type of the message send is obtained from the type of the method by replacing all occurrences of MyType by T.

So far the typing works exactly as if all occurrences of MyType in NodeClass and NodeType were replaced by NodeType. The difference arises in the definition of subclasses.

```
-- type of object holding an integer value
NumType = ObjectType {ge: MyType -> bool;
                      eq: MyType -> bool;
                      getVal: void -> integer;
                      setVal: integer -> void };

-- type of class generating objects of type NumType
NumClassType = ClassType ({ val: integer },
                          { ge: MyType -> bool;
                            eq: MyType -> bool;
                            getVal: void -> integer;
                            setVal: integer -> void });

--Type of singly linked node objects with values of type NumType
NodeType = ObjectType { getNext: void -> MyType;
                        setNext: MyType -> void;
                        getVal: void -> NumType;
                        setVal: NumType -> void };

-- Type of class generating singly linked node objects
NodeClassType = ClassType ({ val: NumType;
                             next: MyType },
                           { getNext: void -> MyType;
                             setNext: MyType -> void;
                             getVal: void -> NumType;
                             setVal: NumType -> void });

--class generating singly linked nodes with values of type NumType
NodeClass = function(v: NumType): NodeClassType
  begin
     return class
         var val = v: NumType;
             next = nil: MyType;
         methods
            getNext = function(): MyType
               begin return next; end
            setNext = procedure(nxt: MyType)
               begin next := nxt; end
            getVal = function(): NumType
               begin return val; end
            setVal = procedure(vl: NumType)
               begin val := vl; end
         end;  -- class
   end;
```

Fig. 2. Node class with `MyType`.

Figure 3 contains the definition of a class that generates doubly linked nodes. It is defined as a subclass of `NodeClass` formed by adding the new instance variable `prev` and methods `getPrev` and `setPrev`. As declared in the header, it also overrides the inherited method `setNext`.

Both the inherited instance variable `next` and the new instance variable `prev` have type `MyType`. Similarly, the inherited methods and new methods have

```
--Type of doubly linked node objects with values of type NumType
DNodeType = ObjectType { getNext: void -> MyType;
                         setNext: MyType -> void;
                         getVal: void -> NumType;
                         setVal: NumType -> void;
                         getPrev: void -> MyType;
                         setPrev: MyType -> void };

-- Type of class generating doubly linked node objects
DNodeClassType = ClassType ({ val: NumType;
                              next: MyType;
                              prev: MyType },
                            { getNext: void -> MyType;
                              setNext: MyType -> void;
                              getVal: void -> NumType;
                              setVal: NumType -> void;
                              getPrev: void -> MyType;
                              setPrev: MyType -> void });

--class generating doubly linked nodes with values of type NumType

DNodeClass = function(v: NumType): DNodeClassType
  begin
     return class
            inherit NodeClass(v) modifying setNext
         var
            prev = nil: MyType;
         methods
            getPrev = function(): MyType
                begin return prev; end
            setPrev = procedure(prv: MyType)
                begin prev := prv; end
            setNext = procedure(nxt: MyType)
                begin super.setNext(nxt); nxt.setPrev(self); end
            end; -- class
   end;
```

Fig. 3. DNode subclass with MyType.

types involving MyType. In the context of class NodeClass, MyType suggests type NodeType; in DNodeClass it suggests type DNodeType. This is very similar to the change of meaning of self depending on context. Because the interpretation of MyType changes automatically from NodeType to DNodeType, the inherited instance variables and methods are consistent with the new ones in DNodeClass.

If one were to attempt defining DNodeClass as a subclass of NodeClass in a language without MyType, severe problems would arise. If all occurrences of MyType in NodeClass were replaced by NodeType, and those in DNodeClass were replaced by DNodeType, then the inherited methods and instance variables would be inconsistent with the new ones. Thus, in DNodeClass, the instance variable next would have type NodeType, but the instance variable prev would have type DNodeType. These and other similar inconsistencies in method signatures would result in problems. Suppose dn were a variable of type DNodeType. If dn were

a doubly linked node in a list, one would expect `dn.getNext().getPrev()` to return `dn`. However, with the class definitions described above without `MyType`, this expression is not even well typed, as the result of `dn.getNext()` has static type `NodeType` rather than `DNodeType`.

Most statically typed object-oriented programming languages (including Java, C++, and Object Pascal) do not include a keyword such as `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. To preserve type safety, the language designer must require the types of subclasses to be subtypes of the type of the superclass. As a result it becomes impossible to properly redefine methods such as `setNext` that take parameters whose type should correspond to that of the receiver (`self`).

Trellis/Owl [Schaffert et al. 1986] was one of the first languages to include a `MyType` construct. Because of concerns with the failure of subtyping, Trellis/Owl required that all subclasses also generate subtypes. This requirement essentially restricted uses of `MyType` to the return types of methods. Occurrences of `MyType` as the type of parameters of methods would not result in subtypes.

Eiffel [Meyer 1992] also includes a construct similar to `MyType` (written `like Current` in Eiffel). Eiffel rejected the restrictions imposed by Trellis/Owl and allowed the use of `like Current` as the type of parameters of methods and as the type of instance variables. Unfortunately, the free use of this construct and the identification of subclasses with subtypes leads to type insecurities [Cook 1989]. We illustrate this problem with an example.

As is clear when we give the formal definition of matching, in our example above, `DNodeType <# NodeType`, but we claim it is *not* a subtype. Suppose we write the following PolyTOIL procedure,

```
breakIt = procedure(sn1, sn2: NodeType)
              { sn1.setNext(sn2) }
```

If we were to adopt the Eiffel rules that assume subclasses generate subtypes, we would treat `DNodeType` as a subtype of `NodeType`. Thus we would allow the call of `breakIt(dnode, lnode)` if `dnode` is an object generated from class `DNodeClass` (and hence of type `DNodeType`) and `lnode` an object generated from class `NodeClass` (and hence of type `NodeType`).

However, the execution of this call would result in the execution of the `setNext` code from `DNodeClass`. As the reader can easily check, the `setNext` code includes sending the `setPrev` message to its parameter, which in this case is of type `NodeType`. Because elements of type `NodeType` need not have a method named `setPrev`, the program will crash. The difficulty is that at runtime the receiver of the message `setNext` is of type `DNodeType`, and hence expects a parameter of the same type. Unfortunately, what it finds is a parameter of a different type, `NodeType`.

PolyTOIL only allows the user to deduce that `DNodeType <# NodeType`, therefore it would not allow the use of actual parameter `dnode` of type `DNodeType` for formal parameter `sn1` with declared type `NodeType`.

Because Eiffel would allow `DNodeType` to be treated as a subtype of `NodeType`, its static type system is not safe. Eiffel designer Bertrand Meyer originally

proposed [Meyer 1992] a linktime "system validity check" for Eiffel to compensate for the failures of the static type system to catch errors. More recently he has proposed a different restriction of the language that would ban "polymorphic CAT-calls" [Meyer 1995]. Neither appears to have been implemented in a publicly available Eiffel compiler, probably because of the loss of expressiveness that would result.

We avoid these problems in PolyTOIL while preserving much of the expressiveness 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 statically typed object-oriented languages such as C++, Java, and Object Pascal.

2.3 PolyTOIL Is Imperative

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`, $\mathtt{e}_T$ is an expression of type `T`, and $\mathtt{x}_U$ is a variable of type `U`, one would like to allow the assignment $\mathtt{x}_U := \mathtt{e}_T$.

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`. As is common in object-oriented languages, sending a message to `nil` will result in a runtime error, which is *not* considered to be a type error. That is, this error is treated in the same way as dividing by 0, an error that the programmer is required to check for and handle, and that cannot be detected by the type system.[3]

If `x` is a variable holding values of type τ, the type-checker treats `x` as having type `ref` τ. Because these reference types are both receivers and producers of values, they have no nontrivial subtypes (see Reynolds [1980]). Similar problems obstruct the use of subtyping with call-by-reference parameters. For instance, a procedure might take a formal parameter of type `U` and assign to it a new object of that type. If an actual parameter `x` of type `T` $<:$ `U` is passed in to the procedure as a call-by-reference parameter, it will be assigned a value of type `U`. This would create a hole in the type system that would show up if a message `m` belonging to `T` but not `U` were sent to `x` after the procedure returns.

For example, let

```
breakIt2 = procedure(a: ref U, b: U)
                    { a := b }
```

where the notation `a : ref U` indicates that `a` is called by reference. Let `T` be a subtype of `U`, `x` be a variable of type `T`, and `exp` be an expression of type `U`. Then `breakIt2(x,exp)` would result in the type error of assigning a value of type `U` to a variable of type `T`, a subtype.

[3]Of course is is possible to define type systems that rule out both of these errors, but they tend to be very awkward in practice. We chose to include `nil` as a keyword with this behavior because we wish to explain languages with features and programming styles similar to those in current use.

To avoid adding special rules to restrict subtyping in such cases, PolyTOIL only supports "call-by-sharing" parameters. This parameter-passing technique is similar to "call-by-constant-value" in that 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 and Java, is more flexible than it might first appear.

2.4 Parameterized Types and Match-Bounded Polymorphism

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

We illustrate the expressiveness of PolyTOIL by extending our earlier example of singly and doubly linked nodes with a class that can construct either singly or doubly linked ordered lists from these nodes. The PolyTOIL type and class definitions can be found in Figure 4. Code using these definitions is given in Figure 5.

`OrdListType` and `OrdListClassType` in Figure 4 are functions that take a type `U` and return an object and class type, respectively. `OrdListClass` is a function that takes a type `U` matching `NodeType`, and returns a class with type `OrdListClassType[U]`. Thus `OrdListType` and `OrdListClassType` are parameterized types, and `OrdListClass` is a parameterized class.

In the case of `OrdListClass`, the type parameter is match-bounded by `NodeType`. Thus it can be instantiated by `NodeType` or by `DNodeType`, since both match `NodeType`. As illustrated in Figure 5, `OrdListType[NodeType]` is the type of ordered singly linked lists, whereas `OrdListType[DNodeType]` is the type of ordered doubly linked lists. Also, evaluation of `new OrdListClass(NodeType)` results in the creation of a new singly linked list, whereas evaluation of `new OrdListClass(DNodeType)` results in the creation of a new doubly linked list. The type-checking rules will make it impossible to add a doubly linked node to a singly linked list or vice versa.

The match constraint on type variable `U` guarantees that values of type `U` (e.g., the local variable `current` in method `find`) can be sent messages `getNext`, `setNext`, `getVal`, and `setVal`. Thus if `current` has type `U`, then `current.getNext()` will return a value of type `U`. Similary, if `newNode` has type `U` then `newNode.setNext(head)` is well typed only if `head` also has type `U`. As this example shows, it is often more important to know what messages can be safely sent to an object than to know whether 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, which provides exactly this information.

Match-bounded polymorphism is similar to F-bounded polymorphism [Canning et al. 1989] in expressiveness, but is somewhat more expressive and meshes more smoothly with subclasses (see Abadi and Cardelli [1996]).

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.

```
-- Type of objects representing ordered linked lists of NumType
OrdListType = TFunc[U] ObjectType { find: NumType -> bool;
                                    add: U -> void };

-- Type of class generating ordered linked lists
OrdListClassType = TFunc[U] ClassType ({ head: U },
                                       { find: NumType -> bool;
                                         add: U -> void });

-- Parameterized class generating ordered linked lists
-- Can be instantiated to make either singly or doubly linked lists
OrdListClass = function(U <# NodeType): OrdListClassType[U]
   begin
      return class
         var
            head = nil: U;
         methods
            find = function(match: NumType): bool
                var
                   done: bool;
                   current: U;
                begin  done := false;
                       current := head;
                       while not(done) & not(current = nil) do
                          done := current.getVal().eq(match);
                          if not(done) then current := current.getNext(); end;
                       end;
                       return (done);
                end
            add = procedure(newNode: U)
               var
                  prev, current: U;
               begin  if head = nil then
                         head := newNode;
                         newNode.setNext(nil);
                      else if head.getVal().ge(newNode.getVal()) then
                              newNode.setNext(head);
                              head := newNode;
                           else ... ; end ; end
               end
         end;  -- class
   end;  -- function
```

Fig. 4. Example of match-bounded polymorphism in PolyTOIL. Part i.

Define

```
Comparable = ObjectType {ge:MyType -> bool;
                         eq:MyType -> bool}
```

Note that `NumType <# Comparable`, although again it will *not* be a subtype. Now we can modify each of `NodeClass`, `DNodeClass`, and `OrdListClass` to take a second type parameter `T <# Comparable`. All occurrences of `NumType` within the bodies of the associated methods can now be changed to `T`. The result is a collection of classes that can be used to generate either singly or doubly linked lists with elements of any type supporting appropriate `ge` and `eq` methods.

Having provided this overview of PolyTOIL, we proceed in the next section to give a formal definition of PolyTOIL and its type-checking rules. In Section 4

```
var
   numObj: NumType;
   lnode: NodeType;
   dnode: DNodeType;
   slist: OrdListType[NodeType];    -- singly linked list
   dlist: OrdListType[DNodeType];   -- doubly linked list

begin
    numObj := new (NumClass);
    numObj.setVal(1);
    slist := new (OrdListClass(NodeType));  -- create singly linked list
    dlist := new (OrdListClass(DNodeType)); -- create doubly linked list
    dnode := new (DNodeClass(numObj));
    dlist.add(dnode);
    printBool (dlist.find (numObj));
end
```

Fig. 5. Example of the use of match-bounded polymorphism in PolyTOIL. *Part* ii.

we specify its semantics in preparation for showing that the language is type safe.

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 that differs in inessential ways from that used in the earlier examples. In particular, we replace $\texttt{ClassType}(\tau_1, \tau_2)$ by $\texttt{ClassType}(\texttt{mt}, \tau_1, \tau_2)$, and $\texttt{ObjectType}\ \tau$ by $\texttt{ObjectType}(\texttt{mt}, \tau)$. The bound type variable `mt` in these two type expressions stands for the `MyType` of the class or object type. Explicit provision of this type name makes it somewhat simpler to handle nested types. Since nesting occurs only infrequently in practice, the concrete syntax of the language always assumes that the type variable is `MyType`. For simplicity in the exposition below, we usually use `MyType` for this bound type variable.

3.1 Kinds and Types in PolyTOIL

Types include the object and class types discussed earlier, but also include other types that are useful in expressing type-checking rules and semantics of the language. Kinds are used to classify types and higher-order functions from types to types.

The *kinds* of the language describe collections of type constructors (types and functions that may result in types). The constant `TYPE` denotes the collection of all types of PolyTOIL. The constant `RECTYPE` denotes the collection of record types (which are not types of the concrete syntax). The higher-order kinds denote collections of functions from types to type constructors.

$$K ::= \quad \texttt{TYPE} \mid \texttt{RECTYPE} \mid \texttt{TYPE} \Rightarrow K.$$

The preconstructors represent elements of the various kinds. They are types, record types, or higher-order functions with types as parameters, or applications

of these functions to types.

$$\kappa ::= \tau \mid \gamma \mid \texttt{TFunc}[t]\kappa \mid \kappa[\tau],$$

where τ is a pretype expression, t is a type variable, and γ is a record pretype expression, defined below.

Once we define pretype and record pretype expressions, we provide kinding rules for determining which of the preconstructors can be assigned a kind. The constructors are those preconstructors that can be assigned a kind.

We next define the pretypes of PolyTOIL. The types of the language are those pretypes that can be assigned a kind by the kinding rules.

Definition 3.1. 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 that includes at least the type constants Bool, Num, `PROGRAM`, `COMMAND`, Void, Null, and Object. The *simple pretype expressions, PreType*, and *record pretype expressions, PreRecType*, 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.

$$\begin{aligned}
\tau \in \mathit{PreType} \quad &::= t \mid c \mid \texttt{ref}\ \tau \mid \tau_1 \to \tau_2 \mid \forall t.\tau \mid \forall t <\# \tau_1.\tau_2 \mid \forall t <: \tau_1.\tau_2 \mid \\
&\qquad \texttt{ClassType}(t, \gamma_1, \gamma_2) \mid \texttt{ObjectType}(t, \gamma) \\
\gamma \in \mathit{PreRecType} &::= \{m_1 : \tau_1; \ldots; m_n : \tau_n\}.
\end{aligned}$$

In the following text we use the metavariables $\tau, \sigma, \delta, \alpha, \beta, \gamma, \xi$, with and without subscripts to describe type expressions. The set of free variables of a type τ, written $FV(\tau)$, is defined as usual. The type variable t is not free in $\forall t.\tau$, $\forall t <\# \tau_1.\tau_2$, $\forall t <: \tau_1.\tau_2$, $\texttt{ClassType}(t, \gamma_1, \gamma_2)$, or $\texttt{ObjectType}(t, \gamma)$.

We let the type constant `PROGRAM` stand for the type of an entire program, and `COMMAND` stand for the type of an imperative command expression. The type `Void` is used when typing parameterless functions. We also encode procedures from the concrete syntax as functions that return the value `command` of type `COMMAND` (which behaves as a null command). Thus the type of parameterless procedures is given as $\texttt{Void} \to \texttt{COMMAND}$.

The type `Null` is a subtype of all object types. It contains the element *nil* that is used in our sample program. The type `Object` is a supertype of all object types, and (in our current implementation) contains built-in `clone` and `deepClone` methods that are (implicitly) inherited by all object types.

Reference types are the types of variables. That is, if x is a variable holding values of type τ, then x has type $\texttt{ref}\ \tau$. This notation allows us to distinguish between values of type τ and variables that hold values of that type.

As is standard, the type $\tau_1 \to \tau_2$ is the type of functions taking a parameter of type τ_1 and returning a value of type τ_2. The types $\forall\ \texttt{t}.\tau$, $\forall\ \texttt{t} <\# \tau_1.\tau_2$, and $\forall\ \texttt{t} <: \tau_1.\tau_2$ represent unbounded and bounded polymorphic functions (i.e., functions that take types as parameters). The identifier `t` is bound by these type expressions. Although only match-bounded polymorphism is supported by the concrete syntax, we find it convenient to include subtype-bounded polymorphism to express the semantics of partially evaluated programs. As usual we identify polymorphic types that are the same up to renaming of the bound variable.

Records were not part of the concrete syntax discussed in the previous section, but it is convenient to have them in the abstract syntax as a way of building up object and class types. Thus the definition of record pretypes is given in the definition of *PreRecType*. The order of fields in records is irrelevant, so record types that are identical up to the order of fields is identified. We often abbreviate records and their types with notation such as $\{m_i = a_i : \tau_i\}_{i \le n}$ and $\{m_i : \tau_i\}_{i \le n}$.

The external view of an object generated from a class with type `ClassType(MyType, `γ_i`, `γ_m`)` is represented by the type `ObjectType(MyType, `γ_m`)`, which does not expose the instance variables. In this notation, `MyType` is the type variable representing the type of `self`. (That is, in the abstract syntax, `MyType` is not a keyword, but is represented by whatever type variable appears as the first component of each class and object type.)

The definition of `NodeType` from Figure 2 would be written in the abstract syntax as follows.

```
NodeType = ObjectType(MyType,
                      {getNext: Void -> MyType;
                        setNext: MyType -> Command;
                        getVal: Void -> NumType;
                        setVal: NumType -> Command})
```

Because programs in the implemented language may not mention `ref`, `record`, or $\forall t <: \tau_1.\tau_2$ types, or the type constants `PROGRAM`, `COMMAND`, `Object`, or `Null`, these types do not complicate the language seen by programmers. However, they are useful in writing the type-checking rules for the language.

The axioms and rules for determining valid types and constructors are given with respect to a set, C, of simple type constraints, which provide information about free type variables. The definition of type constraints, the rules for determining valid types and constructors, and the matching and subtyping rules are mutually recursive.

Definition 3.2. Relations of the form t`:TYPE`, $t <: \tau$, and $t <\# \tau$, where t is a type variable and τ is a type expression, are said to be *simple type constraints*. A *type constraint system* C is defined as follows.

1. The empty set $\emptyset$ is a type constraint system.
2. If C is a type constraint system and t is a type variable that 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 variable that does not appear in C or τ, then $C \cup \{t <: \tau\}$ is a type constraint system.
4. If C is a type constraint system such that $C \vdash \tau <\#$ `Object`, and t is a type variable that does not appear in C or τ, then $C \cup \{t <\# \tau\}$ is a type constraint system.

The restriction that $C \vdash \tau <\#$ *Object* in the last clause ensures that τ will be an object type.

In Figures 6 and 7 we include axioms and rules for determining which are the legal type and constructor expressions. In the `ObjectType` and `ClassType` rules, the types of all methods must be function or polymorphic function types.

$$C \vdash c\colon \texttt{TYPE},\ \ for\ c \in \mathcal{C}$$

$$C \cup \{t\colon \texttt{TYPE}\} \vdash\ \ t\colon \texttt{TYPE}$$

$$C \cup \{t <\colon \tau\} \vdash\ \ t\colon \texttt{TYPE}$$

$$C \cup \{t <\!\!\# \tau\} \vdash\ \ t\colon \texttt{TYPE}$$

$$\frac{C \vdash \sigma\colon \texttt{TYPE} \qquad C \vdash \tau\colon \texttt{TYPE}}{C \vdash \sigma \to \tau\colon \texttt{TYPE}}$$

$$\frac{C \cup \{t\colon \texttt{TYPE}\} \vdash \tau\colon \texttt{TYPE}}{C \vdash \forall t.\tau\colon \texttt{TYPE}}$$

$$\frac{C \cup \{t <\!\!\# \gamma\} \vdash \tau\colon \texttt{TYPE}}{C \vdash \forall t <\!\!\# \gamma.\tau\colon \texttt{TYPE}}$$

$$\frac{C \cup \{t <\colon \gamma\} \vdash \tau\colon \texttt{TYPE}}{C \vdash \forall t <\colon \gamma.\tau\colon \texttt{TYPE}}$$

$$\frac{C \vdash \tau_i\colon \texttt{TYPE}\ for\ 1 \le n}{C \vdash \{l_i\colon \tau_i\}_{i \le n}\colon \texttt{RECTYPE}}$$

$$\frac{C \vdash \tau\colon \texttt{TYPE} \qquad \tau \notin \{\texttt{COMMAND}, \texttt{PROGRAM}\}}{C \vdash \texttt{ref}\ \tau\colon \texttt{TYPE}}$$

$$\frac{C \cup \{\texttt{MyType}\colon \texttt{TYPE}\} \vdash \gamma\colon \texttt{RECTYPE}}{C \vdash \texttt{ObjectType}(\texttt{MyType}, \gamma)\colon \texttt{TYPE}}$$

$$\frac{C \cup \{\texttt{MyType}\colon \texttt{TYPE}\} \vdash \gamma_i\colon \texttt{RECTYPE} \qquad C \cup \{\texttt{MyType}\colon \texttt{TYPE}\} \vdash \gamma_m\colon \texttt{RECTYPE}}{C \vdash \texttt{ClassType}(\texttt{MyType}, \gamma_i, \gamma_m)\colon \texttt{TYPE}}$$

Fig. 6. Type and constructor rules.

The axioms and rules for $<\colon$, which are similar to those for our earlier language TOOPLE, can be found in Figure 8. Neither class nor reference types have nontrivial subtypes. The subtyping rule for function types is contravariant in the argument type and covariant in the result type [Cardelli 1988]. The subtyping rules for polymorphic types support covariant changes in the return type, but no changes in the bounds of the type parameters. Although the rules could be generalized to allow contravariant changes in the type bounds, the decidability of subtyping would be lost [Pierce 1994]. The subtyping rule for record types allows both depth and width subtyping.

$$\frac{C \cup \{t\colon \mathtt{TYPE}\} \vdash \kappa\colon K}{C \vdash \mathtt{TFunc}[t]\ \kappa\colon \mathtt{TYPE} \Rightarrow K}$$

$$\frac{C \vdash \kappa\colon \mathtt{TYPE} \Rightarrow K, \quad C \vdash \tau\colon \mathtt{TYPE}}{\kappa[\tau]\colon K}$$

Fig. 7. Type and constructor rules (cont.).

$$Refl_{<:} \qquad \frac{C \vdash \tau\colon \mathtt{TYPE}}{C \vdash \tau <: \tau}$$

$$Hyp_{<:} \qquad C \cup \{t <: \tau\} \vdash t <: \tau$$

$$Trans_{<:} \qquad \frac{C \vdash \sigma <: \tau \qquad C \vdash \tau <: \delta}{C \vdash \sigma <: \delta}$$

$$Func_{<:} \qquad \frac{C \vdash \sigma <: \sigma' \qquad C \vdash \tau' <: \tau}{C \vdash \sigma' \to \tau' <: \sigma \to \tau}$$

$$Poly_{<:} \qquad \frac{C \cup \{u\colon \mathtt{TYPE}\} \vdash \tau'[t' \mapsto u] <: \tau[t \mapsto u] \qquad u \notin FV(\tau') \cup FV(\tau)}{C \vdash \forall t'.\tau' <: \forall t.\tau}$$

$$MBdedPoly_{<:} \qquad \frac{C \cup \{u <\!\!\# \gamma\} \vdash \tau'[t' \mapsto u] <: \tau[t \mapsto u] \qquad u \notin FV(\tau') \cup FV(\tau) \cup FV(\gamma)}{C \vdash \forall t' <\!\!\# \gamma.\tau' <: \forall t <\!\!\# \gamma.\tau}$$

$$SBdedPoly_{<:} \qquad \frac{C \cup \{u <: \gamma\} \vdash \tau'[t' \mapsto u] <: \tau[t \mapsto u] \qquad u \notin FV(\tau') \cup FV(\tau) \cup FV(\gamma)}{C \vdash \forall t' <: \gamma.\tau' <: \forall t <: \gamma.\tau}$$

$$Record_{<:} \qquad \frac{C \vdash \tau_i' <: \tau_i \quad for\ 1 \le i \le n, \qquad C \vdash \tau_i'\colon \mathtt{TYPE} \quad for\ n+1 \le i \le n+m}{C \vdash \{l_j\colon \tau_j'\}_{j \le n+m} <: \{l_j\colon \tau_j\}_{j \le n}}$$

$$ObjType_{<:} \qquad \frac{\begin{array}{c} C\ \cup \{t\colon \mathtt{TYPE}, s <: t\} \vdash \tau'[\mathtt{MyType}' \mapsto s] <: \tau[\mathtt{MyType} \mapsto t] \\ for\ t, s \notin FV(\tau') \cup FV(\tau) \end{array}}{C \vdash \mathtt{ObjectType}(\mathtt{MyType}', \tau') <: \mathtt{ObjectType}(\mathtt{MyType}, \tau)}$$

$$Null_{<:} \qquad \frac{C \vdash \mathtt{Null} <\!\!\# \tau}{C \vdash \mathtt{Null} <: \tau}$$

Fig. 8. Subtyping rules.

Because object types can be understood as recursive types in which `MyType` refers to the entire type, the subtyping rule for object types is similar to that for determining subtypes of recursive types given in Amadio and Cardelli [1993]. 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 nontrivial subtypes. This intuitively follows from the fact that the parameter type described by `MyType` would change covariantly in a subtype, and that violates subtyping for function types. This problem was illustrated in the last section by the `breakIt` procedure.

$$\textit{Object-Refl}_{<\#} \qquad C \vdash \texttt{Object} <\# \texttt{Object}$$

$$\textit{Hyp}_{<\#} \qquad C \cup \{t <\# \tau\} \vdash t <\# \tau$$

$$\textit{ObjType-Object}_{<\#} \qquad \frac{C \vdash \texttt{ObjectType}(\texttt{MyType}, \tau) : \texttt{TYPE}}{C \vdash \texttt{ObjectType}(\texttt{MyType}, \tau) <\# \texttt{Object}}$$

$$\textit{Refl}_{<\#} \qquad \frac{C \vdash \tau <\# \texttt{Object}}{C \vdash \tau <\# \tau}$$

$$\textit{Null}_{<\#} \qquad \frac{C \vdash \tau <\# \texttt{Object}}{C \vdash \texttt{Null} <\# \tau}$$

$$\textit{Trans}_{<\#} \qquad \frac{C \vdash \sigma <\# \tau \qquad C \vdash \tau <\# \gamma}{C \vdash \sigma <\# \gamma}$$

$$\textit{ObjType}_{<\#} \qquad \frac{C \cup \{u <\# \texttt{ObjectType}(\texttt{MyType}', \tau')\} \vdash \tau'[\texttt{MyType}' \mapsto u] <: \tau[\texttt{MyType} \mapsto u] \quad \textit{for } u \notin FV(\tau') \cup FV(\tau)}{C \vdash \texttt{ObjectType}(\texttt{MyType}', \tau') <\# \texttt{ObjectType}(\texttt{MyType}, \tau)}$$

Fig. 9. Matching rules.

It is convenient to have the type `Null` as a subtype of every object type. `Null` is a type whose only element is `nil`. The element `nil` represents an uninitialized object. We make `Null` a subtype of all object types in order to have `nil` be an element of every object type. Alternatively we could have altered the type-checking rules given later so that `nil` would be given every object type. See Bruce [2002] for a further discussion of how to handle `nil` in the formal description of object-oriented languages.

Applications of `TFunc` terms in type expressions are evaluated before type-checking by explicitly substituting actual parameters for formal parameters in the bodies of the type functions. The substitution is performed to aid in type-checking. We do not include subtyping rules for higher-order constructors (e.g., $C \vdash F <: G$ if and only if $C \cup \{t: \texttt{TYPE}\} \vdash F(t) <: G(t)$). These rules 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.

It is straightforward to show the following lemma.

LEMMA 3.3. *Let* $C \vdash \sigma <: \tau$. *Then* $C \vdash \sigma : \texttt{TYPE}$ *and* $C \vdash \tau : \texttt{TYPE}$.

PROOF. Simple induction. □

The matching relation is defined in Figure 9. Notice that it is defined only on object types. All but the last matching rule are relatively trivial. The last rule, $\textit{ObjType}_{<\#}$, looks complex, but its import is relatively straightforward. Most of the complications in the hypotheses result from the necessity of using the same type variable to stand for `MyType`. It states that matching types can arise by either "width" or "depth" subtyping of the record of methods. That is, one may either add new types, or replace existing types by subtypes of the originals. Because the `MyTypes` on both sides are replaced by the same variable, they are

treated as being equal for the purposes of determining whether two object types match. Notice that all we are allowed to assume about the MyTypes is that they match the smaller of the two object types in the comparison.

It is again easy to show the following.

LEMMA 3.4. *Let* $C \vdash \sigma <\# \tau$. *Then* $C \vdash \sigma$: TYPE *and* $C \vdash \tau$: TYPE.

PROOF. Simple induction. □

3.2 PolyTOIL Expression Syntax

In the examples presented earlier, we used a concrete syntax that is understood by our interpreter. Here we use a desugared abstract syntax that is closer to the way expressions are held internally. An important advantage of the desugared syntax is that it makes the proof of the type safety of the language easier.

As an example of the changes in the abstract syntax, the key words *var* and *methods* 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. As mentioned earlier, procedures are modeled by functions that return the value command of type COMMAND. All functions and procedures are written in curried form for simplicity in the exposition.

The formal syntax of programs, declarations, blocks, expressions, and commands of the language are given as follows.

Definition 3.5. *The set PreTerm of preterms* of PolyTOIL over a set $\mathcal{B}$ of term constants, a set $\mathcal{L}$ of labels, and a set $\mathcal{X}$ of term identifiers is given by the following context-free grammar (we assume $x \in \mathcal{X}$, $b \in \mathcal{B}$, $l, m \in \mathcal{L}$, *and* $\sigma, \tau \in$ *PreType*).

$$
\begin{array}{lll}
\mathit{Prog} & ::= & \texttt{Program}\ x\texttt{;}\ \mathit{Block}. \\
\mathit{Block} & ::= & \mathit{CDcls}\ \mathit{VDcls}\ \ \texttt{begin}\ S\ \ \texttt{return}\ M\ \ \texttt{end} \\
\mathit{CDcls} & ::= & \texttt{const}\ \mathit{CDclLst} \mid \varepsilon \\
\mathit{CDclLst} & ::= & x = M \mid x = M\texttt{;}\ \mathit{CDclLst} \\
\mathit{VDcls} & ::= & \texttt{var}\ \mathit{VDclLst} \mid \varepsilon \\
\mathit{VDclLst} & ::= & x\texttt{:}\tau \mid x\texttt{:}\tau\texttt{;}\ \mathit{VDclLst} \\
M & ::= & x \mid b \mid \ \texttt{val}\ M \mid \ \texttt{function(}x\texttt{:}\tau\texttt{)}\ \mathit{Block} \mid \ \texttt{function(}t\texttt{:}\ \texttt{TYPE)}\ \mathit{Block} \mid \\
 & & \quad \texttt{function(}t <\# \tau\texttt{)}\ \mathit{Block} \mid \ \texttt{function(}t <\texttt{:}\ \sigma\texttt{)}\ \mathit{Block} \mid M(M') \mid M[\tau] \mid \\
 & & \quad \{l_1 = M_1\texttt{:}\tau_1\texttt{;}\ldots\texttt{;}l_n = M_n\texttt{:}\tau_n\} \mid M.l_i \mid \ \texttt{class(}M_1, M_2\texttt{)} \mid \ \texttt{new}\ M \mid \\
 & & \quad M \Leftarrow m_i \mid \ \texttt{class inherit}\ M\ \ \texttt{modifying}\ l, m\ (\{l_i = M_i\texttt{:}\sigma_i\}_{i\le n}, \\
 & & \quad \{m_j = M_j\texttt{:}\tau_j\}_{j\le m}) \\
S & ::= & x\texttt{:=}\ M \mid \ \texttt{if}\ M\ \ \texttt{then}\ S_1\ \ \texttt{else}\ S_2\ \ \texttt{end} \mid \ \texttt{while}\ M\ \ \texttt{do}\ S\ \ \texttt{end} \mid S; S' \mid \varepsilon
\end{array}
$$

In the above, M stands for an expression,[4] and S stands for a statement or command. *CDclLst* is a sequence of constant definitions, and *VDclLst* is a list of variable declarations.

[4]We use M rather than E to represent expressions because we use E below to represent static type information during type-checking.

The intended meaning of most terms should be clear from our previous discussion. The set $\mathcal{B}$ of term constants must always contain the constants `ok`, `command`, and `nil`. `ok` is of type `PROGRAM`, `command` is of type `COMMAND`, and `nil` is of type `Null`. As with types, the order of fields in records is not significant. In the following text we use $\Im[c] \subseteq \mathcal{B}$ to denote the set of constants with type $c \in \mathcal{C}$.

The most significant difference between the concrete and abstract syntax is the way in which we write instance variables and methods. The main idea is that the keywords `MyType` and `self` are replaced by explicit parameters. We also distinguish between sending methods and accessing instance variables by reserving the parameter corresponding to `self` for sending messages, and introducing a new term identifier `selfinst` and type identifier `InstType`for the record of values of instance variables and its type. It is not really necessary to separate `self` into these two pieces. However, we find it convenient for the purposes of writing the semantics and proving the type safety of the language.

We handle the keywords `self`, `MyType`, and so on, in class definitions by making them parameters to the features in which they can appear. Because instance variables can refer to `MyType`, but not `self`, in the abstract syntax, we require that instance variables be functions with a type parameter `MyType`. Methods are more complex, depending on `MyType` and `self`, as well as the new type and term identifiers `InstType` and `selfinst`, so the concrete syntax of methods includes all four of these as parameters.

What are the constraints on these new parameters? Not surprisingly, we declare `self` to have type `MyType`, and `selfinst` to have type `InstType`. But these typings won't help much if we don't know anything about `MyType` or `InstType`.

Let σ and τ be the types of the instance variables and methods of the class being defined (before adding the extra parameters). Recall that the class contains initial values for the instance variables, whereas the objects contain locations at which those initial values are stored. We use the function *RecToMem* to convert a record type σ into one in which the type of each field is a reference to the type in the original. That is, $\mathit{RecToMem}(\{l_i\!:\sigma_i\}_{i\le n}) = \{l_i\!:\ \texttt{ref}\ \sigma_i\}_{i\le n}$.

Because the instance variables and methods of a class may be inherited in subclasses, it would be a mistake to type-check them under the assumption that `MyType` $=$ `ObjectType(MyType,` τ`)` or that `InstType` $=\mathit{RecToMem}(\sigma)$. Instead we need to ensure that the terms remain type correct in all subclasses.

Both in the expression `MyType` $=$ `ObjectType(MyType,` τ`)` *and in type bounds of the form* `MyType <# ObjectType(MyType,` τ`),` `MyType` *occurs both as a free type variable (the first occurrence in each expression) and as a bound variable. Using the same identifier in both places may seem confusing. However, it allows us to simplify the notation of many of our rules.*

Recall that the types of objects generated by subclasses always match the type of objects generated by the superclass. Thus it will always be safe to assume that `MyType <# ObjectType(MyType,` τ`)`. Similarly, because one can not change the type of existing instance variables in subclasses, but only add new ones, we may assume that `InstType` $<:\mathit{RecToMem}(\sigma)$. (Note that because the types of fields of $\mathit{RecToMem}(\sigma)$ all involve reference types, subtyping rules ensure that `InstType` can only differ from $\mathit{RecToMem}(\sigma)$ by the addition of new fields.)

```
function(v: NumType): NodeClassType
   begin
      return class (
          { val = function(MyType <# NodeType) begin return v end:
                          ∀ MyType <# NodeType. NumType;
            next = function(MyType <# NodeType) begin return nil end:
                          ∀ MyType <# NodeType. MyType; },
          { getNext = function(MyType <# NodeType) begin return
                        function(InstType <: NodeInstType) begin return
                          function(self: MyType) begin return
                            function(selfInst: InstType) begin return
                              function() begin return
                                val selfInst.next; end end end end end:
                      ∀ MyType <# NodeType.∀InstType <: NodeInstType.
                                 Mytype -> InstType -> Void -> MyType
            -- other method definitions.
          })
   end; -- function
```

where `NodeInstType` *abbreviates* `{ val: ref NumType; next: ref MyType }`

Fig. 10. `NodeClass` from Figure 2 in abstract syntax.

These constraints convey useful information as the two assumptions that `self: MyType` and `selfinst: InstType`, when combined with the constraints `MyType <# ObjectType(MyType,` τ`)` and `InstType <:` $RecToMem(\sigma)$, allow us to determine when a message send to `self` or the extraction of an instance variable is guaranteed to be safe.

The only reason that subtype-bounded polymorphism is included in the concrete syntax is to be able to express this constraint on `InstType` for methods. The implemented language's syntax only includes match-bounded polymorphism.

Finally, in the concrete syntax, an unaccompanied message name m, or instance variable v, denoted sending a message to `self` or extracting an instance variable of `self`. However, in the abstract syntax we require that these be written explicitly as `self` $\Leftarrow$ m and `selfinst`.v.

To illustrate these changes in syntax, we translate the definition of `NodeClass` given in Figure 2 to the one in Figure 10, which is written in the abstract syntax. Note that there is no reason to include `MyType`, `self`, or any other keyword in the header of class expressions (the way we did with class and object types) because all occurrences of the keywords have been replaced by parameters in the bodies of instance variables and methods. Clearly no programmer would happily write code this ugly, but it can all be generated automatically from the concrete syntax shown in Figure 2. (See Appendix E for details of the translation.)

The type-checking axioms and rules for PolyTOIL are given in terms of a type constraint system C, as defined earlier, and an *identifier type assignment* E, which assigns types to free identifiers.

Definition 3.6. An *identifier type assignment* E (with respect to C) is a finite set of associations between identifier and type expressions of the form $x:\tau$, where each x is unique in E and $C \vdash \tau$: `TYPE`. If the relation $x:\tau \in E$, then we write $E(x) = \tau$.

The collection *Term* of terms of PolyTOIL with respect to C, E is the set of preterms that can be assigned types with respect to the type-assignment axioms and rules in Appendix A. Note that Appendix A has type rules for an extended language that includes terms corresponding to semantics values that may occur during evaluation of a program. Those terms are introduced in the next section.

The declaration type-assignment rules provided in Appendix A yield expanded type assignments rather than just types. These expanded type assignments are used to type-check the rest of the program. Thus an assertion of the form $C, E \vdash_s Dcl \diamond E'$ indicates that if a declaration `Dcl` is processed under the type constraint system C and syntactic type assignment E, then the richer syntactic type assignment E' results. For instance, the rule `VarDecl` below asserts that processing a variable declaration of the form $x{:}\,\tau$ results in adding $x{:}\ \texttt{ref}\ \tau$ to the initial syntactic type assignment.

$$VarDcl \qquad \frac{C \vdash \tau : TYPE \qquad x \notin dom(E)}{C, E \vdash_s x{:}\,\tau \ \diamond\ E \cup \{x{:}\ \texttt{ref}\ \tau\}}.$$

Type-assignment rules for terms are of the form $C, E \vdash_s M{:}\,\tau$, indicating that M has type τ if free identifiers are constrained by the assumptions in C, E. If M is a command then the type τ will be `COMMAND`.

The subscript s on $\vdash_s$ represents a subset of `Loc` and is a technical device that is used in the proof of the subject reduction theorem. It essentially constrains the deduction to only involve locations in s. The set s is only used explicitly in the rule *Location*, discussed later, where it is used to ensure that the location being type-checked is in $dom(s)$. For the purposes of type-checking terms, the s may safely be ignored.

The type-assignment rules for the commands and most non-object-oriented expressions are standard, whereas those for the object-oriented expressions may require some extra explanation. In the following we discuss the most interesting rules. For convenience, those discussed below are repeated in Figure 11.

The type-assignment rule *Class* involves several interesting features. As we noted earlier, we have eliminated the use of keywords `MyType` and `self` in classes in favor of using a type parameter and a regular parameter of that type. We have also introduced another parameter `selfinst` and a type variable `InstType`, representing the record of instance variables and its type. Thus the types of instance variables must be of the form $\forall\ \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \tau).\sigma_i$ if the type of that instance variable in the class type is σ_i. Similarly, type-checking will ensure that the types of methods are of the form $\forall\ \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \tau).\forall\ \texttt{InstType}\ <: RecToMem(\sigma).\ \texttt{MyType} \rightarrow\ \texttt{InstType} \rightarrow \tau_j$. That is, the expanded form of methods will take the parameters: the type `MyType`, the type `InstType`, the value `self` with type `MyType`, and the value `selfinst` with type `InstType`, returning a value with the declared type of the method, τ_j. As discussed earlier, the constraints on the parameters corresponding to `MyType` and `InstType` ensure that the instance variables and methods will continue to be type-correct in subclasses.

We repeat that the σ and τ appearing in the resulting class type are abbreviated from the actual types of the instance variables and methods by only showing the resulting type after applying the parameters representing

$$\textit{Class}\quad \frac{\begin{array}{c} C, E \vdash_s M_a\colon \{iv_i\colon \forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \tau).\sigma_i, \}_{i\le k} \\ C, E \vdash_s M_b\colon \{m_j\colon \forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \tau). \\ \qquad\qquad \forall \texttt{InstType} <\colon \mathit{RecToMem}(\sigma).\texttt{MyType} \to \texttt{InstType} \to \tau_j\}_{j\le n} \end{array}}{C, E \vdash_s \texttt{class}(M_a, M_b)\colon \texttt{ClassType}(\texttt{MyType}, \sigma, \tau)}$$

where $\tau = \{m_j\colon \tau_j\}_{j\le n}$, $\texttt{InstType} \notin FV(\tau)$, $\sigma = \{l_i\colon \sigma_i\}_{i\le k}$, and
$\mathit{RecToMem}(\{l_i\colon \sigma_i\}_{i\le k}) = \{l_i\colon \texttt{ref}\ \sigma_i\}_{i\le k}$

$$\textit{Inherits}\quad \frac{\begin{array}{c} C, E \vdash_s M\colon \texttt{ClassType}(\texttt{MyType}, \{iv_i\colon \sigma_i\}_{i\le n}, \{m_j\colon \tau_j\}_{j\le k}) \\ C \cup \{\texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon \tau'_j\}_{j\le k+1})\} \vdash \tau'_1 <\colon \tau_1 \\ C, E \vdash_s M_1^V\colon \sigma_1^\forall \\ C, E \vdash_s M_{n+1}^V\colon \sigma_{n+1}^\forall \\ C, E \vdash_s M_1^f\colon \{m_j\colon \tau_j^\forall\}_{j\le k} \to \delta_1 \\ C, E \vdash_s M_{k+1}^f\colon \{m_j\colon \tau_j^\forall\}_{j\le k} \to \delta_{k+1} \end{array}}{\begin{array}{c} C, E \vdash_s \texttt{class inherit}\ M\ \texttt{modifying}\ iv_1, m_1; \\ (\{iv_1 = M_1^V\colon \sigma_1^\forall, iv_{n+1} = M_{n+1}^V\colon \sigma_{n+1}^\forall\}, \\ \{m_1 = M_1^f\colon \{m_j\colon \tau_j^\forall\}_{j\le k} \to \delta_1, \\ m_{k+1} = M_{k+1}^f\colon \{m_j\colon \tau_j^\forall\}_{j\le k} \to \delta_{k+1}\}): \\ \texttt{ClassType}(\texttt{MyType}, \{iv_i\colon \sigma_i\}_{i\le n+1}, \{m_i\colon \tau'_i\}_{i\le k+1}) \end{array}}$$

where $\sigma_i^\forall = \forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon \tau'_j\}_{j\le k+1}).\sigma_i$, for $1 \le i \le n+1$,
$\tau_l^\forall = \forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon \tau_j\}_{j\le n}).$
$\qquad \forall \texttt{InstType} <\colon \mathit{RecToMem}(\{iv_i\colon \sigma_i\}_{i\le n}).\texttt{MyType} \to \texttt{InstType} \to \tau_l$, for $1 \le l \le k$,
$\delta_p = \forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon \tau'_j\}_{j\le k+1}).$
$\qquad \forall \texttt{InstType} <\colon \mathit{RecToMem}(\{iv_i\colon \sigma_i\}_{i\le n+1}).\texttt{MyType} \to \texttt{InstType} \to \tau'_p$, for $1 \le p \le k+1$
$\tau'_j = \tau_j$ for $2 \le j \le k$, and
$\texttt{InstType} \notin FV(\tau'_1) \cup FV(\tau'_{k+1})$

$$\textit{New}\quad \frac{C, E \vdash_s M\colon \texttt{ClassType}(\texttt{MyType}, \sigma, \tau)}{C, E \vdash_s \texttt{new}\ M\colon \texttt{ObjectType}(\texttt{MyType}, \tau)}$$

$$\textit{Msg}\quad \frac{C \vdash \gamma <\# \texttt{ObjectType}(\texttt{MyType}, \{m\colon \tau\}) \qquad C, E \vdash_s M\colon \gamma}{C, E \vdash_s M \Leftarrow m\colon (\tau[\texttt{MyType} \mapsto \gamma])}$$

$$\textit{Subsump}\quad \frac{C \vdash_s \sigma <\colon \tau \qquad C, E \vdash_s M\colon \sigma}{C, E \vdash_s M\colon \tau}$$

Fig. 11. Selected type-checking rules.

keywords of the language. This corresponds better to the types observable from the concrete syntax shown in the previous section. Recall the discussion from the beginning of this section (and compare Figures 2 and 10) on the difference between the concrete syntax used in the earlier examples and the abstract syntax used in this section. Although the type-checking rule for classes in the abstract language appears very complex because of the extra parameters added to instance variables and methods, the rule can be expressed in a much simpler way in our original concrete syntax:[5]

$$\textit{Class}\quad \frac{\boldsymbol{C}^{\boldsymbol{IV}}, \boldsymbol{E} \vdash_s \boldsymbol{M}_{\boldsymbol{inst}}\colon \sigma,\ \ \boldsymbol{C}^{\boldsymbol{METH}}, \boldsymbol{E}^{\boldsymbol{METH}} \vdash_s \boldsymbol{M}_{\boldsymbol{meth}}\colon \tau}{\boldsymbol{C}, \boldsymbol{E} \vdash_s \ \ \texttt{class}(\boldsymbol{M}_{\boldsymbol{inst}}, \boldsymbol{M}_{\boldsymbol{meth}})\colon\ \ \texttt{ClassType}(\sigma, \tau)},$$

[5]The reason for using the more complex abstract syntax makes the later proof of subject reduction easier.

where

—$C^{IV} = C \cup \{\texttt{MyType} <\texttt{\#}\ \texttt{ObjectType}\ \tau\}$;
—$C^{METH} = C^{IV} \cup \{\texttt{InstType} <: \mathit{RecToMem}(\sigma)\}$; and
—$E^{METH} = E \cup \{\texttt{self: MyType, selfinst: InstType}\}$.
—Neither `MyType` nor `InstType` may occur free in C or E, and τ must be the type of a record of functions.

Here we have retained the separation of the two uses of `self` so that `self` is used as the receiver of messages and `selfinst` is used to access instance variables.

This rule should be significantly easier to understand for the programmer as instance variables and methods do not have to be written as polymorphic functions in the concrete syntax. The abstract syntax presented here is primarily needed in this article to prove the correspondence between type-checking rules and the semantics. The correctness of this simplified rule is discussed in Appendix E. The intuition behind it is that the extra parameters added to instance variables and methods in the abstract syntax are removed and pushed into the type constraint system C and identifier type assignments.

The type-assignment rule *Inherits* for subclasses (classes with *inherit* clauses) is similar to *Class*, but requires type-checking only the new or modified components.[6] It is not necessary to type-check inherited methods or instance variables, since they were already type-checked under assumptions that are still valid in the subclass. (In fact, even stronger assumptions hold in the subclass.) Unlike Java 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 subclass rule first requires type-checking the superclass, and then making sure that the types of overridden methods are subtypes of the corresponding types of the superclass. Because instance variables of classes are associated with initial values, PolyTOIL allows the programmer to provide new initial values to instance variables inherited from the supertype.

The subclass rule looks significantly more complicated than the class rule, but most of the complications are due to notation. For example, type-checking of new initial values of instance variables is exactly as with classes.

The only significant difference between methods of subclasses and classes is that methods in the subclasses take an extra first parameter, typically written as `super`, representing the record of methods of the superclass. This provides the ability to access methods of the superclass in the subclass. Note that a message send to `super` must be of the form `super.`m `[MyType] [InstType] (self) (selfinst)` in the abstract syntax because the type of m in the record of methods from the superclass takes those four parameters.

[6]For simplicity, the type-checking rule for subclasses is given only for the special case where the value of the first instance variable and the first method are overridden, and only one new instance variable and method are added. The general case is similar, but notationally much messier.

Again, a much simpler type-checking rule for subclasses is available for the original concrete syntax:

$$\textit{Inherits} \quad \frac{\begin{array}{c} C, E \vdash_s M\colon\ \texttt{ClassType}(\{iv_i\colon\sigma_i\}_{i\le n}, \{m_j\colon\tau_j\}_{j\le k}), \\ C^{IV} \vdash_s \tau_1' <\colon \tau_1,\ \ C^{IV}, E \vdash_s M_1^V\colon\sigma_1,\ \ C^{IV}, E \vdash_s M_{n+1}^V\colon\sigma_{n+1}, \\ C^{METH}, E^{METH} \vdash_s M_1^f\colon\tau_1',\ \ C^{METH}, E^{METH} \vdash_s M_{k+1}^f\colon\tau_{k+1} \end{array}}{\begin{array}{l} C, E \vdash_s \ \texttt{class inherit}\ M\ \ \texttt{modifying}\ iv_1, m_1 \\ \qquad \big(\{iv_1 = M_1^V\colon\sigma_1, iv_{n+1} = M_{n+1}^V\colon\sigma_{n+1}\}, \\ \qquad \{m_1 = M_1^f\colon\tau_1', m_{k+1} = M_{k+1}^f\colon\tau_{k+1}\}\big)\colon \\ \qquad \texttt{ClassType}(\{iv_i\colon\sigma_i\}_{i\le n+1}, \{m_j\colon\tau_j'\}_{j\le k+1}) \end{array}},$$

where

—$\tau_j' = \tau_j$ for $2 \le j \le k+1$;

—$C^{IV} = C \cup \{\texttt{MyType} <\#\ \ \texttt{ObjectType}\ \{m_j\colon\tau_j'\}_{j\le k+1}\}$;

—$C^{METH} = C^{IV} \cup \{\texttt{InstType} <\colon \mathit{RecToMem}(\{iv_i\colon\sigma_i\}_{i\le n+1})\}$, and

—$E^{METH} = E \cup \{\texttt{self}\colon\ \texttt{MyType},\ \ \texttt{selfinst}\colon\ \texttt{InstType}, \text{super}\colon\{\mathrm{m_j}\colon\tau_{\mathrm{j}}\}_{\mathrm{j}\le\mathrm{k}}\}$.

—Neither MyType nor InstType may occur free in C or E.

The type-checking rule for message sending is interesting in that it uses the matching relation. If $C \vdash \gamma <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon\tau_j\})$ and $C, E \vdash_s M\colon\gamma$ then M is guaranteed to have a method m_j with type a subtype of τ_j. Thus $M \Leftarrow m_j$ is well typed and has a type obtained by replacing all free occurrences of MyType in τ_j by the type of M.

It might seem that a simpler version of the rule given in Figure 11 might be sufficient. For example,

$$\textit{Msg'} \quad \frac{C, E \vdash_s o\colon\ \texttt{ObjectType}\{m_j\colon\tau_j\}_{j\le n}}{C, E \vdash_s o \Leftarrow m_i\colon\tau_i[\texttt{MyType} \mapsto\ \ \texttt{ObjectType}\{m_j\colon\tau_j\}_{j\le n}]} \quad \text{for } i \le n.$$

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 self ⇐ m, where self has type MyType <# ObjectType(MyType, τ). Since we cannot write the type of self in the form ObjectType(MyType, τ'), the simpler rule (*Msg'*) is not applicable, and we must use the more complex rule given in Figure 11 and Appendix A. (Similar difficulties would arise with sending a message to an object whose type is given by a match-bounded type variable.)

The subsumption rule allows one to promote the type of an expression to a supertype, and is used, for example, when one passes an actual parameter to a function where the type of the actual parameter is a subtype of the type given to the formal parameter.

Because instance variables are represented by a parameter whose type is a subtype of a record of references, instance variables can be typed using subsumption and record field extraction.

The last two type-checking rules from Appendix A have to do with constructs that are not available to programmers, but that are useful in specifying the semantics of programs. We postpone the discussion of the rule for closures to Section 4.2.

Now that we know what well-typed programs of PolyTOIL look like, in the next section we define the semantics of PolyTOIL by defining reduction rules for the language.

4. AN OPERATIONAL SEMANTICS FOR POLYTOIL

In this section we provide a natural (operational) semantics for PolyTOIL. This semantics is similar to that given in Bruce et al. [1994] for TOOPLE. In the following section we state and prove a subject reduction theorem, which ties together our type-checking rules and the semantics. A simple corollary is that our type-assignment rules are safe.

We extend the source language from the previous section by additional constructs that may occur during evaluation of a program, although they are not allowed to occur in the original program. The set M of expressions is extended to include the following.

$$\begin{aligned} M \ ::= \ & \ldots \mid \texttt{error} \mid \texttt{tyerr} \mid \mathit{Loc} \mid \texttt{obj}(M_1, M_2) \mid \\ & \langle \texttt{function}(x{:}\tau)\, \mathit{Block}, \rho\rangle \mid \langle \texttt{function}(t{:}\ \texttt{TYPE})\, \mathit{Block}, \rho\rangle \mid \\ & \langle \texttt{function}(t <\#\, \tau)\, \mathit{Block}, \rho\rangle \mid \langle \texttt{function}(t <:\, \tau)\, \mathit{Block}, \rho\rangle, \end{aligned}$$

where *Loc* represents a collection of memory locations.

The expression `error` stands for a runtime error such as dividing by zero or sending a message to `nil`. Because the static type system is not designed to pick up this sort of runtime error, we would like a computation that results in this value to be considered well typed. One option is to introduce a different error term for every type. However, we adopt the (notationally) simpler strategy of having a single error expression, but allow it to be assigned any type. The expression `tyerr` represents a type error, which should never arise during the evaluation of a well-typed program. There are no typing rules applicable to `typerr`. Expressions of the form $\texttt{obj}(M_1, M_2)$ represent the internal view of objects with instance variables M_1 and method suite M_2. The typing rule *Object* for these terms is simpler than the rule for classes because we need not worry about extending objects with subclasses. The type-checking rules for locations and closures are given later in this section.

Because functions are first-class in PolyTOIL (i.e., they may be assigned to variables, passed in as arguments to functions, and returned as values from functions), they are represented internally as closures. That is, an expression f representing a function will be reduced to a pair $\langle f, \rho\rangle$ consisting of the expression itself and the current environment ρ. (Environments are formally defined in the next section.) The environment ρ must contain interpretations of all of the free variables in f. When the closure is actually applied to an argument, it updates the enclosed environment so that the formal parameter is interpreted as the argument, and then evaluates its body using this updated environment.

The natural semantics for PolyTOIL provides a description of the reduction rules for a simple interpreter for the language. A (runtime) environment keeps track of the current values of identifiers, and the store keeps track of what values are stored in currently active locations in memory. The semantic rules reduce an expression M with associated environment ρ 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 to provide a useful language. For example, 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 should only be allocated when a new object is created. Similarly we do not evaluate the initial values of instance variables or method bodies until a new object is created. Thus if the initial value of an instance variable in a class depends on a global variable, it uses the value of that global variable when the new object is created rather than the value when the class is defined. This delayed evaluation will follow automatically from the fact that all instance variables and methods are held as functions of at least one type variable (representing `MyType`).

Another complication is that methods may refer to `self` and `selfinst`. 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.

In the following subsections we provide some technical definitions that allow us to express the natural semantics of PolyTOIL. We then present and explain the semantics.

4.1 Irreducible Values, Environments, and Stores

We begin by specifying the set of *irreducible values*. Irreducible values can be understood as expressions representing the final values of a computation. That is, there are no further computation rules that can be applied to these expressions to simplify them.

Definition 4.1. The set of *irreducible values* (*IrrVal*) in PolyTOIL is the set of expressions that includes constants in $\mathcal{B}$, locations, error, tyerr, closures (representing functions), and record, class, and object values with irreducible components.

A type expression τ is *closed* if $\emptyset \vdash \tau$: `TYPE`.

Environments determine the values of free type and term variables at runtime. All values of term variables in an environment are irreducible values.

Definition 4.2. An *environment* ρ (we also use κ, η for environments) is a finite mapping of type variables to closed type expressions and of term variables to irreducible values.

A store determines the values stored in locations at runtime. As with environments, the values stored are irreducible values.

Definition 4.3. A *store* s is a finite mapping of locations in *Loc* to irreducible values.

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_τ for the

collection of locations holding values of closed type τ, and interpret the type `ref` τ as Loc_τ. It would be possible instead to have a single collection of locations and keep a "store environment" that keeps track of the intended types of allocated locations, but the approach used here is notationally a bit simpler.

The type-checking rule for locations, *Location*, is the only rule that explicitly depends on the state s. If location `l` is in Loc_τ and is in the domain of s then `l` has type `ref` τ. Because of this dependency of the type-checking rule on the state, all of the type-checking rules are written using the notation $\vdash_s$.

The store is infinitely extensible. The function *GetNewLoc* is used to allocate new memory locations. The new memory values are provided with a value of the appropriate type to keep the memory consistent.

Definition 4.4. The function *GetNewLoc* takes a store s, a type τ, and an irreducible value V, and returns an unused location $l^* \in Loc_\tau$ along with a new store $s[l^* \mapsto V]$. $s^* = s[l^* \mapsto V]$ is defined so that $dom(s^*) = dom(s) \cup \{l^*\}$, and for $l \in dom(s)$, $s^*(l) = s(l)$ and $s^*(l^*) = V$.

4.2 Substitutions Induced by Environments and Type-Checking Closures

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 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 substitute in the values of type variables from the environment as part of the computation. We use the notations M_ρ and τ_ρ to stand for the result of replacing free variables in an expression M and type expression τ by the values assigned to them by the environment ρ. The long, but straightforward, formal definition of these substitutions may be found in Appendix B.

Our subject reduction theorem shows that if we start out with a well-typed term M, appropriate environment ρ, and state s, the resulting value V can be assigned the same type as the original. For this to be true, we will need to be certain that the state and environment are consistent with any assumptions made in the typing of M.

Definition 4.5.
1. The state s is *consistent*, written $\models s$, if and only if for all $l \in Loc_\tau \cap dom(s)$, $\emptyset, \emptyset \vdash_s s(l) : \tau$.
2. The environment ρ is *consistent* with C, E, s, written $\rho \models_s C, E$, if and only if $(dom(C) \cup dom(E)) \subseteq dom(\rho)$ and
   (a) For all $t \in dom(\rho)$, $(t$: `TYPE`$) \in C$ implies $\emptyset \vdash \rho(t)$: `TYPE`;
   (b) For all $t \in dom(\rho)$, $(t <: \tau) \in C$ implies $\emptyset \vdash \rho(t) <: \tau_\rho$;

(c) For all $t \in dom(\rho)$, $(t <\# \tau) \in C$ implies $\emptyset \vdash \rho(t) <\# \tau_\rho$;
(d) For all $x \in dom(\rho)$, $(x{:}\tau) \in E$ implies $\emptyset, \emptyset \vdash_s \rho(x){:}\tau_\rho$.

Thus a state s is consistent if and only if for all τ, all locations simultaneously in Loc_τ and s's domain actually hold values of type τ. The consistency of an environment with respect to C, E, and s depends on the fact that the interpretations of the type and term variables are consistent with the assumptions in C and E.

With these definitions, we are now ready to discuss the type assignment rule for closures. The following is rule *Closure* from Appendix A.

$$\textit{Closure} \qquad \frac{\hat{C}, \hat{E} \vdash_s f{:}\sigma \to \tau \text{ and } \rho \models_s \hat{C}, \hat{E}}{C, E \vdash_s \langle f, \rho\rangle{:}\sigma_\rho \to \tau_\rho}.$$

In a closure $\langle f, \rho\rangle$, the environment ρ contains interpretations for all free variables (term and type) in f. As a result, the values of C and E are irrelevant in determining the type of $\langle f, \rho\rangle$. Instead we create a new type constraint system $\hat{C}$, and syntactic type assignment $\hat{E}$, which are consistent with ρ and type-check f with respect to these systems. That is, create $\hat{C}, \hat{E}$ so that $dom(\hat{C}) \cup dom(\hat{E}) \subseteq dom(\rho)$ and $\rho \models_s \hat{C}, \hat{E}$. If $\hat{C}, \hat{E} \vdash_s f{:}\sigma \to \tau$ then $C, E \vdash_s \langle f, \rho\rangle{:}\sigma_\rho \to \tau_\rho$. The reason for the change in type is that the environment ρ may contain values for type variables in $\sigma \to \tau$ and these need to be reflected in the final type of the closure.

We emphasize again that, because closures contain no free variables (they are all bound in the enclosed environment), the type-checking takes place with respect to a type constraint system and type assignment that are consistent with respect to the enclosed environment.

The following lemma shows that when ρ is consistent with C, E, we can relativize proofs of subtyping, matching, and type-checking. This will be useful in proving subject-reduction.

LEMMA 4.6. *(Substitution) Let* $\rho \models_s C, E$.

1. $C \vdash \tau$: TYPE *implies* $\emptyset \vdash \tau_\rho$: TYPE.
2. $C \vdash \sigma <: \tau$ *implies* $\emptyset \vdash \sigma_\rho <: \tau_\rho$.
3. $C \vdash \sigma <\# \tau$ *implies* $\emptyset \vdash \sigma_\rho <\# \tau_\rho$.
4. $C, E \vdash_s M{:}\tau$ *implies* $\emptyset, \emptyset \vdash_s M_\rho{:}\tau_\rho$.

PROOF. The proof proceeds by a simple induction on complexity of the deduction of the left-hand side. The base case for variables in each case follows from the definition of consistency of ρ. □

4.3 Natural Semantics Rules for PolyTOIL

The natural semantics rules for PolyTOIL can be found in Appendix C. A triple of a declaration, environment, and store reduces (by $\downarrow_{decl}$) to a pair of a new environment and state. That is, $(D, \rho, s) \downarrow_{decl} (\rho', s')$. Programs, blocks, statements, and expressions (with initial environment and state) all reduce (by $\downarrow$) to pairs of irreducible values and states. That is, $(M, \rho, s) \downarrow (v, s')$, where v is an irreducible value. Programs reduce to a default value *ok*, of type PROGRAM,

and statements reduce to a default value *command*, of type `COMMAND`. Thus, as usual, the result of executing a statement is essentially just the updated state.

Because we wish to maintain a consistent state, allocation of new memory via *GetNewLoc* (e.g., in variable declarations and expressions of the form `new c`) requires the type of memory being allocated and an initial value of that type. Since all instance variables are given initial values in class expressions, these can be initialized easily. Other variables need not have initial values declared, so they are initialized with an error value, which by type-checking rule *Error*, can be assigned any type. (Alternatively, we could have insisted that all variable declarations provide an initial value.)

The reduction rules for declarations and statements are relatively straightforward. As expected, the reduction rules for while loops are recursive.

Constants, locations, and closures are irreducible values and so are unchanged by reduction. Variables reduce to the value given by the current environment. If `M` is an expression representing a location (e.g., a variable) then `val M` is reduced by first evaluating `M` to a location and then returning the value stored in that location (as found in the state).[7]

All four types of function expressions reduce to closures using the four rules starting with rule $Function_{Comp}$. Regular function applications proceed using call-by-value according to rule $FuncAppl_{Comp}$ by first reducing the function to a closure and the argument to a value. The environment of the closure is updated to interpret the formal parameter as the actual parameter's value. This new environment is then used in reducing the body of the function.

Polymorphic functions are applied to type expressions. Although there are no reduction rules for type expressions, we must handle properly any type variables contained in the type parameter. To ensure that only closed types are associated with values in the environment, in rule $PolyFuncAppl_{Comp}$, for example, we replace any type variables occurring in the actual type 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.

Record expressions are reduced in rule $Record_{Comp}$ by evaluating each of the fields of the record. Notice that the types of the fields are also evaluated by replacing free type variables by the values assigned by the environment.

Class and object expressions are reduced by evaluating the initialization code for instance variables and method expressions. Because each of these is represented by a function, the results are records of closures.

The reduction rule for `new M` is the most complex. We repeat it in Figure 12 for ease of reference. In evaluating `new M`, the expression `M` is first reduced to an irreducible value of the form `class(IV,Methods)`. Since initial values of instance variables are closures of the form $\langle$`function(MyType <#` $\hat{\gamma}_i$`)` $B_i, \eta_i\rangle$,

[7]We have been rather cavalier in our treatment of errors in these rules. For example, in the $FuncAppl_{Comp}$ rule, we should add that this rule returns `V` only if $\texttt{V}_2 \neq$ `error`. We omitted inserting these qualifiers in these rules to avoid making the computation rules even harder to read. In Appendix C.2, we indicate how these rules should be patched to ensure the propagation of errors.

$$New_{Comp}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(\texttt{class}(\{iv_i=\langle\texttt{function}(\texttt{MyType}<\#\,\hat{\gamma}_i)\ B_i,\eta_i\rangle{:}\,\sigma_i^{\forall}\}_{i\le n}\\ \{m_j=\langle\texttt{function}(\texttt{MyType}<\#\,\hat{\gamma}_j)\ B_j,\kappa_j\rangle{:}\,\tau_j^{\forall}\}_{j\le k})\ ,\ s_1)\\ \vdots\\ (B_i,\eta_i[\texttt{MyType}\mapsto\texttt{ObjectType}(\texttt{MyType},\tau_\rho)],s_i)\downarrow(V_i,s_{i+1})\\ \vdots\\ (newL_i,s_{n+i+1})=(GetNewLoc\ \ s_{n+i}\ \ (\sigma_i)_\rho[\texttt{MyType}\mapsto\texttt{ObjectType}(\texttt{MyType},\tau_\rho)]\ \ V_i)\\ \vdots\\ (B_j,\kappa_j[\texttt{MyType}\mapsto\texttt{ObjectType}(\texttt{MyType},\tau_\rho)],s_{2n+2(j-1)+1})\\ \downarrow(\langle\texttt{function}(\texttt{InstType}<{:}\,\hat{\sigma}_j)\ \hat{B}_j,\hat{\kappa}_j\rangle,s_{2n+2j})\\ (\hat{B}_j,\hat{\kappa}_j[\texttt{InstType}\mapsto RecToMem(\sigma_\rho)],s_{2n+2j})\downarrow(\langle\texttt{function}(\texttt{self}{:}\,\varsigma_j)\ \breve{B}_j,\breve{\kappa}_j\rangle,s_{2n+2j+1})\\ \vdots\end{array}}{(\texttt{new}\ M,\rho,s)\downarrow(ob,s_{2n+2k+1})}$$

where
$\sigma=\{iv_i{:}\,\sigma_i\}_{i\le n}$, $\tau=\{m_j{:}\,\tau_j\}_{j\le k}$,
$C,E\vdash_s M{:}\,\texttt{ClassType}(\texttt{MyType},\sigma,\tau)$,
$\emptyset\vdash\sigma_i^{\forall}<{:}\,\forall\texttt{MyType}<\#\,\texttt{ObjectType}(\texttt{MyType},\tau_\rho).(\sigma_i)_\rho$, for $i\le n$,
$\emptyset\vdash\tau_j^{\forall}<{:}\,\forall\texttt{MyType}<\#\,\texttt{ObjectType}(\texttt{MyType},\tau_\rho).\forall\texttt{InstType}<{:}\,RecToMem(\sigma_\rho).$
$\texttt{MyType}\to\texttt{InstType}\to(\tau_j)_\rho$, for $j\le k$,
and
$ob=\texttt{obj}(\{iv_i=newL_i{:}\,\texttt{ref}\ (\sigma_i)_\rho[\texttt{MyType}\mapsto\texttt{ObjectType}(\texttt{MyType},\tau_\rho)]\}_{i\le n},$
$\{\ m_j=\langle function(\texttt{self}{:}\,\varsigma^j)\ \breve{B}^j,\breve{\kappa}^j\rangle{:}\,(\texttt{MyType}\to RecToMem(\sigma_\rho)\to(\tau_j)_\rho)$
$[\texttt{MyType}\mapsto\texttt{ObjectType}(\texttt{MyType},\tau_\rho)]\}_{j\le k}).$

Fig. 12. Reduction rule for new expressions.

representing functions parameterized by MyType, we apply each of these functions to the meaning of MyType, which is the type ObjectType(MyType, τ_ρ).

We accomplish this application by augmenting the environment of the closure by interpreting MyType by ObjectType(MyType, τ_ρ), where ObjectType (MyType, τ) is the type of object generated by M, and ρ is the environment in which new M is evaluated. The body of the function B_i, representing the initial value of the instance variable, is then evaluated in this augmented environment to give a result V_i. The function *GetNewLoc* is used to allocate new locations $newL_i$ for each of the instance variables of the object and the initial values V_i of the instance variables are stored in those locations.

The methods are evaluated similarly by applying each of the methods of the class successively to the intended values of MyType and InstType and reducing. Note that the resulting methods are functions expecting values for self and selfinst. The term ob, an object whose instance variables are set to be the new locations $newL_i$ holding the initial values specified by the class and whose methods are obtained as described above, is the result of evaluating new M.

Message sends are reduced in rule Msg_{Comp} by extracting the appropriate method body for the method call from the object, supplying appropriate values for self and selfinst (recall the methods needed values for self and selfinst), evaluating, and finally returning the result (which is a closure as methods always have functional type) as the value.

Extracting the value of instance variables from inside a method simply involves extracting the location from selfinst, the record of instance variables, so no new semantic rule beyond that given to extract fields of a record is needed.

The semantics of inheritance is not quite the obvious one. That is, one would expect the new or modified values would be added to or replace the

corresponding values in the original class. However, we also rebind the type bounds of type parameters in inherited methods and instance variables with a constraint more appropriate for the subclass. As suggested in the type-checking rules, `super` is interpreted inside each of the methods as the record of methods of the superclass.

Appendix C.2 contains the semantic rules that generate runtime errors. The object `nil` responds to all messages with a value `error`. The other rules ensure that errors are propagated during computations.

We discuss the computation rules in Appendix C.3 in the next section, where we discuss the type safety of the computation rules.

5. SUBJECT REDUCTION THEOREM

In this section we state the subject reduction theorem for PolyTOIL, and use it to prove the type safety of the language. The details of the actual proof of subject reduction can be found in Appendix D. Roughly, the subject reduction theorem states that reduction (computation) preserves types.

THEOREM 5.1 (SUBJECT-REDUCTION). *Let* $\rho \models_s C, E$ *and* $\models s$.

1. *If* $C, E \vdash_s M \diamond E^*$ *and* $(M, \rho, s) \downarrow_{decl} (\rho^*, s^*)$ *then* $\rho^* \models_{s^*} C, E^*, \models s^*$ *and* $dom(s) \subseteq dom(s^*)$.
2. *If* $C, E \vdash_s M : \tau$ *and* $(M, \rho, s) \downarrow (V, s^*)$ *then* $\emptyset, \emptyset \vdash_{s^*} V : \tau_\rho$, $dom(s) \subseteq dom(s^*)$ *and* $\models s^*$.

Part one of the theorem states that the runtime environment resulting from processing declarations is consistent with the type assignment generated by the type-checker. Part 2 states that if a well-typed term is reduced, then, if the reduction terminates, the resulting irreducible value has a type corresponding to that of the original term.

The theorem is stated carefully to take care of the case where the type τ of the original term M involves one or more type variables. If M reduces to an irreducible value V, then that value cannot involve any free term or type variables. Hence one would expect V to have a type that is some instantiation of τ. Because the environment ρ that M is reduced in has interpretations for all type variables occurring in M, it is used to instantiate the type variables in τ. Thus the theorem asserts that the type of the irreducible value is the type τ in which all of the type variables have been replaced by their interpretations in ρ.

Note that if M evaluates to V then the "minimal" type of V may be a subtype of the "minimal" type of M. A simple example is the following. Let $A <: B$, let id_B be the identity function taking arguments of type B, and let a have type A. Then the "minimal" type of $id_B(a)$ is B, yet $id_B(a)$ clearly reduces to a with type A.

The proof of subject reduction is given in Appendix D. It is a proof by induction on the depth of the computation tree. Most cases are straightforward. The most complex and interesting cases are for function applications and for terms involving constructing new objects (i.e., of the form `new c`), message passing, and subclasses.

The subject reduction theorem shows that our type assignment rules are sound in that computations preserve the types of terms. We now wish to prove something a bit stronger: that computations of well-typed terms never get stuck, that is, never produce a type error. In particular, a computation of a well-typed term will never result in sending a message m to an object that does not have m as a method. We say a computation from (M, ρ, s) becomes *stuck* if in one of its computation subtrees a hypothesis of the form $(M_1, \rho_1, s_1) \downarrow (V_1, s_1^*)$ must be established, but $(M_1, \rho_1, s_1) \downarrow (V^*, s^*)$ and V^* does not have the same form as V_1.

We provide the reader with some examples of when a computation might get stuck. First, we try to evaluate the application 0(0) of the number 0 to itself in a given environment ρ and state s. The only rule whose conclusion describes how to derive a value for an application is the rule $FuncAppl_{Comp}$. We must establish the hypothesis $(0, \rho, s) \downarrow (\langle \texttt{function}(x{:}\hat{\sigma})\ B, \eta\rangle, s_1)$, but clearly $(0, \rho, s) \downarrow (0, s)$ (using $Constant_{Comp}$).

As a second example, suppose we try to evaluate the result of sending a message `changeMood` to an object of class `helloClass` in Figure 1, for example, evaluating `new helloClass` $\Leftarrow$ `changeMood` when started with environment ρ and state s. The only rule whose conclusion describes how to derive a value for a message send is the rule Msg_{Comp}. We have no problems with establishing the hypothesis $(\texttt{new helloClass}, \rho, s) \downarrow (\texttt{ob}, s_1)$, but `ob` is an object that does not contain the label `changeMood` in the record of object methods. In both cases a runtime type error will occur. Of course in our type system, terms of the form 0(0) and `new helloClass` $\Leftarrow$ `changeMood` fail to type-check, and hence are illegal.

Our proof that reductions of well-typed terms do not become stuck is based on the presentation in Section 7.2 from Gunter [1992]. We make a few changes to our natural semantics. We do this by ensuring that we have appropriate computation rules for each construct to encompass all of those cases that are not currently handled by the semantics. In each of these cases, a new constant, `tyerr`, will be the result of the computation. For example, in the evaluation of a term of the form $M(N)$, if M does not evaluate to a function closure, then $M(N)$ will evaluate to `tyerr`. A selection of these rules can be found in Appendix C.3. It is somewhat tedious, but straightforward to write all of these rules. (The rules can even be generated mechanically, since new rules are written for cases when an original rule does not apply.) We consider `tyerr` to be an (untyped) irreducible value. It is now straightforward to show that computations starting from well-typed terms never get stuck.

We call the system that includes the rules for type errors the *extended natural semantics* of PolyTOIL and use the notation $\downarrow^+$ for its computation. Except for the error terms (type and otherwise), we presume that all constants of the language have a base type. We also assume that the only values of type `Bool` are `true`, `false`, and `error`. It would also be possible to add new computation rules for constants of functional types (e.g., arithmetic operators on numbers), but we satisfy ourselves with this simplified version here. We can now prove a subject reduction theorem for this new system.[8]

Using this new subject reduction theorem, it follows that

[8]Actually the proof of subject reduction given in the appendix is for this extended natural semantics.

COROLLARY 5.2 (TYPE SAFETY). *Let* $\rho \models_s C, E$ *and* $\models s$. *If* $C, E \vdash_s M : \tau$ *then it is not the case that* $(M, \rho, s) \downarrow^+ (tyerr, s^*)$.

This corollary follows directly from the subject reduction theorem for our extended natural semantics. If $(M, \rho, s) \downarrow^+ (\texttt{tyerr}, s^*)$ then it would follow from subject reduction that $C, E \vdash_s \texttt{tyerr} : \tau_\rho$, which is impossible since `tyerr` cannot be assigned a type. This contradiction shows that no type error can arise during the computation of a well-typed term.

Of course, we cannot ensure that those computations will actually terminate. Simple examples of nonterminating computations include `while` statements with `true` as the Boolean guard, and expressions involving sending a message to an object whose corresponding method body sends the same message to `self`.

6. 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++, Java, 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 one chooses to identify these hierarchies then one is either left with an unsafe language (either by design, or with conventions that require many type casts, essentially bypassing the type system), a language with limited expressiveness, or a language that requires extra runtime or linktime checks in order to preserve type safety.

PolyTOIL's type system allows much greater flexibility than languages like C++, Java, and Object Pascal, which do not support a special name for the type of `self` or allow the modification of types of methods in subclasses. The lack of these features greatly limits the expressiveness of these languages, particularly in defining subclasses.

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 `DNodeType` to be a subtype of `NodeType` in the example in Figure 2. This design leads to type errors unless another mechanism, such as a linktime 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 that offer both increased expressiveness and a safe type system. An example of this impact is the language Strongtalk [Bracha and Griswold 1993], which essentially has adopted the typing rules for our earlier language TOOPLE [Bruce 1994] (along with a few extensions) to type-check a subset of Smalltalk.

The language Theta [Day et al. 1995] was developed independently of this work, but shares many of the features of PolyTOIL. In particular it supports a mechanism for constraining polymorphism that 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 article can be used to add flexibility to their type system, while providing a proof of its type safety.

The theoretical work most similar to that described here is Eifrig et al. [1994]. That paper presents an analysis of a statically typed object-oriented language, LOOP, which is similar to the nonpolymorphic parts of PolyTOIL. 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 (although 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 that involve `MyType` generally do not have subtypes. (More recent work on LOOP has resulted in a more flexible type system that does capture more subtypes of object types.)

LOOP does not explicitly identify the concept of "matching," although it appears that the concept is implicitly supported in the type-checking rules for methods. It would be interesting to design a language that 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, whereas we provide a more direct natural semantics for PolyTOIL that includes objects as primitives.

Other researchers have performed interesting investigations of imperative object-oriented languages by looking at translations into simpler calculi. Pierce [1993], has extended his encoding of object-oriented languages in higher-order bounded lambda calculi [Pierce and Turner 1994] to imperative languages. The semantics are 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 expressiveness in that methods like the `setNext` method of `NodeClassType` in Figure 2, which have parameters of type `MyType`, may not be written as methods of the class.

Abadi and Cardelli [1994b,a, 1995] have written several papers investigating a series of low-level object calculi designed to serve as a foundation for higher-level object-oriented programming languages. The same authors show

in their monograph [Abadi and Cardelli 1996] 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.

More recent work includes Igarashi et al.'s [1999] development of Featherweight Java as a testbed for experimenting with extensions of Java. Because the core language is very simple, it is possible to prove type safety and other theorems about simple extensions to this language.

A language $\mathcal{PSOOL}$, similar to PolyTOIL, is described in the book, *Foundations of Object-Oriented Languages: Types and Semantics* [Bruce 2002], by the first author. That book provides a very different translational semantics for the language. It also provides a very extensive list of references for related work on the foundations of object-oriented languages.

The designs of TOOPLE, TOIL, and PolyTOIL were inspired by work over the last 10 years in the study of typed object-oriented languages by the theoretical programming languages community. Early analysis by Cardelli [1988] (originally presented in 1984) led to the influential paper by Cardelli and Wegner [1985], which introduced the construct of bounded quantification as a means of modeling object-oriented features. The study of typed languages in Cook et al. [1990] clearly explained the differences between subtyping and inheritance, and proposed a way of modeling inheritance using bounded quantification. Meanwhile Mitchell [1990] presented an operational semantics for delegation-based object-oriented languages. (See also more recent work in Fisher et al. [1998] and Fisher and Mitchell [1998].) The paper by Canning et al. [1989] pointed out the necessity of using a more general notion of bounded quantification, called F-bounded quantification, in order to truly model features of object-oriented programming languages.

A series of other papers [Bruce and Longo 1990; Amadio 1991; Cardone 1989; Abadi and Plotkin 1990; Bruce and Mitchell 1992] provided models sufficient for interpreting the denotational semantics of higher-order bounded calculi, and hence for the object-oriented languages that were encoded in these calculi.

It was only with the strong theoretical understanding of the semantic underpinnings of object-oriented languages that we were able to make the progress represented by this article on the design of type-safe object-oriented programming languages.

7. FURTHER RESULTS AND EXTENSIONS TO POLYTOIL

We have proved the decidability of a type-checking algorithm for PolyTOIL that is similar to the one described in Bruce et al. [1993]. Using this algorithm we have built a prototype interpreter for PolyTOIL that is based on the natural semantics given here.

With the assistance of Leaf Petersen and Jasper Rosenberg, then students at Williams College, we have added new features to the language, improved the interpreter, and improved 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 [Katiyar et al. 1994]), and programmer control over visibility of methods. We also added

a new subtyping rule similar to that in Eifrig et al. [1994] that allows us to deduce more subtype relations between object types. The new rule states that if $\sigma <\# \tau$ and all occurrences of `MyType` in τ are positive, then $\sigma <: \tau$. We have also written and run a large number of PolyTOIL programs that have led us to these changes and have provided us with greater confidence in the strength and flexibility of the language.

The type system for PolyTOIL allows the programmer greater flexibility than most statically typed languages, while providing assurance that the static type-checking rules guarantee type safety. We have been somewhat concerned, however, with the added complexity for the programmer in needing to keep track of two related, yet different, ordering 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.

The paper by Gawecki and Matthes [1996] in ECOOP'96 presented a language TooL, which is more complex than PolyTOIL in that the bound on polymorphic types and classes could be specified using either subtyping or matching, and type-checking of classes could be done assuming that `MyType` either matched or was a subtype of the intended type of objects generated by the class. After experimenting with the language, the authors decided their language was too complex for programmers, and suggested dropping matching.

Because of our experience we have come to the opposite conclusion: subtyping is not as useful as matching (as long as we provide a mechanism to support heterogeneous data structures). Research with Leaf Petersen resulted in the design of an object-oriented language, LOOM, [Bruce et al. 1997] that supports matching, but not subtyping, while still providing sufficient expressiveness for programmers. A key feature of this language is the provision of type expressions that allow the programmer to express that a value can have any type that matches a given expression. We have been at work on this language as a possible successor to PolyTOIL.

APPENDIX

A. TYPE-CHECKING RULES

The rules are defined with respect to a given store s.

OK $$C, E \vdash_s \texttt{ok}: \texttt{PROGRAM}$$

Command $$C, E \vdash_s \texttt{command: COMMAND}$$

Nil $$C, E \vdash_s \texttt{nil}: \texttt{Null}$$

Constant $$\frac{\emptyset \vdash c: \texttt{TYPE} \qquad b \in \Im(c)}{C, E \vdash_s b: c}$$

Error $$\frac{\emptyset \vdash \tau: \texttt{TYPE}}{C, E \vdash_s error: \tau}$$

Location
$$\frac{\emptyset \vdash \tau : \texttt{TYPE}, \quad L \in Loc_\tau \cap dom(s)}{C, E \vdash_s L : \texttt{ref}\ \tau}$$

Variable
$$C, E \vdash_s x : \tau \qquad \text{if} \quad E(x) = \tau$$

Program
$$\frac{C, E \vdash_s B : \texttt{COMMAND}}{C, E \vdash_s \texttt{program}\ p;\ B. : \texttt{PROGRAM}}$$

ConstDecls
$$\frac{C, E \vdash_s CDcl\,Lst \ \diamond\ E*}{C, E \vdash_s \texttt{const}\ CDcl\,Lst \ \diamond\ E*}$$

*ConstDcl**
$$\frac{C, E \vdash_s x = M \ \diamond\ E_1 \qquad C, E_1 \vdash_s CDcl\,Lst \ \diamond\ E_2}{C, E \vdash_s x = M\ ;\ CDcl\,Lst \ \diamond\ E_2}$$

ConstDcl
$$\frac{C, E \vdash_s M : \tau \qquad x \notin dom(E)}{C, E \vdash_s x = M \ \diamond\ E \cup \{x : \tau\}}$$

VarDecls
$$\frac{C, E \vdash_s V\,Dcl\,Lst \ \diamond\ E*}{C, E \vdash_s \texttt{var}\ V\,Dcl\,Lst \ \diamond\ E*}$$

*VarDcl**
$$\frac{C, E \vdash_s x : \tau \ \diamond\ E_1 \qquad C, E_1 \vdash_s V\,Dcl\,Lst \ \diamond\ E_2}{C, E \vdash_s x : \tau\ ;\ V\,Dcl\,Lst \ \diamond\ E_2}$$

VarDcl
$$\frac{C \vdash \tau : TYPE \qquad x \notin dom(E)}{C, E \vdash_s x : \tau \ \diamond\ E \cup \{x : \texttt{ref}\ \tau\}}$$

Block
$$\frac{\begin{array}{c} C, E \vdash_s CDcls \ \diamond\ E_1 \qquad C, E_1 \vdash_s V\,Dcls \ \diamond\ E_2 \\ C, E_2 \vdash_s S : \texttt{COMMAND} \qquad C, E_2 \vdash_s M : \tau \end{array}}{C, E \vdash_s CDcls\ \ V\,Dcls\ \texttt{begin}\ S\ \texttt{return}\ M\ \texttt{end} : \tau}$$

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

Cond
$$\frac{C, E \vdash_s M : \texttt{Bool} \quad C, E \vdash_s S_1 : \texttt{COMMAND} \quad C, E \vdash_s S_2 : \texttt{COMMAND}}{C, E \vdash_s \texttt{if}\ \ M\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2\ \ \texttt{end}\colon\ \texttt{COMMAND}}$$

While
$$\frac{C, E \vdash_s M : \texttt{Bool} \quad C, E \vdash_s S : \texttt{COMMAND}}{C, E \vdash_s \texttt{while}\ \ B\ \ \texttt{do}\ \ S\ \ \texttt{end}\colon\ \texttt{COMMAND}}$$

StmtList
$$\frac{C, E \vdash_s S_1 : \texttt{COMMAND} \quad C, E \vdash_s S_2 : \texttt{COMMAND}}{C, E \vdash_s S_1; S_2 : \texttt{COMMAND}}$$

Value
$$\frac{C, E \vdash_s M : \texttt{ref}\ \tau}{C, E \vdash_s \texttt{val}\ \ M : \tau}$$

Function
$$\frac{C, E \cup \{x:\sigma\} \vdash_s B:\tau}{C, E \vdash_s \texttt{function}(x:\sigma)\ B:\sigma \to \tau}$$

PolyFunction
$$\frac{C \cup \{t:\texttt{TYPE}\}, E \vdash_s B:\tau}{C, E \vdash_s \texttt{function}(t:\texttt{TYPE})\ B:\forall t.\tau}$$

<# BdPolyFunction
$$\frac{C \cup \{t\ <\#\ \gamma\}, E \vdash_s B:\tau}{C, E \vdash_s \texttt{function}(t\ <\#\ \gamma)\ B:\forall t\ <\#\ \gamma.\tau}$$

<: BdPolyFunction
$$\frac{C \cup \{t <: \gamma\}, E \vdash_s B:\tau}{C, E \vdash_s \texttt{function}(t <: \gamma)\ B:\forall t <: \gamma.\tau}$$

FuncAppl
$$\frac{C, E \vdash_s M_1:\sigma \to \tau \quad C, E \vdash_s M_2:\sigma}{C, E \vdash_s M_1(M_2):\tau}$$

PolyFuncAppl
$$\frac{C, E \vdash_s M:\forall t.\tau \qquad C \vdash_s \sigma:TYPE}{C, E \vdash_s M[\sigma]:\tau[t \mapsto \sigma]}$$

<# BdPolyFuncAppl
$$\frac{C, E \vdash_s M:\forall t\ <\#\ \gamma.\tau \qquad C \vdash_s \sigma\ <\#\ \gamma}{C, E \vdash_s M[\sigma]:\tau[t \mapsto \sigma]}$$

<: BdPolyFuncAppl
$$\frac{C, E \vdash_s M:\forall t <: \gamma.\tau \qquad C \vdash_s \sigma <: \gamma}{C, E \vdash_s M[\sigma]:\tau[t \mapsto \sigma]}$$

Record
$$\frac{C, E \vdash_s M_i:\tau_i \text{ and } l_i \in \mathcal{L} \text{ for } 1 \le i \le n}{C, E \vdash_s \{l_i = M_i:\tau_i\}_{i \le n}:\{l_i:\tau_i\}_{i \le n}}$$

Proj
$$\frac{C, E \vdash_s M:\{l_i:\tau_i\}_{i \le n}}{C, E \vdash_s M.l_i:\tau_i} \quad \text{for all } 1 \le i \le n$$

Class
$$\frac{\begin{array}{l} C, E \vdash_s M_a:\{iv_i:\forall\ \texttt{MyType}\ <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \tau).\sigma_i, \}_{i \le k} \\ C, E \vdash_s M_b:\{m_j:\forall\ \texttt{MyType}\ <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \tau). \\ \forall\ \texttt{InstType} <: RecToMem(\sigma).\ \texttt{MyType} \to\ \texttt{InstType} \to \tau_j\}_{j \le n} \end{array}}{C, E \vdash_s \texttt{class}(M_a, M_b):\ \texttt{ClassType}(\texttt{MyType}, \sigma, \tau)}$$

where $\tau = \{m_j:\tau_j\}_{j \le n}$, $\texttt{InstType} \notin FV(\tau)$, $\sigma = \{l_i:\sigma_i\}_{i \le k}$, and
$RecToMem(\{l_i:\sigma_i\}_{i \le k}) = \{l_i:\ \texttt{ref}\ \sigma_i\}_{i \le k}$

New
$$\frac{C, E \vdash_s M:\ \texttt{ClassType}(\texttt{MyType}, \sigma, \tau)}{C, E \vdash_s \texttt{new}\ M:\ \texttt{ObjectType}(\texttt{MyType}, \tau)}$$

Msg
$$\frac{C \vdash \gamma\ <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m:\tau\}) \quad C, E \vdash_s M:\gamma}{C, E \vdash_s M \Leftarrow m:(\tau[\texttt{MyType} \mapsto \gamma])}$$

Inherits

$$\frac{\begin{array}{c} C, E \vdash_s M\colon\ \texttt{ClassType}(\texttt{MyType}, \{iv_i\colon\sigma_i\}_{i\le n}, \{m_j\colon\tau_j\}_{j\le k}) \\ C \cup \{\texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon\tau'_j\}_{j\le k+1})\} \vdash \tau'_1 <\colon \tau_1 \\ C, E \vdash_s M_1^V : \sigma_1^\forall \\ C, E \vdash_s M_{n+1}^V : \sigma_{n+1}^\forall \\ C, E \vdash_s M_1^f : \{m_j\colon\tau_j^\forall\}_{j\le k} \to \delta_1 \\ C, E \vdash_s M_{k+1}^f : \{m_j\colon\tau_j^\forall\}_{j\le k} \to \delta_{k+1} \end{array}}{\begin{array}{l} C, E \vdash_s \texttt{class inherit}\ M\ \texttt{modifying}\ iv_1, m_1; \\ \quad (\{iv_1 = M_1^V : \sigma_1^\forall, iv_{n+1} = M_{n+1}^V : \sigma_{n+1}^\forall\}, \\ \quad \{m_1 = M_1^f : \{m_j\colon\tau_j^\forall\}_{j\le k} \to \delta_1, \\ \quad\ m_{k+1} = M_{k+1}^f : \{m_j\colon\tau_j^\forall\}_{j\le k} \to \delta_{k+1}\}): \\ \quad \texttt{ClassType}(\texttt{MyType}, \{iv_i\colon\sigma_i\}_{i\le n+1}, \{m_i\colon\tau'_i\}_{i\le k+1}) \end{array}}$$

where $\sigma_i^\forall = \forall\texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon\tau'_j\}_{j\le k+1}).\sigma_i$,
for $1 \le i \le n+1$,
$\tau_l^\forall = \forall\texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon\tau_j\}_{j\le n})$.
$\forall\ \texttt{InstType} <\colon \mathit{RecToMem}(\{iv_i\colon\sigma_i\}_{i\le n}).\ \texttt{MyType}$
$\to\ \texttt{InstType} \to \tau_l$, for $1 \le l \le k$,
$\delta_p = \forall\texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m_j\colon\tau'_j\}_{j\le k+1})$.
$\forall\ \texttt{InstType} <\colon \mathit{RecToMem}(\{iv_i\colon\sigma_i\}_{i\le n+1}).\texttt{MyType}$
$\to\ \texttt{InstType} \to \tau'_p$, for $1 \le p \le k+1$,
$\tau'_j = \tau_j$ for $2 \le j \le k$, and
$\texttt{InstType} \notin FV(\tau'_1) \cup FV(\tau'_{k+1})$

Subsump

$$\frac{C \vdash_s \sigma <\colon \tau \qquad C, E \vdash_s M\colon\sigma}{C, E \vdash_s M\colon\tau}$$

Closure $_\to$

$$\frac{\hat{C}, \hat{E} \vdash_s f\colon\sigma\to\tau \ \ \mathit{and}\ \ \rho \models_s \hat{C}, \hat{E}}{C, E \vdash_s \langle f, \rho\rangle\colon\sigma_\rho \to \tau_\rho}$$

Closure $_\forall$

$$\frac{\hat{C}, \hat{E} \vdash_s f\colon\forall t.\tau \ \ \mathit{and}\ \ \rho \models_s \hat{C}, \hat{E}}{C, E \vdash_s \langle f, \rho\rangle\colon\forall t.\tau_\rho}$$

Closure $_{\forall <\#}$

$$\frac{\hat{C}, \hat{E} \vdash_s f\colon\forall t <\#\ \gamma.\tau \ \ \mathit{and}\ \ \rho \models_s \hat{C}, \hat{E}}{C, E \vdash_s \langle f, \rho\rangle\colon\forall t <\#\ \gamma_\rho.\tau_\rho}$$

Closure $_{\forall <\colon}$

$$\frac{\hat{C}, \hat{E} \vdash_s f\colon\forall t <\colon \gamma.\tau \ \ \mathit{and}\ \ \rho \models_s \hat{C}, \hat{E}}{C, E \vdash_s \langle f, \rho\rangle\colon\forall t <\colon \gamma_\rho.\tau_\rho}$$

An analogous rule holds for polymorphic (resp., bounded polymorphic) functions.

Object

$$\frac{\begin{array}{c} C, E\ \vdash_s M_a\colon(\mathit{RecToMem}(\sigma)[\texttt{MyType} \mapsto\ \ \texttt{ObjectType}(\texttt{MyType}, \tau)]) \\ C, E \vdash_s M_b\ :\ \{m_j\colon(\texttt{MyType} \to \mathit{RecToMem}(\sigma) \to \tau_j) \\ \qquad\qquad [\texttt{MyType} \mapsto\ \ \texttt{ObjectType}(\texttt{MyType}, \tau)]\}_{j\le m} \end{array}}{C, E \vdash_s \texttt{obj}(M_a, M_b)\ :\ \ \texttt{ObjectType}(\texttt{MyType}, \tau)}$$

LEMMA A.1. *Let* $C, E \vdash_s M : \tau$. *Then* $C \vdash \tau$: TYPE.
Let $C, E \vdash_s M \diamond E^*$. *Then for all* $(x:\tau) \in E^*$, $C \vdash \tau$:TYPE.

PROOF. Simple induction. □

B. SUBSTITUTIONS WITH ENVIRONMENTS

Definition B.1. Let ρ be an environment (see Definition 4.2).
Define τ_ρ by induction on the complexity of pretype expressions.

1. $t_\rho = \rho(t)$ if $t \in dom(\rho)$; otherwise $t_\rho = t$,
2. $c_\rho = c$,
3. $(\texttt{ref}\ \tau)_\rho = \ \texttt{ref}\ \tau_\rho$,
4. $(\sigma \to \tau)_\rho = \sigma_\rho \to \tau_\rho$,
5. $(\forall t.\tau)_\rho = \forall t.(\tau)_{\rho'}$, where $\rho' = \rho\backslash\{t\}$ (i.e., remove t from domain of ρ),
6. $(\forall t <\# \sigma.\tau)_\rho = \forall t <\# \sigma_{\rho'}.(\tau)_{\rho'}$, where $\rho' = \rho\backslash\{t\}$,
7. $\{m_1:\tau_1;\ldots;m_n:\tau_n\}_\rho = \{m_1:(\tau_1)_\rho;\ldots;m_n:(\tau_n)_\rho\}$,
8. $(\texttt{ClassType}(t,\sigma,\tau))_\rho = \ \texttt{ClassType}(t, \sigma_{\rho'}, \tau_{\rho'})$, where $\rho' = \rho\backslash\{t\}$,
9. $(\texttt{ObjectType}(t,\tau))_\rho = \ \texttt{ObjectType}(t, \tau_{\rho'})$, where $\rho' = \rho\backslash\{t\}$,
10. $(\forall t <: \sigma.\tau)_\rho = \forall t <: \sigma_{\rho'}.(\tau)_{\rho'}$, where $\rho' = \rho\backslash\{t\}$.

Define M_ρ by induction on the complexity of preterms.

1. $(\texttt{Program}\ x;\ Block)_\rho = \texttt{Program}\ x;\ (Block)_\rho$,
2. $(CDecls\ VDecls\ \ \texttt{begin}\ S\ \ \texttt{return}\ E\ \ \texttt{end})_\rho = CDecls_\rho\ VDecls_\rho\ \ \texttt{begin}\ S_\rho\ \ \texttt{return}E_\rho\ \ \texttt{end}$,
3. $(\texttt{const}\ CDclLst)_\rho = \ \texttt{const}\ CDclLst_\rho$,
4. $(x = M)_\rho = \ x = M_\rho$,
5. $(x = M;\ CDclLst)_\rho = \ x = M_\rho;\ CDclLst_\rho$,
6. $(\texttt{var}\ VDclLst)_\rho = \ \texttt{var}\ VDclLst_\rho$,
7. $(x:\tau)_\rho = \ x:\tau_\rho$,
8. $(x:\tau;\ VDclLst)_\rho = \ x:\tau_\rho;\ VDclLst_\rho$,
9. $x_\rho = \rho(x)$ if $x \in dom(\rho)$; otherwise $x_\rho = x$,
10. $b_\rho = b$,
11. $(\texttt{val}\ M)_\rho = \ \texttt{val}\ M_\rho$,
12. $(\texttt{function}(x:\sigma)\ Block)_\rho = \ \texttt{function}(x:\sigma_\rho)\ Block_{\rho'}$, where $\rho' = \rho\backslash\{x\}$,
13. $(\texttt{function}(t:\ \texttt{TYPE})\ Block)_\rho = \ \texttt{function}(t:\ \texttt{TYPE})\ \ Block_{\rho'}$, where $\rho' = \rho\backslash\{t\}$,
14. $(\texttt{function}(t <\# \sigma)\ Block)_\rho = \ \texttt{function}(t <\# \sigma_\rho)\ Block_{\rho'}$, where $\rho' = \rho\backslash\{t\}$,
15. $(M(M'))_\rho = (M_\rho(M'_\rho)$,
16. $(M[\tau])_\rho = M_\rho[\tau_\rho]$,
17. $\{m_1 = M_1:\tau_1, \ldots, m_n = M_n:\tau_n\}_\rho = \{m_1 = (M_1)_\rho:(\tau_1)_\rho, \ldots, m_n = (M_n)_\rho:(\tau_n)_\rho\}$,
18. $(M.l_i)_\rho = M_\rho.l_i$,

19. $(\texttt{class}(M_1, M_2))_\rho = \texttt{class}((M_1)_\rho, (M_2)_\rho)$,
20. $(\texttt{new}\ M)_\rho = \texttt{new}\ M_\rho$,
21. $(M \Leftarrow m_i)_\rho = M_\rho \Leftarrow m_i$,
22. $(\texttt{class inherit}\ M\ \texttt{modifying}\ iv_1, m_1;$
    $(\{iv_1 = M_1^V : \sigma_1, iv_n = M_{n+1}^V : \sigma_{n+1}\}, \{m_1 = M_1^f : \tau_1, m_{k+1} = M_{k+1}^f : \tau_{k+1}\}))_\rho =$
    $\texttt{class inherit}\ M_\rho\ \texttt{modifying}\ iv_1, m_1;$
    $(\{iv_1 = (M_1^V)_\rho : (\sigma_1)_\rho, iv_n = (M_{n+1}^V)_\rho : (\sigma_{n+1})_\rho\}, \{m_1 = (M_1^f)_\rho : (\tau_1)_\rho, m_{n+1}$
    $= (M_{k+1}^f)_\rho : (\tau_{k+1})_\rho\})$,
23. $\texttt{error}_\rho = \texttt{error}$,
24. $\texttt{tyerr}_\rho = \texttt{tyerr}$,
25. $(\texttt{function}(t <: \sigma)\ Block)_\rho = \texttt{function}(t <: \sigma_\rho)\ Block_{\rho'}$, where $\rho' = \rho \backslash \{t\}$,
26. $(\langle\!\langle \texttt{function}(x : \tau)\ Block, \eta \rangle\!\rangle)_\rho = \langle\!\langle \texttt{function}(x : \tau)\ Block, \eta \rangle\!\rangle$,
27. $(\langle\!\langle \texttt{function}(t : TYPE)\ Block, \eta \rangle\!\rangle)_\rho = \langle\!\langle \texttt{function}(t : TYPE)\ Block, \eta \rangle\!\rangle$,
28. $(\langle\!\langle \texttt{function}(t <\#\ \tau)\ Block, \eta \rangle\!\rangle)_\rho = \langle\!\langle \texttt{function}(t <\#\ \tau)\ Block, \eta \rangle\!\rangle$,
29. $(\langle\!\langle \texttt{function}(t <: \tau)\ Block, \eta \rangle\!\rangle)_\rho = \langle\!\langle \texttt{function}(t <: \tau)\ Block, \eta \rangle\!\rangle$,
30. $(\texttt{obj}(M_1, M_2))_\rho = \texttt{obj}((M_1)_\rho, (M_2)_\rho)$,
31. $(Loc_i)_\rho = Loc_i$,
32. $(x := M)_\rho = (x := M_\rho)$,
33. $(\texttt{if}\ M\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end})_\rho = (\ \texttt{if}\ M_\rho\ \texttt{then}\ (S_1)_\rho\ \texttt{else}\ (S_2)_\rho\ \texttt{end})$,
34. $(\texttt{while}\ M\ \texttt{do}\ S\ \texttt{end})_\rho = (\ \texttt{while}\ M_\rho\ \texttt{do}\ S_\rho\ \texttt{end})$,
35. $(S; S')_\rho = S_\rho; S'_\rho$.

Remark B.2. We use $\tau[t \mapsto \sigma]$ as a shortcut for τ_ρ, where $dom(\rho) = \{t\}$ and $\rho(t) = \sigma$.

C. THE NATURAL SEMANTICS OF POLYTOIL

The natural semantics of PolyTOIL is given by the relations

1. $\downarrow \subseteq (Term \times Env \times State) \times (IrrVal \times State)$, and
2. $\downarrow_{decl} \subseteq (Term \times Env \times State) \times (Env \times State)$.

The first relation represents a computation, starting with a triple (M, ρ, s) consisting of a term, runtime environment, and state, and ending with a pair (V, s') representing the final value of the term and the state at the end of the computation. The second relation represents processing declarations, starting with a triple (D, ρ, s) consisting of a declaration, runtime environment, and state, and ending with a pair (ρ', s') representing a new environment and the state at the end of the elaboration. (The state might change when evaluating an expression used to initialize a constant, for example.) The new environment results from adding new bindings to the existing environment.

Each relation is defined as the least relation that satisfies the following axioms and rules.

C.1 NORMAL COMPUTATIONS

$Constant_{Comp}$ $\quad (b, \rho, s) \downarrow (b, s)$ if $b \in \mathcal{B}$

$Location_{Comp}$ $\quad (L, \rho, s) \downarrow (L, s)$ if $L \in Loc$

$Variable_{Comp}$ $\quad (x, \rho, s) \downarrow (\rho(x), s)$ if $x \in \mathcal{X}$ and $x \in dom(\rho)$

$$Program_{Comp} \quad \frac{(B, \rho, s) \downarrow (\texttt{command}, s^*)}{(\texttt{program}\ p; B.\ , \rho, s) \downarrow (ok, s^*)}$$

$$ConstDecls_{Comp} \quad \frac{(CDclLst, \rho, s) \downarrow_{decl} (\rho^*, s^*)}{(\texttt{const}\ CDclLst, \rho, s) \downarrow_{decl} (\rho^*, s^*)}$$

$$ConstDcl*_{Comp} \quad \frac{(x = M, \rho, s) \downarrow_{decl} (\rho_1, s_1)\ (CDclLst, \rho_1, s_1) \downarrow_{decl} (\rho_2, s_2)}{(x = M;\ CDclLst, \rho, s) \downarrow_{decl} (\rho_2, s_2)}$$

$$ConstDcl_{Comp} \quad \frac{(M, \rho, s) \downarrow (V, s^*)}{(x = M, \rho, s) \downarrow_{decl} (\rho[x \mapsto V], s^*)}$$

$$VarDecls_{Comp} \quad \frac{(VDclLst, \rho, s) \downarrow_{decl} (\rho^*, s^*)}{(\texttt{var}\ VDclLst, \rho, s) \downarrow_{decl} (\rho^*, s^*)}$$

$$VarDcl*_{Comp} \quad \frac{(x\colon \tau, \rho, s) \downarrow_{decl} (\rho_1, s_1) \qquad (VDclLst, \rho_1, s_1) \downarrow_{decl} (\rho_2, s_2)}{(x\colon \tau;\ VDclLst, \rho, s) \downarrow_{decl} (\rho_2, s_2)}$$

$$VarDcl_{Comp} \quad \frac{(newL, s*) = (GetNewLoc\ s\ \tau_\rho\ \texttt{error})}{(x\colon \tau, \rho, s) \downarrow_{decl} (\rho[x \mapsto newL], s*)}$$

$$Block_{Comp} \quad \frac{\begin{array}{c}(CDcls, \rho, s) \downarrow_{decl} (\rho_1, s_1) \qquad (VDcls, \rho_1, s_1) \downarrow_{decl} (\rho_2, s_2) \\ (S, \rho_2, s_2) \downarrow (\texttt{command}, s_3) \qquad (M, \rho_2, s_3) \downarrow (V, s^*)\end{array}}{(CDcls\ \ VDcls\ \texttt{begin}\ S\ \texttt{return}\ M\ \texttt{end}, \rho, s) \downarrow (V, s^*)}$$

$$Assn_{Comp} \quad \frac{(x, \rho, s) \downarrow (L, s^*) \qquad (M, \rho, s^*) \downarrow (V, s^{**})}{(x := M, \rho, s) \downarrow (\texttt{command}, s^{**}[L \mapsto V])}$$

$$Cond^{true}_{Comp} \quad \frac{(M, \rho, s) \downarrow (\texttt{true}, s_1) \qquad (S_1, \rho, s_1) \downarrow (\texttt{command}, s_2)}{(\texttt{if}\ M\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}, \rho, s) \downarrow (\texttt{command}, s_2)}$$

$$Cond^{false}_{Comp} \quad \frac{(M, \rho, s) \downarrow (\texttt{false}, s_1) \quad (S_2, \rho, s_1) \downarrow (\texttt{command}, s_2)}{(\texttt{if } M \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, \rho, s) \downarrow (\texttt{command}, s_2)}$$

$$While^{true}_{Comp} \quad \frac{(M, \rho, s) \downarrow (\texttt{true}, s_1) \quad (S, \rho, s_1) \downarrow (\texttt{command}, s_2) \quad (\texttt{while } M \texttt{ do } S \texttt{ end}, \rho, s_2) \downarrow (\texttt{command}, s_3)}{(\texttt{while } M \texttt{ do } S \texttt{ end}, \rho, s) \downarrow (\texttt{command}, s_3)}$$

$$While^{false}_{Comp} \quad \frac{(M, \rho, s) \downarrow (\texttt{false}, s*)}{(\texttt{while } M \texttt{ do } S \texttt{ end}, \rho, s) \downarrow (\texttt{command}, s*)}$$

$$StmtList_{Comp} \quad \frac{(S_1, \rho, s) \downarrow (\texttt{command}, s_1) \quad (S_2, \rho, s_1) \downarrow (\texttt{command}, s_2)}{(S_1; S_2, \rho, s) \downarrow (\texttt{command}, s_2)}$$

$$Value_{Comp} \quad \frac{(M, \rho, s) \downarrow (L, s^*)}{(\texttt{val } M, \rho, s) \downarrow (s^*(L), s^*)}$$

$$Function_{Comp} \quad (\texttt{function}(x\texttt{:}\sigma)\ B, \rho, s) \downarrow (\langle \texttt{function}(x\texttt{:}\sigma)\ B, \rho \rangle, s)$$

$$PolyFunction_{Comp} \quad (\texttt{function}(t\texttt{:TYPE})\ B, \rho, s) \downarrow (\langle \texttt{function}(t\texttt{:TYPE})\ B, \rho \rangle, s)$$

$$<\#\ BdPolyFunction_{Comp} \quad (\texttt{function}(t\ <\#\ \gamma)\ B, \rho, s) \downarrow (\langle \texttt{function}(t\ <\#\ \gamma)\ B, \rho \rangle, s)$$

$$<\!:\ BdPolyFunction_{Comp} \quad (\texttt{function}(t\ <\!:\ \gamma)\ B, \rho, s) \downarrow (\langle \texttt{function}(t\ <\!:\ \gamma)\ B, \rho \rangle, s)$$

$$FuncAppl_{Comp} \quad \frac{(M_1, \rho, s) \downarrow (\langle \texttt{function}(x\texttt{:}\hat{\sigma})\ B, \eta \rangle, s_1) \quad (M_2, \rho, s_1) \downarrow (V_2, s_2) \quad (B, \eta[x \mapsto V_2], s_2) \downarrow (V, s_3)}{(M_1 M_2, \rho, s) \downarrow (V, s_3)}$$

$$PolyFuncAppl_{Comp} \quad \frac{(M, \rho, s) \downarrow (\langle \texttt{function}(t\texttt{:TYPE})\ B, \eta \rangle, s_1) \quad (B, \eta[t \mapsto \sigma_\rho], s_1) \downarrow (V, s_2)}{(M[\sigma], \rho, s) \downarrow (V, s_2)}$$

$$<\#\ BdPolyFuncAppl_{Comp} \quad \frac{(M, \rho, s) \downarrow (\langle \texttt{function}(t\ <\#\ \hat{\gamma})\ B, \eta \rangle, s_1) \quad (B, \eta[t \mapsto \sigma_\rho], s_1) \downarrow (V, s_2)}{(M[\sigma], \rho, s) \downarrow (V, s_2)}$$

$$<\!:\ BdPolyFuncAppl_{Comp} \quad \frac{(M, \rho, s) \downarrow (\langle \texttt{function}(t\ <\!:\ \hat{\gamma})\ B, \eta \rangle, s_1) \quad (B, \eta[t \mapsto \sigma_\rho], s_1) \downarrow (V, s_2)}{(M[\sigma], \rho, s) \downarrow (V, s_2)}$$

$$Record_{Comp} \quad \frac{(M_i, \rho, s_i) \downarrow (V_i, s_{i+1}) \quad \text{for all } 1 \le i \le n}{(\{l_1 = M_1\texttt{:}\tau_1, \ldots, l_n = M_n\texttt{:}\tau_n\}, \rho, s_1) \downarrow (\{l_1 = V_1\texttt{:}(\tau_1)_\rho, \ldots, l_n = V_n\texttt{:}(\tau_n)_\rho\}, s_{n+1})}$$

$$Proj_{Comp} \qquad \frac{(M, \rho, s) \downarrow (\{l_1 = V_1 : \hat{\tau}_1, \ldots, l_n = V_n : \hat{\tau}_n\}, \rho, s*)}{(M.l_i, \rho, s) \downarrow (V_i, s*) \quad \text{for all } 1 \le i \le n}$$

$$Class_{Comp} \qquad \frac{(M_a, \rho, s) \downarrow (V_a, s_1) \qquad (M_b, \rho, s_1) \downarrow (V_b, s_2)}{(\texttt{class}(M_a, M_b), \rho, s) \downarrow (\texttt{class}(V_a, V_b), s_2)}$$

$$New_{Comp} \qquad \frac{\begin{array}{c}(M, \rho, s) \downarrow (\texttt{class}(\{iv_i = \langle \texttt{function}(\texttt{MyType} <\# \hat{\gamma}_i)\ B_i, \eta_i \rangle : \sigma_i^{\forall}\}_{i \le n} \\ \{m_j = \langle \texttt{function}(\texttt{MyType} <\# \hat{\gamma}_j)\ B_j, \kappa_j \rangle : \tau_j^{\forall}\}_{j \le k}),\ s_1) \\ \vdots \\ (B_i, \eta_i[\texttt{MyType} \mapsto \ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)], s_i) \downarrow (V_i, s_{i+1}) \\ \vdots \\ (newL_i, s_{n+i+1}) = (GetNewLoc\ \ s_{n+i}\ (\sigma_i)_\rho[\texttt{MyType} \\ \mapsto\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]\ \ V_i) \\ \vdots \\ (B_j, \kappa_j[\texttt{MyType} \mapsto \ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)], s_{2n+2(j-1)+1}) \\ \downarrow (\langle \texttt{function}(\texttt{InstType} <: \hat{\sigma}_j)\ \hat{B}_j, \hat{\kappa}_j \rangle, s_{2n+2j}) \\ (\hat{B}_j, \hat{\kappa}_j[\ \texttt{InstType} \mapsto RecToMem(\sigma_\rho)], s_{2n+2j}) \\ \downarrow (\langle \texttt{function}(\texttt{self} : \varsigma_j)\ \breve{B}_j, \breve{\kappa}_j \rangle, s_{2n+2j+1}) \\ \vdots \end{array}}{(\texttt{new}\ M, \rho, s) \downarrow (ob, s_{2n+2k+1})}$$

where
$\sigma = \{iv_i : \sigma_i\}_{i \le n}$, $\tau = \{m_j : \tau_j\}_{j \le k}$,
$C, E \vdash_s M$: $\texttt{ClassType}(\texttt{MyType}, \sigma, \tau)$,
$\emptyset \vdash \sigma_i^{\forall} <: \forall \texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).(\sigma_i)_\rho$, for $i \le n$,
$\emptyset \vdash \tau_j^{\forall} <: \forall \texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).\forall\ \texttt{InstType} <: RecToMem(\sigma_\rho).$
$\texttt{MyType} \to\ \ \texttt{InstType} \to (\tau_j)_\rho$, for $j \le k$,
and
$ob = \texttt{obj}(\{iv_i = newL_i : \ \texttt{ref}\ (\sigma_i)_\rho[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]\}_{i \le n}$,
$\{m_j = \langle function(\texttt{self} : \varsigma^j)\ \breve{B}^j, \breve{\kappa}^j \rangle : (\texttt{MyType} \to RecToMem(\sigma_\rho) \to (\tau_j)_\rho)$
$[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]\}_{j \le k})$

$$Msg_{Comp} \qquad \frac{\begin{array}{c}(M, \rho, s) \downarrow (ob, s_1) \\ (B_b, \eta_b[\texttt{self} \mapsto ob], s_1) \downarrow (\langle \texttt{function}(\texttt{inst} : \hat{\delta}_{\texttt{InstType}})\ B_c, \eta_c \rangle, s_2) \\ (B_c, \eta_c[\texttt{inst} \mapsto V_a], s_2) \downarrow (\langle \texttt{function}(x : \hat{\alpha})\ B, \kappa \rangle, s_3)\end{array}}{(M \Leftarrow m_j, \rho, s) \downarrow (\langle \texttt{function}(x : \hat{\alpha})\ B, \kappa \rangle,\ s_3)}$$

where
$ob = \texttt{obj}(V_a, \{\ldots, m_j = \langle \texttt{function}(\texttt{self} : \hat{\delta}_{\texttt{MyType}})\ B_b, \eta_b \rangle : \hat{\tau}_j, \ldots\})$

Inherits$_{Comp}$

$$\frac{\begin{array}{c}(M,\rho,s)\downarrow\big(\texttt{class}\big(\{\ldots, iv_i = V_i^V : \sigma_i^\forall, \ldots\}, \{\ldots, m_j = V_j^f : \tau_j^\forall, \ldots\}\big), s_1\big)\\ (M_1^V,\rho,s_1)\downarrow(\hat{V}_1^V, s_2)\\ (M_{n+1}^V,\rho,s_2)\downarrow(V_{n+1}^V, s_3)\\ (M_1^f,\rho,s_3)\downarrow(\langle\texttt{function(super:}\,\hat{\gamma}_1)\ B_1,\kappa_1\rangle\ ,\ s_4)\\ (M_{k+1}^f,\rho,s_4)\downarrow(\langle\texttt{function(super:}\,\hat{\gamma}_{k+1})\ B_{k+1},\kappa_{k+1}\rangle\ ,\ s_5)\\ (B_1,\kappa_1[\texttt{super}\mapsto\{\ldots, m_j = V_j^f:\tau_j^\forall,\ldots\}], s_5)\downarrow(\hat{V}_1^f, s_6)\\ (B_{k+1},\kappa_{k+1}[\texttt{super}\mapsto\{\ldots, m_j = V_j^f:\tau_j^\forall,\ldots\}], s_6)\downarrow(V_{k+1}^f, s_7)\end{array}}{\begin{array}{l}(\texttt{class inherit}\ M\ \texttt{modifying}\ iv_1, m_1;\\ \quad(\{iv_1 = M_1^V:\hat{\sigma}_1^\forall, iv_{n+1} = M_{n+1}^V:\hat{\sigma}_{n+1}^\forall\},\\ \quad\{m_1 = M_1^f:\{m_1:\tau_1^\forall,\ldots,m_k:\tau_k^\forall\}\to\delta_1,\\ \quad\ m_{k+1} = M_{k+1}^f:\{m_1:\tau_1^\forall,\ldots,m_k:\tau_k^\forall\}\to\delta_{k+1}\}),\rho,s)\downarrow\\ (\texttt{class}(\{iv_1 = \hat{V}_1^V:(\hat{\sigma}_1^\forall)_\rho,\ldots,iv_i = \breve{V}_i^V:(\hat{\sigma}_i^\forall)_\rho,\ldots,iv_{n+1} = V_{n+1}^V:(\hat{\sigma}_{n+1}^\forall)_\rho\},\\ \quad\{m_1 = \hat{V}_1^f:(\hat{\tau}_1^\forall)_\rho,\ldots,m_j = \breve{V}_j^f:(\hat{\tau}_j^\forall)_\rho,\ldots,m_{k+1} = V_{k+1}^f:(\hat{\tau}_{k+1}^\forall)_\rho\}), s_7)\end{array}}$$

where

$\emptyset\vdash\sigma_i^\forall <: \forall\texttt{MyType} <\#\ \ \texttt{ObjectType(MyType,}\ \tau_\rho).(\sigma_i)_\rho,\ \ \text{for } 1\le i\le n$

$\emptyset\vdash\tau_j^\forall <: \forall\texttt{MyType} <\#\ \ \texttt{ObjectType(MyType,}\ \tau_\rho).\forall\ \texttt{InstType} <: \mathit{RecToMem}(\sigma_\rho).$
$\texttt{MyType}\to\ \texttt{InstType}\to(\tau_j)_\rho,\ \ \text{for } 1\le j\le k$

$\hat{\sigma}_i^\forall = \forall\texttt{MyType} <\#\ \ \texttt{ObjectType(MyType,}\ \{m_1:\hat{\tau}_1,\ldots,m_{k+1}:\tau_{k+1}\}).\sigma_i,$
$\text{for } 1\le i\le n+1$

$\hat{\tau}_j^\forall = \forall\texttt{MyType} <\#\ \ \texttt{ObjectType(MyType,}\ \{m_1:\hat{\tau}_1,\ldots,m_{k+1}:\tau_{k+1}\}).$
$\forall\ \texttt{InstType} <: \mathit{RecToMem}(\{iv_1:\sigma_1,\ldots,iv_{n+1}:\sigma_{n+1}\}).$
$\texttt{MyType}\to\ \texttt{InstType}\to\tau_j,\ \text{for } 1\le j\le k+1$

$\delta_p = \forall\texttt{MyType} <\#\ \ \texttt{ObjectType(MyType,}\ m_1:\hat{\tau}_1,\ldots,m_{k+1}:\tau_{k+1}).$
$\forall\ \texttt{InstType} <: \mathit{RecToMem}(\{iv_1:\sigma_1,\ldots,iv_{n+1}:\sigma_{n+1}\}).$
$\texttt{MyType}\to\ \texttt{InstType}\to\tilde{\tau}_p,\ \text{for } 1\le p\le k+1$

$\tilde{\tau}_1 = \hat{\tau}_1\ \text{ and }\ \tilde{\tau}_{k+1} = \tau_{k+1}$

$\breve{V}_i^V = \langle\texttt{function(MyType <\# ObjectType(MyType,}\ \{m_1:(\hat{\tau}_1)_\rho,\ldots,m_{k+1}:(\tau_{k+1})_\rho\})).$
$V_i^V[\texttt{MyType}],\ \eta_\emptyset\rangle,\ \text{for } 2\le i\le n$

$\breve{V}_j^f = \langle\texttt{function(MyType <\# ObjectType(MyType,}\ \{m_1:(\hat{\tau}_1)_\rho,\ldots,m_{k+1}:(\tau_{k+1})_\rho\})).$
$\texttt{function(InstType} <: \mathit{RecToMem}(\{iv_1:(\sigma_1)_\rho,\ldots,iv_{n+1}:(\sigma_{n+1})_\rho\}))$
$V_j^f[\texttt{MyType}][\texttt{InstType}],\ \eta_\emptyset\rangle,\ \text{for } 2\le j\le k$

and $\eta_\emptyset$ is the empty environment

Closure$_{Comp}$ $\qquad (\langle M,\eta\rangle,\rho,s)\downarrow(\langle M,\eta\rangle,s)$

Object$_{Comp}$ $\qquad \dfrac{(M_a,\rho,s)\downarrow(V_a,s_1)\quad(M_b,\rho,s_1)\downarrow(V_b,s_2)}{(\texttt{obj}(M_a,M_b),\rho,s)\downarrow(\texttt{obj}(V_a,V_b),s_2)}$

C.2 Computations Resulting in Nontype Errors

These errors may occur in type-correct programs and thus are not prevented by the type system. The only significant error that can be generated during a computation is that resulting from sending a message to *nil*. Other rules simply propagate errors through computations.

Of course one would probably choose to have primitive operations (division by zero, arithmetic overflow, etc.) also generate errors. We have not specified such primitive operations here.

$$Error_{error} \qquad (\texttt{error}, \rho, s) \downarrow (\texttt{error}, s)$$

$$Msg_{nil} \qquad \frac{(M, \rho, s) \downarrow (\texttt{nil}, s_1)}{(M \Leftarrow m_j, \rho, s) \downarrow (\texttt{error},\ s_1)}$$

$$Msg_{error} \qquad \frac{\begin{array}{c}(M, \rho, s) \downarrow (\texttt{obj}(V_a, \{\ldots, m_j = \texttt{error}\colon \hat{\tau}_j, \ldots\}), s_1) \\ \text{or} \quad (M, \rho, s) \downarrow (\texttt{obj}(\texttt{error}, \{\ldots\}), s_1)\end{array}}{(M \Leftarrow m_j, \rho, s) \downarrow (\texttt{error},\ s_1)}$$

$$Error_{propagating} \qquad \frac{\ldots,\ (M_i, \rho_i, s_i) \downarrow (\texttt{error}, \hat{s}_i)\ , \ldots}{(M, \rho, s) \downarrow (\texttt{error}, \hat{s}_i)}$$

Here for each term construction

$$\frac{C, E \vdash_s M_1\colon \tau_1\ , \ldots,\ C, E \vdash_s M_n\colon \tau_n}{C, E \vdash_s M\colon \tau}$$

we add extra semantics rules which indicate that if any of the subterms M_i needed in the computation evaluate to error, then the term M evaluates to error.

C.3 Computations Resulting in Type Errors

These rules specify computation errors that should never occur in type-correct programs. They are only included for use in the subject reduction theorem and its corollaries, where it is shown that they can never apply if we start off with a program that is assigned a type by our type-checking rules.

Accordingly, rather than listing several pages of rules that will never apply we simply provide representative samples. (A complete listing of these rules is available from the authors.)

$$Error_{tyerr} \qquad (\texttt{tyerr}, \rho, s) \downarrow (\texttt{tyerr}, s)$$

$$Variable_{Env} \qquad (x, \rho, s) \downarrow (\texttt{tyerr}, s) \quad \text{if } x \in \mathcal{X} \text{ and } x \notin dom(\rho)$$

$$Assn_{Location} \qquad \frac{(x, \rho, s) \downarrow (V, s^*) \qquad V \notin (Loc \cap dom(s^*)) \cup \{\texttt{error}\}}{(x\colon = M, \rho, s) \downarrow (\texttt{tyerr}, s^*)}$$

$$Cond_{Bool} \qquad \frac{(M, \rho, s) \downarrow (V, s_1) \qquad V \notin \{\texttt{true}, \texttt{false}, \texttt{error}\}}{(\texttt{if}\ M\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}, \rho, s) \downarrow (\texttt{tyerr}, s_1)}$$

$$Cond_{Command}\ \frac{(M,\rho,s)\downarrow(V,s_1)\quad (S_i,\rho,s_1)\downarrow(V_i,s_2)\quad V_i\notin\{\texttt{command},\texttt{error}\}}{(\texttt{if}\ M\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end},\rho,s)\downarrow(\texttt{tyerr},s_2)}$$

$$Value_{Location}\ \frac{(M,\rho,s)\downarrow(V,s^*)\quad V\notin(Loc\cap dom(s^*))\cup\{\texttt{error}\}}{(\texttt{val}\ M,\rho,s)\downarrow(\texttt{tyerr},s^*)}$$

$$FuncAppl_{Abstr}\ \frac{(M_1,\rho,s)\downarrow(V,s_1)\quad V\notin\{\langle\texttt{function}(x{:}\sigma)\ B,\eta\rangle,\texttt{error}\}}{(M_1M_2,\rho,s)\downarrow(\texttt{tyerr},s_1)}$$

$$FuncAppl_{Arg}\ \frac{(M_1,\rho,s)\downarrow(\langle\texttt{function}(x{:}\hat{\sigma})\ B,\eta\rangle,s_1)\ \ (M_2,\rho,s_1)\downarrow(\texttt{tyerr},s_2)}{(M_1M_2,\rho,s)\downarrow(\texttt{tyerr},s_2)}$$

$$PolymorphicFuncAppl_{Abstr}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(V,s_1)\quad V\notin\{\langle\texttt{function}(t{:}\,\texttt{TYPE})\ B,\eta\rangle,\\ \langle\texttt{function}(t\ <\#\ \tau)\ B,\eta\rangle\\ \langle\texttt{function}(t<:\tau)\ B,\eta\rangle,\ \texttt{error}\}\end{array}}{(M[\sigma],\rho,s)\downarrow(\texttt{tyerr},s_1)}$$

$$New_{Class}\ \frac{(M,\rho,s)\downarrow(V,\ s_1)\quad V\notin\{\texttt{class}(V_1,V_2),\ \texttt{error}\}}{(\texttt{new}\ M,\rho,s)\downarrow(\texttt{tyerr},s_1)}$$

$$New_{<\#BdPolyFunc}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(\texttt{class}(\{\ldots,iv_i=V_i{:}\sigma_i^{\forall},\ldots\},\{\ldots,m_j=V_j{:}\tau_j^{\forall},\ldots\}),s_1)\\ V_i\text{ or }V_j\ \notin\{\langle\texttt{function}(t\ <\#\ \tau)\ B,\eta\rangle,\ \texttt{error}\}\end{array}}{(\texttt{new}\ M,\rho,s)\downarrow(\texttt{tyerr},s_1)}$$

plus similar rules for later stages of the computation of `new M`

$$Msg_{Obj}\ \frac{(M,\rho,s)\downarrow(V,s_1)\quad V\notin\{\texttt{obj}(V_1,V_2),\ \texttt{nil},\ \texttt{error}\}}{(M\Leftarrow m_j,\rho,s)\downarrow(\texttt{tyerr},\ s_1)}$$

$$Msg_{Not\_Understood}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(\texttt{obj}(V_a,\{\ldots,m_{k_i}\\ =\langle\texttt{function}(\texttt{self}{:}\hat{\delta}_{MT})\ B_b,\eta_b\rangle{:}\hat{\tau}_{k_i},\ldots\}),s_1)\forall i{:}\,m_{k_i}\neq m_j\end{array}}{(M\Leftarrow m_j,\rho,s)\downarrow(\texttt{tyerr},s_1)}$$

$$Msg_{InstVar}\ \frac{(M,\rho,s)\downarrow\texttt{obj}(\texttt{tyerr},\{\ldots,m_j=\langle\texttt{function}(\texttt{self}{:}\hat{\delta}_{MT})B_b,\eta_b\rangle{:}\hat{\tau}_j,\ldots\})}{(M\Leftarrow m_j,\rho,s)\downarrow(\texttt{tyerr},\ s_1)}$$

$$Msg_{Closure_1}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(\texttt{obj}(V_a,\{\ldots,m_j=V_j,\ldots\}),s_1)\\ V_j\notin\{\langle\texttt{function}(x{:}\sigma)\ B,\eta\rangle,\texttt{error}\}\end{array}}{(M\Leftarrow m_j,\rho,s)\downarrow(\texttt{tyerr},\ s_1)}$$

$$Msg_{Closure_2}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(ob,s_1)\\ (B_b,\eta_b[\texttt{self}\mapsto ob],s_1)\downarrow(V_b,s_2)\\ V_b\notin\{\langle\texttt{function}(x{:}\sigma)\ B,\eta\rangle,\texttt{error}\}\end{array}}{(M\Leftarrow m_j,\rho,s)\downarrow(\texttt{tyerr},\ s_2)}$$

where $ob=\texttt{obj}(V_a,\{\ldots,m_j=\langle\texttt{function}(\texttt{self}{:}\hat{\delta}\ \texttt{MyType})\ B_b,\eta_b\rangle{:}\hat{\tau}_j,\ldots\})$

$$Msg_{Closure_3}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(ob,s_1)\\ (B_b,\eta_b[\texttt{self}\mapsto ob],s_1)\downarrow(\langle\texttt{function(inst:}\hat{\delta}_{\texttt{InstType}})\ B_c,\eta_c\rangle,s_2)\\ (B_c,\eta_c[\texttt{inst}\mapsto V_a],s_2)\downarrow(V_c,s_3)\\ V_c\notin\{\langle\texttt{function}(x{:}\sigma)\ B,\eta\rangle,\texttt{error}\}\end{array}}{(M\Leftarrow m_j,\rho,s)\downarrow(\texttt{tyerr},\ s_3)}$$

where $ob=\texttt{obj}(V_a,\{\ldots,m_j=\langle\texttt{function(self:}\hat{\delta}\ \texttt{MyType})\ B_b,\eta_b\rangle{:}\hat{\tau}_j,\ldots\})$

$$Inherits_{Class}\ \frac{(M,\rho,s)\downarrow(V,\ s_1)\quad V\notin\{\texttt{class}(V_1,V_2),\ \texttt{error}\}}{\begin{array}{l}(\texttt{class inherit}\ M\ \texttt{modifying}\ iv_1,m_1;\\ \quad(\{iv_1=M_1^V{:}\hat{\sigma}_1^\forall,iv_{n+1}=M_{n+1}^V{:}\hat{\sigma}_{n+1}^\forall\},\\ \quad\{m_1=M_1^f{:}\{m_1{:}\tau_1^\forall,\ldots,m_k{:}\tau_k^\forall\}\to\delta_1,\\ \quad\ m_{k+1}=M_{k+1}^f{:}\{m_1{:}\tau_1^\forall,\ldots,m_k{:}\tau_k^\forall\}\to\delta_{k+1}\}),\rho,s)\downarrow\\ (\texttt{tyerr},s_1)\end{array}}$$

$$Inherits_{Closure_1}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(\texttt{class}(\{\ldots,iv_i{=}V_i^V{:}\sigma_i^\forall,\ldots\},\{\ldots,m_j{=}V_j^f{:}\tau_j^\forall,\ldots\}),s_1)\\ (M_1^V,\rho,s_1)\downarrow(\hat{V}_1^V,\ ,\ s_2)\\ (M_{n+1}^V,\rho,s_1)\downarrow(V_{n+1}^V,,\ s_3)\\ (M_1^f,\rho,s_3)\downarrow(V_1^f,\ s_4)\quad V_1^f\notin\{\langle\texttt{function}(x{:}\sigma)\ B,\eta\rangle,\texttt{error}\}\end{array}}{\begin{array}{l}(\texttt{class inherit}\ M\ \texttt{modifying}\ iv_1,m_1;\\ \quad(\{iv_1=M_1^V{:}\hat{\sigma}_1^\forall,iv_{n+1}=M_{n+1}^V{:}\hat{\sigma}_{n+1}^\forall\},\\ \quad\{m_1=M_1^f{:}\{m_1{:}\tau_1^\forall,\ldots,m_k{:}\tau_k^\forall\}\to\delta_1,\\ \quad\ m_{k+1}=M_{k+1}^f{:}\{m_1{:}\tau_1^\forall,\ldots,m_k{:}\tau_k^\forall\}\to\delta_{k+1}\}),\rho,s)\downarrow\\ (\texttt{tyerr},s_4)\end{array}}$$

$$Inherits_{Closure_2}\ \frac{\begin{array}{c}(M,\rho,s)\downarrow(\texttt{class}(\{\ldots,iv_i{=}V_i^V{:}\sigma_i^\forall,\ldots\},\{\ldots,m_j{=}V_j^f{:}\tau_j^\forall,\ldots\}),s_1)\\ (M_1^V,\rho,s_1)\downarrow(\hat{V}_1^V,\ ,\ s_2)\\ (M_{n+1}^V,\rho,s_1)\downarrow(V_{n+1}^V,,\ s_3)\\ (M_1^f,\rho,s_3)\downarrow(\langle\texttt{function(super:}\hat{\gamma}^1)\ B^1,\kappa^1\rangle\ ,\ s_4)\\ (M_{k+1}^f,\rho,s_4)\downarrow(V_{k+1}^f,\ s_5)\quad V_{k+1}^f\notin\{\langle\texttt{function}(x{:}\sigma)\ B,\eta\rangle,\texttt{error}\}\end{array}}{\begin{array}{l}(\texttt{class inherit}\ M\ \texttt{modifying}\ iv_1,m_1;\\ \quad(\{iv_1=M_1^V{:}\hat{\sigma}_1^\forall,iv_{n+1}=M_{n+1}^V{:}\hat{\sigma}_{n+1}^\forall\},\\ \quad\{m_1=M_1^f{:}\{m_1{:}\tau_1^\forall,\ldots,m_k{:}\tau_k^\forall\}\to\delta_1,\\ \quad\ m_{k+1}=M_{k+1}^f{:}\{m_1{:}\tau_1^\forall,\ldots,m_k{:}\tau_k^\forall\}\to\delta_{k+1}\}),\rho,s)\downarrow\\ (\texttt{tyerr},s_5)\end{array}}$$

D. SUBJECT REDUCTION FOR POLYTOIL

In this section we present the proof of the type safety of PolyTOIL. In particular we first show that types are preserved under computation. It will follow from the proof of this theorem that type-correct terms never evaluate to `tyerr`. Another way of saying this is that computations on type-correct terms will never get "stuck." (Recall that the rules resulting in `tyerr` were generated by writing down rules for all cases not handled by the other computation rules.) Thus the results in this section imply that the language PolyTOIL is statically type safe.

We begin by characterizing the set of irreducible values of each type of the language.

Definition D.1. Let $\emptyset \vdash \tau$: TYPE. Define $[\tau]_s = \{V \in IrrVal \mid \emptyset, \emptyset \vdash_s V : \tau\}$.

LEMMA D.2. *Let* $\emptyset \vdash \tau$: TYPE. *The set of irreducible elements of each type is characterized by the following:*

1. $[c]_s = \Im[c] \cup \{\texttt{error}\}$
2. $[Null]_s = \{\texttt{nil}, \texttt{error}\}$
3. $[Object]_s = \bigcup_{\emptyset \vdash \texttt{ObjectType(MyType,}\tau\texttt{)}:\texttt{TYPE}} [\texttt{ObjectType(MyType,}\ \tau\texttt{)}]_s$
4. $[\sigma \to \tau]_s =$
   $\{\langle \texttt{function}(x:\hat{\sigma})\ B, \eta\rangle \mid \exists\, \hat{C}, \hat{E}, \hat{\sigma}, \hat{\tau}\texttt{:}\ \hat{C}, \hat{E} \cup \{x:\hat{\sigma}\} \vdash_s B:\hat{\tau},$
   $\emptyset \vdash \hat{\tau}_\eta <: \tau,\ \emptyset \vdash \sigma <: \hat{\sigma}_\eta,\ \eta \models_s \hat{C}, \hat{E}\}\ \cup\ \{\texttt{error}\}$
5. $[\forall t.\tau]_s =$
   $\{\langle \texttt{function}(t\texttt{:}\ \texttt{TYPE})\ B, \eta\rangle \mid \exists\, \hat{C}, \hat{E}, \hat{\tau}\ \texttt{:}\ \hat{C} \cup \{t\texttt{:}\ \texttt{TYPE}\}, \hat{E} \vdash_s B:\hat{\tau},$
   $\{u\texttt{:}\ \texttt{TYPE}\} \vdash \hat{\tau}_\eta[t \mapsto u] <: \tau[t \mapsto u],\ \eta \models_s \hat{C}, \hat{E}\} \cup \{\texttt{error}\}$
6. $[\forall t <\# \gamma.\tau]_s =$
   $\{\langle \texttt{function}(t <\# \hat{\gamma})\ B, \eta\rangle \mid \exists\, \hat{C}, \hat{E}, \hat{\gamma}, \hat{\tau}\ \texttt{:}\ \hat{C} \cup \{t <\# \hat{\gamma}\}, \hat{E} \vdash_s B:\hat{\tau},$
   $\{u <\# \hat{\gamma}\} \vdash \hat{\tau}_\eta[t \mapsto u] <: \tau[t \mapsto u],\ \hat{\gamma}_\eta = \gamma,\ \eta \models_s \hat{C}, \hat{E}\} \cup \{\texttt{error}\}$
7. $[\forall t <: \gamma.\tau]_s =$
   $\{\langle \texttt{function}(t <: \hat{\gamma})\ B, \eta\rangle \mid \exists\, \hat{C}, \hat{E}, \hat{\gamma}, \hat{\tau}\ \texttt{:}\ \hat{C} \cup \{t <: \hat{\gamma}\}, \hat{E} \vdash_s B:\hat{\tau},$
   $\{u <: \hat{\gamma}\} \vdash \hat{\tau}_\eta[t \mapsto u] <: \tau[t \mapsto u],\ \hat{\gamma}_\eta = \gamma,\ \eta \models_s \hat{C}, \hat{E}\} \cup \{\texttt{error}\}$
8. $[\{m_1:\tau_1, \ldots, m_n:\tau_n\}]_s =$
   $\{\ \{m_1 = V_1:\hat{\tau}_1, \ldots, m_n = V_n:\hat{\tau}_n, \ldots, m_{n+m} = V_{n+m}:\hat{\tau}_{n+m}\} \mid \text{for all } 1 \le i$
   $\le n+m,\ V_i \in [\hat{\tau}_i]_s \text{ and for all } 1 \le i \le n,\ \emptyset \vdash \hat{\tau}_i <: \tau_i\ \} \cup \{\texttt{error}\}$
9. $[\texttt{ref}\ \tau]_s = (Loc_\tau \cap dom(s)) \cup \{\texttt{error}\}$
10. $[\texttt{ObjectType(MyType,}\ \tau\texttt{)}]_s = \bigcup_{\{\ \texttt{MyType:}\ \texttt{TYPE}\} \vdash \sigma:\ \texttt{TYPE}}$
    $\bigcup_{\emptyset \vdash \texttt{ObjectType(MyType,}\sigma^*\texttt{)} <:\ \texttt{ObjectType(MyType,}\tau\texttt{)}} \{\texttt{obj}(M_a, M_b) \mid M_a$
    $\in [RecToMem(\sigma)[\texttt{MyType} \mapsto \texttt{ObjectType(MyType,}\ \sigma^*\texttt{)}]]_s, M_b$
    $\in [\{\ldots, m_j:(\texttt{MyType} \to RecToMem(\sigma) \to \tau_j^*)$
    $[\texttt{MyType} \mapsto \texttt{ObjectType(MyType,}\ \sigma^*\texttt{)}], \ldots\}]_s\} \cup \{\texttt{error}\}$
11. $[\texttt{ClassType(MyType,}\ \sigma, \tau\texttt{)}]_s = \{\texttt{class}(M_a, M_b) \mid M_a \in [\{\ldots, iv_i:\forall \texttt{MyType}$
    $<\# \texttt{ObjectType(MyType,}\ \tau\texttt{)}.\sigma_i, \ldots\}]_s, M_b \in [\{\ldots, m_j:\forall \texttt{MyType}$
    $<\# \texttt{ObjectType(MyType,}\ \tau\texttt{)}.\forall \texttt{InstType} <: RecToMem(\sigma).$
    $\texttt{MyType} \to \texttt{InstType} \to \tau_j, \ldots\}]_s\} \cup \{\texttt{error}\}$

PROOF. The proof is straightforward. We show here only two cases.
$[\sigma \to \tau]_s$: $\subseteq$: Let $V \in IrrVal$ such that $\emptyset, \emptyset \vdash_s V : \sigma \to \tau$. Then examination of the typing rules shows that V must be either error or a closure. The former case is trivial, so let us assume $V = \langle \texttt{function}(x:\hat{\sigma})\ B, \eta\rangle$. The result follows from examination of the typing derivation for $\langle \texttt{function}(x:\hat{\sigma})\ B, \eta\rangle$.

$$\dfrac{\dfrac{\dfrac{\hat{C}, \hat{E} \cup \{x{:}\hat{\sigma}\} \vdash_s B{:}\hat{\tau}}{\hat{C}, \hat{E} \vdash_s \texttt{function}(x{:}\hat{\sigma})\ B{:}\hat{\sigma} \to \hat{\tau}} \quad \emptyset \vdash \hat{\tau} <: \tilde{\tau},\ \emptyset \vdash \tilde{\sigma} <: \hat{\sigma}}{\hat{C}, \hat{E} \vdash_s \texttt{function}(x{:}\hat{\sigma})\ B{:}\tilde{\sigma} \to \tilde{\tau}} \quad \eta \models_s \hat{C}, \hat{E}}{\emptyset, \emptyset \vdash \langle \texttt{function}(x{:}\hat{\sigma})\ B, \eta\rangle{:}\tilde{\sigma}_\eta \to \tilde{\tau}_\eta}$$

and

$$\dfrac{\emptyset, \emptyset \vdash \langle \texttt{function}(x{:}\hat{\sigma})\ B, \eta\rangle{:}\tilde{\sigma}_\eta \to \tilde{\tau}_\eta \quad \emptyset \vdash \hat{\tau}_\eta <: \tau,\ \emptyset \vdash \sigma <: \hat{\sigma}_\eta}{\emptyset, \emptyset \vdash \langle \texttt{function}(x{:}\hat{\sigma})\ B, \eta\rangle{:}\sigma \to \tau}$$

$\supseteq$: Follows again from examination of the above derivation.

$[\texttt{ObjectType}(\texttt{MyType}, \tau)]_s$. Let $V \in \mathit{IrrVal}$ such that $\emptyset, \emptyset \vdash_s V{:}\texttt{ObjectType}(\texttt{MyType}, \tau)$. By inspection of the typing (and subtyping) rules, V must be `error`, `nil`, or an object with a derivation as below.

$$\dfrac{\begin{array}{l}\emptyset, \emptyset \vdash_s M_a{:}(\mathit{RecToMem}(\sigma)[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau^*)]) \\ \emptyset, \emptyset \vdash_s M_b : \{\ldots, m_j{:}(\texttt{MyType} \to \mathit{RecToMem}(\sigma) \to \tau_j^*) \\ \qquad\qquad [\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau^*)], \ldots\}\end{array}}{\emptyset, \emptyset \vdash_s \texttt{obj}(M_a, M_b) : \texttt{ObjectType}(\texttt{MyType}, \tau^*)}$$

and

$$\dfrac{\begin{array}{c}\emptyset, \emptyset \vdash_s \texttt{obj}(M_a, M_b){:}\texttt{ObjectType}(\texttt{MyType}, \tau^*) \\ \emptyset \vdash \texttt{ObjectType}(\texttt{MyType}, \tau^*) <: \texttt{ObjectType}(\texttt{MyType}, \tau)\end{array}}{\emptyset, \emptyset \vdash_s \texttt{obj}(M_a, M_b){:}\texttt{ObjectType}(\texttt{MyType}, \tau)}$$

The result follows. □

Having characterized the semantics of types, it is now easy to verify that our typing rules are preserved by the semantics.

LEMMA D.3. *If* $\emptyset \vdash \tau <: \tau^*$ *then* $[\tau]_s \subseteq [\tau^*]_s$.

PROOF. By definition, $V \in [\tau]_s$ if and only if $V \in \mathit{IrrVal}$ and $\emptyset, \emptyset \vdash V{:}\tau$. But this implies that $V \in \mathit{IrrVal}$ and $\emptyset, \emptyset \vdash V{:}\tau^*$ (using subsumption), and hence $V \in [\tau^*]_s$. □

LEMMA D.4. *Let* $dom(s) \subseteq dom(s^*)$. *Then for any* $\emptyset \vdash \tau$: TYPE, $[\tau]_s \subseteq [\tau]_{s^*}$

PROOF. Simple induction. □

The following is the main theorem of the section, stating that the types of terms are preserved by the computation rules. As noted earlier, if E evaluates to V then the "minimal" type of V may be a subtype of the "minimal" type of E.

THEOREM D.5 (SUBJECT-REDUCTION (SAME AS THEOREM 5.1). Let $\rho \models_s C, E$ and $\models s$.

1. If $C, E \vdash_s M \diamond E^*$ and $(M, \rho, s) \downarrow_{decl} (\rho^*, s^*)$ then $\rho^* \models_{s^*} C, E^*$, $\models s^*$ and $dom(s) \subseteq dom(s^*)$.
2. If $C, E \vdash_s M{:}\tau$ and $(M, \rho, s) \downarrow (V, s^*)$ then $V \in [\tau_\rho]_{s^*}$, $dom(s) \subseteq dom(s^*)$ and $\models s^*$.

PROOF. The proof is by induction on the depth of computation $(M, \rho, s)\downarrow(V, s^*)$, where the base cases are : $Constant_{Comp}$, $Location_{Comp}$, $Variable_{Comp}$, $Function_{Comp}$, $PolyFunction_{Comp}$, $<: BdPolyFunction_{Comp}$, $<\# BdPolyFunction_{Comp}$, $Closure_{Comp}$, $Error_{error}$, $Error_{tyerr}$, and $Variable_{Env}$.

In most cases $\models s^*$ and $dom(s) \subseteq dom(s^*)$ follow immediately from the induction hypothesis, so that generally we do not bother to state the fact explicitly.

All type derivations are of the form

$$\dfrac{\dfrac{\dfrac{\vdots}{C, E \vdash M : \tau_1 \quad C \vdash \tau_1 <: \tau_2}}{C, E \vdash M : \tau_2 \quad C \vdash \tau_2 <: \tau_3}}{\dfrac{\vdots}{C, E \vdash M : \tau_n \quad C \vdash \tau_n <: \tau}}{C, E \vdash M : \tau}$$

where the last n steps involve only the application of subsumption. Suppose $(M, \rho, s) \downarrow (V, s^*)$, and we know that $V \in [(\tau_1)_\rho]_s$. By Lemma 4.6, $\emptyset \vdash (\tau_1)_\rho <: (\tau)_\rho$, and therefore by Lemma D.3, $V \in [\tau_\rho]_s$. To simplify the proof, we use this fact implicitly and presume that the last step of the typing of M is not an application of subsumption. In the following (unless indicated otherwise) for each computation rule (e.g., $Value_{Comp}$) the typing judgment is obtained by applying the corresponding typing rule (e.g., Value) in the last step of type derivation.

We leave out many of the simplest cases in the following proof. Cases omitted include $Constant_{Comp}$, $Program_{Comp}$, $Cond^{true}_{Comp}$ and $Cond^{false}_{Comp}$, $While^{true}_{Comp}$ and $While^{false}_{Comp}$, $StmtList_{Comp}$, $Record_{Comp}$, and $Proj_{Comp}$.

$Location_{Comp}$. The claim holds as for all closed types τ, $\tau_\rho = \tau$.

$Variable_{Comp}$. The claim holds as $x \in dom(E) \subseteq dom(\rho)$ and, therefore, $\rho(x) \in [\tau_\rho]_s$ (by the definition of $\rho \models_s C, E$).

$ConstDecls_{Comp}$. Follows trivially by the induction hypothesis on ConstDcl and ConstDcl*.

$ConstDcl^*_{Comp}$. By the induction hypothesis on the first hypothesis of the rule,

$$\rho_1 \models_{s_1} C, E_1,\ dom(s) \subseteq dom(s_1) \text{ and } \models s_1,$$

and induction on the second hypothesis gives

$$\rho_2 \models_{s_2} C, E_2,\ dom(s_1) \subseteq dom(s_2) \text{ and } \models s_2.$$

The conclusion follows immediately.

$ConstDcl_{Comp}$. By induction hypothesis $V \in [\tau_\rho]_{s^*}$, $dom(s) \subseteq dom(s^*)$ and $\models s^*$. It follows that $\rho[x \mapsto V](x) \in [\tau_\rho]_{s^*}$ and therefore $\rho[x \mapsto V] \models_{s^*} C, E \cup \{x : \tau\}$.

$VarDcl_{Comp}$. If $(newL, s^*) = (GetNewLoc\ s\ \tau_\rho\ \texttt{error})$, then by the definition of $GetNewLoc$, $\rho[x \mapsto newL](x) \in Loc_{\tau_\rho} \cap dom(s^*) \subseteq [\texttt{ref}\ \tau_\rho]_{s^*}$ and hence $\rho[x \mapsto newL] \models_{s^*} C, E \cup \{x : \texttt{ref}\ \tau\}$. It is clear that $dom(s) \subseteq dom(s^*)$.

The cases for VarDecls$_{Comp}$ and VarDcl*$_{Comp}$ are similar to those above for constant declarations.

Block$_{Comp}$. By induction hypothesis on the four hypotheses of the rule, $\rho_1 \models C, E_1, \models s_1, \rho_2 \models C, E_2, \models s_2, \models s_3, \models s^*$, and $V \in [\tau_{\rho_2}]_{s^*}$. However, because processing constant and variable declarations does not add or change the values of type variables in the environment, $[\tau_{\rho_2}]_{s^*} = [\tau_\rho]_{s^*}$. The conclusion follows.

Assign$_{Comp}$. By induction hypothesis $\models s^*$, $L \in [\texttt{ref}\ \tau_\rho]_{s^*} = (Loc_{\tau_\rho} \cap dom(s^*)) \cup \{\texttt{error}\}$, $\models s^{**}$ and $V \in [\tau_\rho]_{s^{**}}$. Thus $s^{**}[L \mapsto V](L) \in [\tau_\rho]_{s^{**}}$, so $\models s^{**}[L \mapsto V]$ (if L = error, see rule $Error_{propagating}$).

Value$_{Comp}$. By induction hypothesis $\models s^*, L \in [\texttt{ref}\ \tau_\rho]_{s^*} = (Loc_{\tau_\rho} \cap dom(s^*)) \cup \{\texttt{error}\}$ (if L = error, see rule $Error_{propagating}$). Therefore $s^*(L) \in [\tau_\rho]_{s^*}$.

Function$_{Comp}$. $(\texttt{function}(x{:}\sigma)\ B, \rho, s) \downarrow (\langle \texttt{function}(x{:}\sigma)\ B, \rho\rangle, s)$. It follows easily that $\langle \texttt{function}(x{:}\sigma)\ B, \rho\rangle \in [\sigma_\rho \to \tau_\rho]_s$ by taking $\hat{C} = C$, $\hat{E} = E$, $\hat{\sigma} = \sigma$, $\hat{\tau} = \tau$, and $\eta = \rho$. (Here we are using the assumption that the last step of the proof does not involve subtyping.)

The cases for PolyFunction$_{Comp}$, <# BdPolyFunction$_{Comp}$, and <: BdPolyFunction$_{Comp}$ are similar to Function$_{Comp}$.

FuncAppl$_{Comp}$. By induction hypothesis $\models s_1$ and $\langle \texttt{function}(x{:}\hat{\sigma})\ B, \eta\rangle \in [\sigma_\rho \to \tau_\rho]_{s_1}$. Thus there exist some $\hat{C}, \hat{E}, \hat{\sigma}, \hat{\tau}$ such that

$$\hat{C}, \hat{E} \cup \{x{:}\hat{\sigma}\} \vdash B{:}\hat{\tau},\ \emptyset \vdash \hat{\tau}_\eta <: \tau_\rho,\ \emptyset \vdash \sigma_\rho <: \hat{\sigma}_\eta,\ \text{and}\ \eta \models_{s_1} \hat{C}, \hat{E}.$$

Again by the induction hypothesis we have $\models s_2$ and $V_2 \in [\sigma_\rho]_{s_2}$. By Lemma D.3, $V_2 \in [\hat{\sigma}_\eta]_{s_2}$. Thus $\eta[x \mapsto V_2] \models_{s_2} \hat{C}, \hat{E} \cup \{x{:}\hat{\sigma}\}$. Applying the induction hypothesis one more time we get $\models s_3$ and $V \in [\hat{\tau}_\eta]_{s_3}$. Lemma D.3 gives us $V \in [\tau_\rho]_{s_3}$.

<# *BdPolyFuncAppl*$_{Comp}$. By induction, $\models s_1$ and $\langle \texttt{function}(t\ <\#\ \hat{\gamma})\ B, \eta\rangle \in [\forall t\ <\#\ \hat{\gamma}_\rho.\tau_\rho]_{s_1}$. Thus there exist some $\hat{C}, \hat{E}, \hat{\gamma}, \hat{\tau}$ such that

$$\hat{C} \cup \{t\ <\#\ \hat{\gamma}\}, \hat{E} \vdash B{:}\hat{\tau},\ \{u\ <\#\ \hat{\gamma}_\eta\} \vdash \hat{\tau}_\eta[t \mapsto u] <: \tau_\rho[t \mapsto u],\ \hat{\gamma}_\eta = \gamma_\rho,$$

and $\eta \models_{s_1} \hat{C}, \hat{E}$. By Lemma 4.6, $\emptyset \vdash \sigma_\rho <\#\ \gamma_\rho$, and hence $\eta[t \mapsto \sigma_\rho] \models_{s_2} \hat{C} \cup \{t\ <\#\ \hat{\gamma}\}, \hat{E}$. Applying the induction hypothesis one more time we get $\models s_2$ and $V \in [\hat{\tau}_{\eta[t\mapsto\sigma_\rho]}]_{s_2} = [\hat{\tau}_\eta[t \mapsto \sigma_\rho]]_{s_2}$. Substituting σ_ρ for u we get $\emptyset \vdash \hat{\tau}_\eta[t \mapsto \sigma_\rho] <: \tau_\rho[t \mapsto \sigma_\rho]$. $V \in [\tau_\rho[t \mapsto \sigma_\rho]]_{s_2}$ follows from Lemma D.3.

The cases for PolyFuncAppl$_{Comp}$ and <: BdPolyFuncAppl$_{Comp}$ are similar.

Class$_{Comp}$. By induction,

$$\models s_1,\ V_a \in [\{\ldots, iv_i{:}\forall \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).(\sigma_i)_\rho, \ldots\}]_{s_1},\ \models s_2$$

and

$$V_b \in [\{\ldots, m_j{:}\forall \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).\forall \texttt{InstType} <: \mathit{RecToMem}(\sigma_\rho).\ \texttt{MyType} \to \texttt{InstType} \to (\tau_j)_\rho, \ldots\}]_{s_2}.$$

The claim follows from Lemmas D.4 and D.2.

New$_{Comp}$. By induction, $\models s_1$ and

$$\begin{aligned}&\texttt{class}(\{\ldots, iv_i = \langle \texttt{function}(\texttt{MyType} <\#\ \hat{\gamma}_i)\ B_i, \eta_i\rangle : \sigma_i^{\forall}, \ldots\},\\ &\quad \{\ldots, m_j = \langle \texttt{function}(\texttt{MyType} <\#\ \hat{\gamma}_j)\ B_j, \kappa_j\rangle : \tau_j^{\forall}, \ldots\})\\ &\quad \in [\texttt{ClassType}(\texttt{MyType}, \sigma_\rho, \tau_\rho)]_{s_1}.\end{aligned}$$

Therefore

$$\langle \texttt{function}(\texttt{MyType} <\#\ \hat{\gamma}_i)\ B_i, \eta_i\rangle \in [\forall \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).(\sigma_i)_\rho]_{s_1}$$

and

$$\begin{aligned}\langle \texttt{function}(\texttt{MyType} <\#\ \hat{\gamma}_j)\ B_j, \kappa_j\rangle \in [&\forall \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).\\ &\forall \texttt{InstType} <: \mathit{RecToMem}(\sigma_\rho).\ \texttt{MyType} \to \texttt{InstType} \to (\tau_j)_\rho]_{s_1}.\end{aligned}$$

First, there exist some $\hat{C}_i, \hat{E}_i, \hat{\gamma}_i, \hat{\sigma}_i$ such that

$$\begin{aligned}&\hat{C}_i \cup \{\texttt{MyType} <\#\ \hat{\gamma}_i\},\ \hat{E}_i \vdash_{s_1} B_i : \hat{\sigma}_i,\quad \eta_i \models_{s_1} \hat{C}_i, \hat{E}_i,\\ &\{u <\#\ (\hat{\gamma}_i)_{\eta_i}\} \vdash (\hat{\sigma}_i)_{\eta_i}[\texttt{MyType} \mapsto u] <: (\sigma_i)_\rho[\texttt{MyType} \mapsto u],\end{aligned}$$

and

$$(\hat{\gamma}_i)_{\eta_i} = \texttt{ObjectType}(\texttt{MyType}, \tau_\rho).$$

Obviously

$$\eta_i[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)] \models \hat{C}_i \cup \{\texttt{MyType} <\#\ \hat{\gamma}_i\}, \hat{E}_i.$$

Applying the induction hypothesis again we get

$$\models s_{i+1} \text{ and } V_i \in [(\hat{\sigma}_i)_{\eta_i[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]}]_{s_{i+1}}.$$

Substituting $\texttt{ObjectType}(\texttt{MyType}, \tau_\rho)$ for u we get

$$\begin{aligned}\vdash (\hat{\sigma}_i)_{\eta_i}[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)] <: (\sigma_i)_\rho[\texttt{MyType} \mapsto\\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)].\end{aligned}$$

It follows from Lemma D.3 that

$$V_i \in [(\sigma_i)_\rho[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]]_{s_{i+1}}.$$

This proves that for $1 \le i \le n$, $\models s_{n+i+1}$ and

$$\mathit{newL}_i \in \mathit{Loc}_{(\sigma_i)_\rho[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]} \cap \mathit{dom}(s_{n+i+1}).$$

Analogous to the above steps, we show that $\models s_{2n+2j}$, $\models s_{2n+2j+1}$,

$$\begin{aligned}&\langle \texttt{function}(\texttt{InstType} <: \hat{\sigma}_j)\ \hat{B}_j, \hat{\kappa}_j\rangle\\ &\quad \in [\forall \texttt{InstType} <: \mathit{RecToMem}(\sigma_\rho)[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)].\\ &\qquad \texttt{ObjectType}(\texttt{MyType}, \tau_\rho) \to \texttt{InstType} \to (\tau_j)_\rho[\texttt{MyType} \mapsto\\ &\qquad \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]]_{s_{2n+2j}}\end{aligned}$$

and

$$\begin{aligned}\langle \texttt{function}(\texttt{self}: \varsigma_j)\ \breve{B}_j, \breve{\kappa}_j\rangle \in [&\texttt{ObjectType}(\texttt{MyType}, \tau_\rho) \to\\ &\mathit{RecToMem}(\sigma_\rho)[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)] \to\\ &(\tau_j)_\rho[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]]_{s_{2n+2j+1}}.\end{aligned}$$

Obviously

$$\{\ldots, iv_i = newL_i\colon \mathtt{ref}\ (\sigma_i)_\rho[\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau_\rho)], \ldots\} \in [\mathit{RecToMem}(\sigma_\rho)[\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau_\rho)]],$$

$$\{\ldots, m_j = \langle \mathtt{function}(\mathtt{self}\colon \varsigma_j)\ \breve{B}_j, \breve{\kappa}_j\rangle\colon (\mathtt{MyType} \to \mathit{RecToMem}(\sigma_\rho) \to (\tau_j)_\rho) [\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau_\rho)], \ldots\} \in [\{\ldots, m_j\colon (\mathtt{MyType} \to \mathit{RecToMem}(\sigma_\rho) \to (\tau_j)_\rho) [\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau_\rho)], \ldots\}]_{s_{2n+2k+1}}$$

and by the Object rule,

$$ob \in [\mathtt{ObjectType}(\mathtt{MyType}, \tau_\rho)]_{s_{2n+2k+1}}.$$

Msg_{Comp}. By Lemma 4.6, $\emptyset \vdash \gamma_\rho <\#\ \mathtt{ObjectType}(\mathtt{MyType}, \{m_j\colon (\tau_j)_\rho\})$ and therefore (γ_ρ cannot be a type variable since $C = \emptyset$)

$$\gamma_\rho = \mathtt{ObjectType}\ (\mathtt{MyType}, \{\ldots, m_j\colon \hat{\tau}_j, \ldots\})$$

and

$$\{\mathtt{MyType} <\#\ \mathtt{ObjectType}(\mathtt{MyType}, \{\ldots, m_j\colon \hat{\tau}_j, \ldots\})\} \vdash \{\ldots, m_j\colon \hat{\tau}_j, \ldots\} <\colon \{m_j\colon (\tau_j)_\rho\}. \tag{1}$$

By the induction hypothesis $\models s_1$ and $ob \in [\mathtt{ObjectType}\ (\mathtt{MyType}, \{\ldots, m\colon \hat{\tau}_j, \ldots\})]$. This means that

$$V_a \in [\mathit{RecToMem}(\sigma)[\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau^*)]]_{s_1},$$

and

$$\langle \mathtt{function}(\mathtt{self}\colon \hat{\delta}_{\mathtt{MyType}})\ B_b, \eta_b\rangle \in [(\mathtt{MyType} \to \mathit{RecToMem}(\sigma) \to \tau_j^*) [\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau^*)]]_{s_1}.$$

for some σ and τ^* such that

$$\emptyset \vdash \mathtt{ObjectType}(\mathtt{MyType}, \tau^*) <\colon \mathtt{ObjectType}(\mathtt{MyType}, \{\ldots, m_j\colon \hat{\tau}_j, \ldots\}) = \gamma_\rho,$$

and therefore

$$\{t\colon \mathtt{TYPE}, s <\colon t\} \vdash \tau^*[\mathtt{MyType} \mapsto s] <\colon \{\ldots, m_j\colon \hat{\tau}_j, \ldots\}[\mathtt{MyType} \mapsto t]. \tag{2}$$

There exist $\hat{C}, \hat{E}, \hat{\delta}_{\mathtt{MyType}}, \hat{\xi}$ such that

$$\hat{C}, \hat{E} \cup \{\mathtt{self}\colon \hat{\delta}_{\mathtt{MyType}}\} \vdash B_b\colon \hat{\xi}, \quad \eta_b \models_{s_1} \hat{C}, \hat{E},$$
$$\emptyset \vdash \hat{\xi}_{\eta_b} <\colon (\mathit{RecToMem}(\sigma) \to \tau_j^*)[\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau^*)]$$

and

$$\emptyset \vdash \mathtt{ObjectType}(\mathtt{MyType}, \tau^*) <\colon (\hat{\delta}_{\mathtt{MyType}})_{\eta_b}.$$

$\eta_b[\mathtt{self} \mapsto ob] \models \hat{C}, \hat{E} \cup \{\mathtt{self}\colon \hat{\delta}_{\mathtt{MyType}}\}$ because

$$ob \in [\mathtt{ObjectType}(\mathtt{MyType}, \tau^*)]_{s_1},$$

and, by Lemma D.3, $ob \in [(\hat{\delta}_{\mathtt{MyType}})_{\eta_b}]_{s_1}$. Then by the induction hypothesis

$$\langle \mathtt{function}(\mathtt{inst}\colon \hat{\delta}_{\mathtt{InstType}})\ B_c, \eta_c\rangle \in ([\mathit{RecToMem}(\sigma) \to \tau_j^*)[\mathtt{MyType} \mapsto \mathtt{ObjectType}(\mathtt{MyType}, \tau^*)]]_{s_2}.$$

Similarily we can show that

$$\langle \texttt{function}(x{:}\hat{\alpha})\ B, \kappa\rangle \in [\tau_j^*[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau^*)]]_{s_3}.$$

We have by Equation (2) that

$$\emptyset \vdash \tau^*[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau^*)] <: \{\ldots, m_j{:}\hat{\tau}_j, \ldots\}[\texttt{MyType} \mapsto \gamma_\rho]$$

and by Equation (1),

$$\emptyset \vdash \{\ldots, m_j{:}\hat{\tau}_j[\texttt{MyType} \mapsto \gamma_\rho], \ldots\} <: \{m_j{:}(\tau_j)_\rho[\texttt{MyType} \mapsto \gamma_\rho]\}.$$

Therefore

$$\emptyset \vdash \tau_j^*[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau^*)] <: (\tau_j)_\rho[\texttt{MyType} \mapsto \gamma_\rho]$$

and thus

$$\langle \texttt{function}(x{:}\hat{\alpha})\ B, \kappa\rangle \in [(\tau_j)_\rho[\texttt{MyType} \mapsto \gamma_\rho]]_{s_3},$$

as desired. Obviously $\models s_i$ for all i.

Inherits$_{Comp}$. By induction hypothesis $\models s_1$,

$$V_i^V \in [\forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\tau_1)_\rho, \ldots, m_k{:}(\tau_k)_\rho\}).(\sigma_i)_\rho]_{s_1}$$

and

$$\begin{aligned} V_j^f \in [&\forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\tau_1)_\rho, \ldots, m_k{:}(\tau_k)_\rho\}).\\ &\forall \texttt{InstType} <: \mathit{RecToMem}(\{iv_1{:}(\sigma_1)_\rho, \ldots, iv_n{:}(\sigma_n)_\rho\}).\\ &\texttt{MyType} \to \texttt{InstType} \to (\tau_j)_\rho]_{s_1}. \end{aligned}$$

Then it is easy to see that

$$\breve{V}_i^V \in [\forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\hat{\tau}_1)_\rho, \ldots, m_{k+1}{:}(\tau_{k+1})_\rho\}).(\sigma_i)_\rho]_{s_7}$$

and that

$$\begin{aligned} \breve{V}_i^f \in [&\forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\hat{\tau}_1)_\rho, \ldots, m_{k+1}{:}(\tau_{k+1})_\rho\}).\\ &\forall \texttt{InstType} <: \mathit{RecToMem}(\{iv_1{:}(\sigma_1)_\rho, \ldots, iv_{n+1}{:}(\sigma_{n+1})_\rho\}).\texttt{MyType} \to\\ &\texttt{InstType} \to (\tilde{\tau}_i)_\rho]_{s_7}. \end{aligned}$$

Again by the induction hypothesis, for $i \in \{1, n+1\}$,

$$\begin{aligned} \hat{V}_i^V \in [&\forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\hat{\tau}_1)_\rho, \ldots, m_k{:}(\tau_k)_\rho, m_{k+1}{:}(\tau_{k+1})_\rho\}).\\ &(\sigma_i)_\rho]_{s_2\ (resp.,\ s_3)} \end{aligned}$$

and for $j \in \{1, k+1\}$,

$$\begin{aligned} &\langle \texttt{function}(\texttt{super}{:}\hat{\gamma}_j)\ B_j, \kappa_j\rangle\\ &\quad \in [\{m_1{:}(\tau_1^\forall)_\rho, \ldots, m_k{:}(\tau_k^\forall)_\rho\} \to\\ &\qquad \forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\hat{\tau}_1)_\rho, \ldots, m_{k+1}{:}(\tau_{k+1})_\rho\}).\\ &\qquad\quad \forall \texttt{InstType} <: \mathit{RecToMem}(\{iv_1{:}(\sigma_1)_\rho, \ldots, iv_{n+1}{:}(\sigma_{n+1})_\rho\}).\\ &\qquad\qquad \texttt{MyType} \to \texttt{InstType} \to (\tilde{\tau}_1)_\rho]_{s_4\ (resp.,\ s_5)}. \end{aligned}$$

As in the previous cases we can show that

$$\kappa_j[\texttt{super} \mapsto \{\ldots, m_j = V_j^f : \tilde{\tau}_j^\forall, \ldots\}] \models_{s_5\ (resp.,\ s_6)} \hat{C}, \hat{E} \cup \{\texttt{super}{:}\hat{\gamma}_j\}.$$

Applying the induction hypothesis one more time, we get

$$\hat{V}_j^f \in [\forall \texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m_1{:}(\hat{\tau}_1)_\rho, \ldots, m_{k+1}{:}(\tau_{k+1})_\rho\}).\\ \forall \texttt{InstType} <: \mathit{RecToMem}(\{iv_1{:}(\sigma_1)_\rho, \ldots, iv_{n+1}{:}(\sigma_{n+1})_\rho\}).\\ \texttt{MyType} \rightarrow \texttt{InstType} \rightarrow (\tilde{\tau}_1)_\rho]_{s_6\ (resp.,\ s_7)}$$

Now it is straightforward to see that

$$\texttt{class}(\{iv_1 = \hat{V}_1^V{:}(\hat{\sigma}_1^\forall)_\rho, \ldots, iv_i = \breve{V}_i^V{:}(\hat{\sigma}_i^\forall)_\rho, \ldots, iv_{n+1} = V_{n+1}^V{:}(\hat{\sigma}_{n+1}^\forall)_\rho\},\\ \{m_1 = \hat{V}_1^f{:}(\hat{\tau}_1^\forall)_\rho, \ldots, m_j = \breve{V}_j^f{:}(\hat{\tau}_j^\forall)_\rho, \ldots, m_{k+1} = V_{k+1}^f{:}(\hat{\tau}_{k+1}^\forall)_\rho\})\\ \in [\texttt{ClassType}(\texttt{MyType}, \{iv_1{:}(\sigma_1)_\rho, \ldots, iv_{n+1}{:}(\sigma_{n+1})_\rho\},\\ \{m_1{:}(\hat{\tau}_1)_\rho, \ldots, m_{k+1}{:}(\tau_{k+1})_\rho\})]_{s_7}.$$

It's also obvious that for all i, $\models s_i$.

Closure$_{Comp}$. Is trivial. (Note that $\hat{\sigma}_\eta$ and $\hat{\tau}_\eta$ are closed types.)

Object$_{Comp}$. By the induction hypothesis $\models s_1$, $\models s_2$,

$$V_a \in [\mathit{RecToMem}(\sigma_\rho)[\texttt{MyType} \mapsto \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]]_{s_1}$$

and

$$V_b \in [\{m_j{:}(\texttt{MyType} \rightarrow \mathit{RecToMem}(\sigma_\rho) \rightarrow (\tau_j)_\rho)[\texttt{MyType} \mapsto\\ \texttt{ObjectType}(\texttt{MyType}, \tau_\rho)]\}_{j \le m}]_{s_2}.$$

The claim follows from the typing rule for $\texttt{ObjectType}(\texttt{MyType}, \tau_\rho)$.

Error$_{error}$, *Msg*$_{nil}$, *Msg*$_{error}$, *and Error*$_{propagating}$. These all rely on the fact that $\texttt{error} \in [\tau]_s$ for any closed type τ.

Computations resulting in type errors. These rules cannot apply to well-typed terms as the last condition of each computation rule will never be satisfied. We show here only a few cases.

Variable$_{Env}$. This rule cannot apply to a well-typed x as $dom(C, E) \subseteq dom(\rho)$ by the assumption that $\rho \models C, E$.

Cond$_{Bool}$. Using the type rule *Cond* we have $C, E \vdash_s M{:}\texttt{Bool}$ and by the induction hypothesis $V \in [\texttt{Bool}]_{s_1}$.

Value$_{Location}$. Using the type rule *Value* we have $C, E \vdash_s M{:}\ \texttt{ref}\ \tau$ and by the induction hypothesis $V \in [\texttt{ref}\ \tau_\rho]_{s^*}$. Now $[\texttt{ref}\ \tau_\rho]_{s^*} = (Loc_\tau \cap dom(s)) \cup \{\texttt{error}\}$.

FuncAppl$_{Abstr}$. Using the type rule *FuncAppl* we have $C, E \vdash_s M_1{:}\sigma \rightarrow \tau$ and by the induction hypothesis $V \in [\sigma_\rho \rightarrow \tau_\rho]_{s^*}$. So either V is a closure or V is error.

FuncAppl$_{Arg}$. Using the type rule *FuncAppl* we have $C, E \vdash_s M_2{:}\sigma$. By the induction hypothesis M_2 must reduce to an element in $[\sigma_\rho]_{s_2}$. But $tyerr \notin [\gamma]_{s_2}$ for any closed type γ.

PolymorphicFuncAppl$_{Abst}$. We have three possible type derivations: *PolyFuncAppl*, $<\#$ *BdPolyFuncAppl*, and $<:$ *BdPolyFuncAppl*. In each case, by the induction hypothesis, either V is a closure or V is equal to `error`. If $V = \texttt{error}$ then rule *Error*$_{propagating}$ applies.

New$_{<\#BdPolyFunc}$. Using the type rule *New* we have $C, E \vdash_s M : \texttt{ClassType}(\texttt{MyType}, \sigma, \tau)$. By the induction hypothesis M must reduce to an element in $[\texttt{ClassType}(\texttt{MyType}, \sigma, \tau)_\rho]_{s_1}$. Then either V_i, V_j are closures or they are equal to `error`.

Msg$_{Not\_understood}$. By the induction hypothesis $\models s_1$ and $ob = \texttt{obj}(V_a, \{\ldots, m_{k_i} = \langle \texttt{function}(\texttt{self}: \hat{\delta}_{MT})\ B_b, \eta_b \rangle : \hat{\tau}_{k_i}, \ldots\}) \in [\texttt{ObjectType}(\texttt{MyType}, \hat{\tau})]_{s_1}$ Therefore ob must be an object with the method m_j. □

COROLLARY D.6 (TYPE-SAFETY). *Let $\rho \models_s C, E$ and $\models s$. If $C, E \vdash_s M : \tau$ then it is not the case that $(M, \rho, s) \downarrow (tyerr, s^*)$.*

PROOF. The result follows from Theorem 5.1, since $\texttt{tyerr} \notin [\tau]_s$ for any type τ. □

E. CONCRETE VERSUS ABSTRACT SYNTAX

We provide here the concrete syntax for PolyTOIL and the translation rules from concrete to abstract syntax. To keep the notational overhead as small as possible, we work with a language that is a restricted version of PolyTOIL (only one instance variable and one method in a class, exactly one parameter to functions, no procedures, etc.). We assume that `val`, `self` $\Leftarrow$ (...) and `selfinst`.(...) are inferred and inserted by the type-checker at compile-time; the source language does not require them. The constant `super` is annotated with the `ClassType` of the superclass. This information is also inferred and inserted by the type-checker.

E.1 Concrete Syntax and Types

Prog	::=	`program` *id* ; *Block*
Block	::=	*ConstDecl VarDecl* `begin` *StmtList* `return` *Expr* `end`
ConstDecl	::=	`const` *id* = *Expr* : *TypeExpr*
VarDecl	::=	`var` *id* = *Expr* : *TypeExpr*
ParDecl	::=	*id* : *TypeExpr* |
		id $<\#$ *TypeExpr* |
		id $<:$ *TypeExpr*
StmtList	::=	*id* := *Expr* ; |
		`if` *Expr* `then` *StmtList* `else` *StmtList* `end` ; |
		ϵ
TypeExpr	::=	`bool` | `num` | `MyType` | `void`
		TypeExpr $\rightarrow$ *TypeExpr* |
		`ClassType`({*id* : *TypeExpr*}, {*id* : *TypeExpr*}) |
		`ObjectType` {*id* : *TypeExpr*}

$$
\begin{array}{lll}
\mathit{Expr} & ::= & b \mid \mathit{id} \mid \texttt{nil} \mid \texttt{self} \mid \\
 & & \texttt{val}\ \mathit{Expr} \mid \\
 & & \texttt{selfinst}.\mathit{id} \mid \\
 & & \texttt{super}_{\mathit{ClassType}(\tau_1,\tau_2)}.\mathit{id} \mid \\
 & & \mathit{Expr} <= \mathit{id} \mid \\
 & & \texttt{function}(\mathit{ParDecl})\ :\ \mathit{TypeExpr}\ \mathit{Block} \mid \\
 & & \mathit{Expr}(\mathit{Expr}) \mid \\
 & & \mathit{Expr}[\mathit{TypeExpr}] \mid \\
 & & \texttt{new}(\mathit{Expr}) \mid \\
 & & \texttt{class}\ \mathit{VarDecl}\ \mathit{MethodDecl}\ \texttt{end} \mid \\
 & & \texttt{class inherit}\ \mathit{Expr}\ \texttt{modifying}\ \mathit{id}\ \mathit{MethodDecl}\ \texttt{end} \\
\mathit{MethodDecl} & ::= & \texttt{methods}\ \mathit{id}\ =\ \mathit{function}(\mathit{id}\ :\ \mathit{TypeExpr})\ :\ \mathit{TypeExpr}\ \mathit{Block}
\end{array}
$$

Types

$$
\begin{array}{lll}
\tau & ::= & t \mid c \mid \texttt{ref}\ \tau \mid \tau_1 \to \tau_2 \mid \{l:\tau\} \mid \\
 & & \texttt{ClassType}(\tau_1,\ \tau_2) \mid \texttt{ObjectType}\ \tau \mid \\
 & & \forall t\ <\#\ \tau_1.\tau_2 \mid \forall t\ <:\ \tau_1.\tau_2
\end{array}
$$

E.2 Type-Checking, Subtyping and Matching Rules

OK

$$C, E \vdash \texttt{ok:PROGRAM}$$

Command

$$C, E \vdash\ \texttt{command: COMMAND}$$

Nil

$$C, E \vdash \texttt{nil:Null}$$

Constant

$$\frac{\emptyset \vdash c:\ \texttt{TYPE} \qquad b \in \Im(c)}{C, E \vdash b{:}c}$$

Variable

$$C, E \cup \{\mathit{id}{:}\tau\} \vdash \mathit{id}{:}\tau \quad \mathit{id} \neq \texttt{super}_{\mathit{ClassType}(\tau_1,\tau_2)}$$

Program

$$\frac{C, E \vdash \mathit{block}\ :\ \texttt{COMMAND}}{C, E \vdash \texttt{program}\ p\ ;\ \mathit{block}\ :\ \texttt{PROGRAM}}$$

ConstDecl

$$\frac{C, E \vdash \mathit{exp}\ :\ \tau \quad \mathit{id} \notin \mathit{dom}(E)}{C, E \vdash \texttt{const}\ \mathit{id}\ =\ \mathit{exp}\ :\ \tau\ \diamond\ E \cup \{\mathit{id}\ :\ \tau\}}$$

VarDecl

$$\frac{C, E \vdash \mathit{exp}\ :\ \tau \quad \mathit{id} \notin \mathit{dom}(E)}{C, E \vdash \mathit{var}\ \mathit{id}\ =\ \mathit{exp}\ :\ \tau\ \diamond\ E \cup \{\mathit{id}\ :\ \tau\}}$$

Block

$$\frac{\begin{array}{c} C, E \vdash \mathit{condecl}\ \diamond\ E_1 \quad C, E_1 \vdash \mathit{vardecl}\ \diamond\ E_2 \\ C, E_2 \vdash \mathit{stm}\ :\ \ \texttt{COMMAND} \quad C, E_2 \vdash \mathit{exp}\ :\ \tau \end{array}}{C, E \vdash \mathit{condecl}\ \ \mathit{vardecl}\ \ \texttt{begin}\ \ \mathit{stm}\ \ \texttt{return}\ \ \mathit{exp}\ \ \texttt{end}\ :\ \tau}$$

Assn

$$\frac{C, E \vdash \mathit{id}\ :\ \ \texttt{ref}\ \tau \qquad C, E \vdash \mathit{exp}\ :\ \tau}{C, E \vdash \mathit{id}\ :=\ \mathit{exp}\ :\ \ \texttt{COMMAND}}$$

$$\textit{Cond}\quad \frac{C, E \vdash exp : bool \quad C, E \vdash stm_1 : \texttt{COMMAND} \quad C, E \vdash stm_2 : \texttt{COMMAND}}{C, E \vdash \texttt{if}\ exp\ \texttt{then}\ stm_1\ \texttt{else}\ stm_2\ \texttt{end:}\ \texttt{COMMAND}}$$

$$\textit{Value}\quad \frac{C, E \vdash exp : \texttt{ref}\ \tau}{C, E \vdash \texttt{val}\ exp : \tau}$$

$$\textit{Function}\quad \frac{C, E \cup \{id : \sigma\} \vdash bock : \tau}{C, E \vdash \texttt{function}(id : \sigma) : \tau\ block : \sigma \to \tau}$$

$$<\#\ \textit{BdPolyFunction}\quad \frac{C \cup \{t\ <\#\ \gamma\}, E \vdash block : \tau}{C, E \vdash \texttt{function}(t\ <\#\ \gamma) : \tau\ block : \forall t\ <\#\ \gamma.\tau}$$

$$<:\ \textit{BdPolyFunction}\quad \frac{C \cup \{t\ <:\ \gamma\}, E \vdash block : \tau}{C, E \vdash \texttt{function}(t\ <:\ \gamma) : \tau\ block : \forall t\ <:\ \gamma.\tau}$$

$$\textit{FuncAppl}\quad \frac{C, E \vdash exp_1 : \sigma \to \tau \quad C, E \vdash exp_2 : \sigma}{C, E \vdash exp_1(exp_2) : \tau}$$

$$<\#\ \textit{BdPolyFuncAppl}\quad \frac{C, E \vdash_s exp : \forall t\ <\#\ \gamma.\tau \quad C \vdash_s \sigma\ <\#\ \gamma}{C, E \vdash_s exp[\sigma] : \tau[t \mapsto \sigma]}$$

$$<:\ \textit{BdPolyFuncAppl}\quad \frac{C, E \vdash_s exp : \forall t <: \gamma.\tau \quad C \vdash_s \sigma <: \gamma}{C, E \vdash_s exp[\sigma] : \tau[t \mapsto \sigma]}$$

$$\textit{SupSel}\quad \frac{C^{IV} \vdash \tau' <: \tau}{C^{METH}, E^{METH} \vdash \texttt{super}_{ClassType(\{id:\sigma\},\{m:\tau\})}.m : \tau}$$

where C^{IV}, C^{METH} and E^{METH} have the same form as in rule *Inherits*.

$$\textit{VarSel}\quad \frac{C, E \vdash \texttt{selfinst} : \{id : \sigma\}}{C, E \vdash \texttt{selfinst}.id : \sigma}$$

$$\textit{Class}\quad \frac{C^{IV}, E \vdash exp : \sigma \quad C^{METH}, E^{METH} \vdash f_m : \tau}{\begin{array}{c} C, E \vdash \texttt{class}\ var\ id = exp : \sigma\ \texttt{methods}\ m = f_m\ :\ \tau\ \texttt{end} \\ : \texttt{ClassType}(\{id : \sigma\}, \{m : \tau\}) \end{array}}$$

where

—$C^{IV} = C \cup \{\texttt{MyType}\ <\#\ \texttt{ObjectType}\ \{m : \tau\}\}$
—$C^{METH} = C^{IV} \cup \{\texttt{InstType}\ <:\ RecToMem(\{id : \sigma\})\}$ and
—$E^{METH} = E \cup \{\texttt{self: MyType, selfinst: InstType}\}$

$$\textit{New}\quad \frac{C, E \vdash exp\ \texttt{ClassType}(\{id : \sigma\}, \{m : \tau\})}{C, E \vdash \texttt{new}(exp) : \texttt{ObjectType}\ \{m : \tau\}}$$

$$\textit{Msg} \qquad \frac{C \vdash \gamma <\#\texttt{ObjectType}\ \{m:\tau\} \quad C, E \vdash exp : \gamma}{C, E \vdash exp <= m : (\tau[\texttt{MyType} \mapsto \gamma])}$$

$$\textit{Inherits} \qquad \frac{\begin{array}{c} C, E \vdash exp :\ \texttt{ClassType}(\{id:\sigma\}, \{m:\tau\}), \\ C^{IV} \vdash \tau' <: \tau, \quad C^{METH}, E^{METH} \vdash f_m : \tau' \end{array}}{\begin{array}{l} C, E \vdash \texttt{class inherit}\ exp\ \texttt{modifying}\ m\ \texttt{methods}\ m = f_m : \tau'\ \texttt{end} \\ \quad : \texttt{ClassType}(\{id:\sigma\}, \{m:\tau'\}) \end{array}}$$

where

—$C^{IV} = C \cup \{\texttt{MyType} <\#\ \ \texttt{ObjectType}\ \{m:\tau'\}\}$
—$C^{METH} = C^{IV} \cup \{\texttt{InstType} <: \mathit{RecToMem}(\{id:\sigma\})\}$
—$E^{METH} = E \cup \{\texttt{self: MyType, selfinst: InstType, super}_{\mathit{ClassType}(\{id:\sigma\},\{\mathrm{m}:\tau\})}\texttt{:}$ $\{\mathrm{m}:\tau\}\}$

$$\textit{Subsump} \qquad \frac{C \vdash \sigma <: \tau \quad C, E \vdash exp : \sigma}{C, E \vdash exp : \tau}$$

E.3 Translation Rules

Type Translation

1. $[c] \Rightarrow c$
2. $[t] \Rightarrow t$
3. $[\{l:\delta\}] \Rightarrow \{l:[\delta]\}$
4. $[\sigma \to \tau] \Rightarrow [\sigma] \to [\tau]$
5. $[\texttt{ClassType}(\sigma, \tau)] \Rightarrow \texttt{ClassType}(\texttt{MyType}, [\sigma], [\tau])$
6. $[\texttt{ObjectType}\ \tau] \Rightarrow \texttt{ObjectType}(\texttt{MyType}, [\tau])$
7. $[\delta]^{m,\tau} \Rightarrow \forall \texttt{MyType} <\#\ \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau]\}).[\delta]$
8. $[\delta]^{id,m,\sigma,\tau} \Rightarrow \forall \texttt{MyType} <\#\ \ \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau]\}).$
   $\forall\ \texttt{InstType} <: \mathit{RecToMem}(\{id:[\sigma]\}).\texttt{MyType} \to \texttt{InstType} \to [\delta]$

Expression Translation

1. $[b] \Rightarrow b$
2. $[id] \Rightarrow id$
3. $[\texttt{val}\ exp] \Rightarrow \texttt{val}\ [exp]$
4. $[\texttt{selfinst}.id] \Rightarrow\ \ \texttt{selfinst}.id$
5. $[\texttt{super}_{\mathit{ClassType}(\tau_1,\tau_2)}.id] \Rightarrow \texttt{super}.id\ [\texttt{MyType}]\ [\texttt{InstType}]\ (\texttt{self})\ (\texttt{selfinst})$
6. $[expr \Leftarrow id] \Rightarrow [expr] \Leftarrow id$
7. $[\texttt{function}(id:\gamma):\varphi\ block] \Rightarrow \texttt{function}(id:[\gamma]):[\varphi]\ [block]$
8. $[\texttt{new}(expr)] \Rightarrow \texttt{new}\ [expr]$
9. $[\texttt{class}\ var\ id = exp:\sigma\ \texttt{methods}\ m = f_m:\tau\ \texttt{end}]$
   $\Rightarrow \texttt{class}(\{id = [exp]^{m,\tau}:[\sigma]^{m,\tau}\}, \{m = [f_m]^{id,m,\sigma,\tau}:[\tau]^{id,m,\sigma,\tau}\})$
10. $[\texttt{class inherit}\ exp\ \texttt{modifying}\ m\ \texttt{methods}\ m = f_m:\tau'\ \texttt{end}]$
    $\Rightarrow \texttt{class inherit}\ [exp]\ \texttt{modifying}\ m$

$$(\{\}, \{m = \texttt{function(super}: \{m: [\tau]^{id,m,\sigma,\tau}\})$$
$$\texttt{begin return } [f_m]^{id,m,\sigma,\tau'} \texttt{ end}$$
$$: \{m: [\tau]^{id,m,\sigma,\tau}\} \to [\tau']^{id,m,\sigma,\tau'}\})$$

Here τ is the type of method m in *exp* and σ is the type of the instance variable *id* in *exp*

11. $[exp]^{m,\tau} \Rightarrow$ `function(MyType <# ObjectType(MyType,` $\{m: [\tau]\}$`)) begin return` $[exp]$ `end`
12. $[f_m]^{id,m,\sigma,\tau} \Rightarrow$ `function(MyType <# ObjectType(MyType,` $\{m: [\tau]\}$`))`
    `begin return function(InstType <:` $RecToMem(\{id: [\sigma]\})$`)`
    `begin return function(self : MyType)`
    `begin return function(selfinst : InstType)`
    `begin return` $[f_m]$ `end end end end`

Statement Translation

1. $[id := exp] \Rightarrow id := [exp]$
2. [`if` *exp* `then` stm_1 `else` stm_2 `end`] $\Rightarrow$ `if` $[exp]$ `then` $[stm_1]$ `else` $[stm_2]$ `end`

Block Translation

1. $[const\ id_1 = exp_1 : texp_1\ var\ id_2 = exp_2 : texp_2$ `begin` stm `return` exp_3 `end`]
   $\Rightarrow const\ id_1 = [exp_1] : [texp_1]\ var\ id_2 : [texp_2]$ `begin` $id_3 := [exp_3]$; $[stm]$
   `return` $[exp]$ `end`

Program Translation

1. [`program` *id*; *block*] $\Rightarrow$ `program` *id*; $[block]$

Type Constraints and Type Assignments Translation

1. $[t <: \tau] \Rightarrow t <: [\tau]$
2. $[t <\# \tau] \Rightarrow t <\# [\tau]$
3. $[id: \tau] \Rightarrow id: [\tau]$, $id \neq$ `super`
4. $[\texttt{super}_{ClassType(\{id:\sigma\},\{m:\tau\})}: \{m: \tau\}] \Rightarrow \texttt{super}: \{m: [\tau]^{id,m,\sigma,\tau}\}$

PROPOSITION E.1. *If* $C, E \vdash \tau_1 <: \tau_2$ *then* $[C], [E] \vdash [\tau_1] <: [\tau_2]$. *If* $C, E \vdash \tau_1 <\# \tau_2$ *then* $[C], [E] \vdash [\tau_1] <\# [\tau_2]$.

PROPOSITION E.2. *Let* M *be an expression in the concrete syntax. If* $C, E \vdash M: \tau$ *then* $[C], [E] \vdash_s [M] : [\tau]$.

PROOF. We use induction over the height of type derivation. The interesting cases are *Variable*, *SupSel*, *Class*, and *Inherits*.

[Variable]. The claim holds trivially as $id \neq$ `super`.

[SupSel]. We have to show that

$$\begin{array}{l} [C] \cup \{\texttt{MyType} <\# \texttt{ObjectType(MyType,} \{m: [\tau']\}), \\ \qquad \texttt{InstType} <: RecToMem(\{id: [\sigma]\})\}, \\ [E] \cup \{\texttt{self: MyType, selfinst: InstType, super:} \{\texttt{m:} [\tau]^{id,\texttt{m},\sigma,\tau}\}\} \\ \vdash_s \texttt{super.}m\ [\texttt{MyType}]\ [\texttt{InstType}]\ (\texttt{self})\ (\texttt{selfinst}) : [\tau]. \end{array} \tag{3}$$

Using *Variable* and *Proj* we have:

$$\begin{array}{l}[C] \cup \{\texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau']\}),\\ \qquad\quad \texttt{InstType} <: \mathit{RecToMem}(\{id:[\sigma]\})\},\\ [E] \cup \{\texttt{self: MyType, selfinst: InstType, super:}\{\mathrm{m}:[\tau]^{\mathrm{id,m},\sigma,\tau}\}\}\\ \vdash_s \texttt{super}.m : [\tau]^{id,m,\sigma,\tau}.\end{array} \tag{4}$$

The claim follows using the typing rules *Variable*, $<\#$ *BdPolyFuncAppl*, $<:$ *BdPolyFuncAppl*, *FuncAppl* (twice), and Proposition E.1 applied to the judgment $C^{IV} \vdash \tau' <: \tau$.

[Class]. By the induction hypothesis

$$[C] \cup \{\texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau]\})\}, [E] \vdash_s [exp]:[\sigma] \tag{5}$$

and

$$\begin{array}{l}[C] \cup \{\texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau]\}),\\ \qquad\quad \texttt{InstType} <: \mathit{RecToMem}(\{id:[\sigma]\})\},\\ [E] \cup \{\texttt{self:MyType, selfinst: InstType}\}\\ \vdash_s [f_m]:[\tau].\end{array} \tag{6}$$

We apply the rules *Record* and $<\#$ *BdPolyFunction* to judgment (5) and the rules *Record*, *Function*, (twice), $<:$ *BdPolyFunction* and $<\#$ *BdPolyFunction*, to judgment (6). The claim follows after applying the rule *Class* to the resulting judgments.

[Inherits]. By the induction hypothesis and Proposition E.1:

$$[C], [E] \vdash_s [exp]: \texttt{ClassType}(\texttt{MyType}, \{id:[\sigma]\}, \{m:[\tau]\}), \tag{7}$$

$$[C] \cup \{\texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau']\}\} \vdash [\tau'] <: [\tau] \tag{8}$$

and

$$\begin{array}{l}[C] \cup \{\texttt{MyType} <\# \texttt{ObjectType}(\texttt{MyType}, \{m:[\tau']\}),\\ \qquad\quad \texttt{InstType} <: \mathit{RecToMem}(\{id:[\sigma]\})\},\\ [E] \cup \{\texttt{self:MyType, selfinst: InstType, super:}\{\mathrm{m}:[\tau]^{\mathrm{id,m},\sigma,\tau}\}\}\\ \vdash_s [f_m]:[\tau'] \qquad .\end{array} \tag{9}$$

We apply the rules *Record*, *Function* (twice), $<:$ *BdPolyFunction*, $<\#$ *BdPolyFunction*, and one more time *Function* to judgment (9). The claim follows after applying the rule *Inherits* to the resulting judgment and judgments (7) and (8). □

REFERENCES

AMADIO, R. AND CARDELLI, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15*, 4, 575–631.

ABADI, M. AND CARDELLI, L. 1994a. A theory of primitive objects: Second-order systems. In *Proceedings of ESOP '94*, Springer-Verlag, New York, 1–24.

ABADI, M. AND CARDELLI, L. 1994b. A theory of primitive objects: Untyped and first-order systems. In *Proceedings of Theoretical Aspects of Computer Software*, Springer-Verlag, New York, 296–320.

ABADI, M. AND CARDELLI, L. 1995. An imperative object calculus. In TAPSOFT '95: Theory and Practice of Software Development, P.D. Mosses and M. Nielsen, Eds., Lecture Notes in Computer Science, vol. 915, Springer-Verlag, New York, 471–485.

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag, New York.

ARNOLD, K. AND GOSLING, J. 1996. *Java*. Addison-Wesley, Reading MA.

AMADIO, R. M. 1991. Recursion over realizability structures. *Inf. Comput. 91*, 1, 55–86.

ABADI, M. AND PLOTKIN, G. D. 1990. A PER model of polymorphism and recursive types. In *Proceedings of the Symposium on Logic in Computer Science*, 355–365.

BRUCE, K. B., CRABTREE, J., DIMOCK, A., MULLER, R., MURTAGH, T., AND VAN GENT, R. 1993. Safe and decidable type checking in an object-oriented language. In *Proceedings of the ACM Symposium on Object-Oriented Programming: Systems, Languages, and Applications*, 29–46.

BRUCE, K. B., CRABTREE, J., AND KANAPATHY, G. 1994. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, Eds., *Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, vol. 802, Springer-Verlag, New York, 603–626.

BRUCE, K. B., FIECH, A., AND PETERSEN, L. 1997. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of ECOOP '97*, Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, New York, 104–127.

BRACHA, G. AND GRISWOLD. D. 1993. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the ACM Symposium on Object-Oriented Programming: Systems, Languages, and Applications*, 215–230.

BRUCE, K. B. AND LONGO, G. 1990. A modest model of records, inheritance and bounded quantification. *Inf. Comput. 87*, 1/2, 196–240.

BRUCE, K. B. AND MITCHELL, J. C. 1992. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 316–327.

BRUCE, K. B. 1993. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 285–298.

BRUCE, K. B. 1994. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Funct. Program. 4*, 2, 127–206. An earlier version of this paper appeared in the 1993 *POPL Proceedings*.

BRUCE, K. B. 2002. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA.

BRUCE, K. B. AND VAN GENT, R. 1993. TOIL: A new type-safe object-oriented imperative language. Tech. Rep. Williams College.

CARDELLI, L. 1988. A semantics of multiple inheritance. *Inf. Comput. 76*, 138–164. (Special issue devoted to the *Symposium on Semantics of Data Types* (Sophia-Antipolis France, 1984).

CARDONE, F. 1989. Relational semantics for recursive types and bounded quantification. In *ICALP*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, New York, 164–178.

CANNING, P., COOK, W. R., HILL, W., MITCHELL, J. C., AND OLTHOFF, W. 1989. F-bounded quantification for object-oriented programming. In *Funct. Prog. Comput. Arch.* 273–280.

COOK, W. R., HILL, W. L., AND CANNING, P. S. 1990. Inheritance is not subtyping. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages* (January), 125–135.

COOK, W. R. 1989. A proposal for making Eiffel type-safe. In *Proceedings of the European Conference on Object-Oriented Programming*, 57–72.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surv. 17*, 4, 471–522.

DAY, M., GRUBER, R., LISKOV, B., AND MEYERS, A. C. 1995. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the ACM Symposium on Object-Oriented Programming: Systems, Languages, and Applications*, 156–168.

ELLIS, M. A. AND STROUSTROP, B. 1990. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA.

EIFRIG, J., SMITH, S., TRIFONOV, V., AND ZWARICO, A. 1994. Application of OOP type theory: State, decidability, integration. In *Proceedings of OOPSLA '94*, 16–30.

FISHER, K., HONSELL, F., AND MITCHELL, J. C. 1993. A lambda calculus of objects and method specialization. *Nordic J. Comput. 1*, 3–37. An earlier version of this paper appeared in *Proceedings of the Eighth IEEE Symposium on Logic in Computer Science*, 1993, 26–38.

FISHER, K. AND MITCHELL, J. C. 1998. On the relationship between classes, objects, and data abstraction. *TAPOS 4*, 3–32.

GAWECKI, A. AND MATTHES, F. 1996. Integrating subtyping, matching and type quantification: A practical perspective. In *Proceedings of ECOOP '96*, Lecture Notes in Computer Science, vol. 1098, Springer-Verlag, New York, 26–47.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk–80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.

GUNTER, C. A. 1992. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA.

IGARASHI, A., PIERCE, B., AND WADLER, P. 1999. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA Proceedings* (October). Full version to appear in *ACM Trans. Program. Lang. Syst.* 2001.

KATIYAR, D., LUCKHAM, D., AND MITCHELL, J. 1994. A type system for prototyping languages. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, 138–150.

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, NJ.

MEYER, B. 1995. Static typing and other mysteries of life. Tech. Rep., Interactive Software Engineering, Inc. Text of invited address to *OOPSLA '95*.

MITCHELL, J. C. 1990. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages* (January), 109–124.

MADSEN, O., MAGNUSSON, B., AND MOLLER-PEDERSEN, B. 1990. Strong typing of object-oriented languages revisited. In *OOPSLA-ECOOP '90 Proceedings*, 140–150. *ACM SIGPLAN Not. 25*, 10, (Oct.).

PIERCE, B. C. 1993. Mutable objects. Tech. Rep., University of Edinburgh.

PIERCE, B. C. 1994. Bounded quantification is undecidable. *Inf. Comput. 112*, 1, (July), 131–165. Reprinted in *Theoretical Aspects of Object-Oriented Programming*, Gunter and Mitchell, Eds., MIT Press, Cambridge, MA, 427–459. Summary in *POPL* '92.

PIERCE, B. C. AND TURNER, D. N. 1993. Object-oriented programming without recursive types. In *Proceedings of the Twentieth ACM Symposium Principles of Programming Languages*, 299–312.

PIERCE, B. C. AND TURNER, D. N. 1994. Simple type-theoretic foundations for object-oriented programming. *J. Funct. Program. 4*, 207–247. An earlier version appeared in *Proceedings of of POPL* '93, 299–312.

REYNOLDS, J. C. 1980. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, N. D. Jones, Ed., Lecture Notes in Computer Science, vol. 94, Springer-Verlag, New York, 211–2580.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. 1986. An introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, 9–16. *ACM SIGPLAN Not. 21*, 11, (Nov.).

TESLER, L. 1985. Object Pascal report. Tech. Rep. 1, Apple Computer.

VAN GENT, R. 1993. TOIL: An imperative type-safe object-oriented language. Williams College Senior Honors Thesis.

Received July 2000; revised December 2001; accepted August 2002