Subtyping is not a good "Match" for object-oriented languages *

Kim B. Bruce[1], Leaf Petersen[1]**, and Adrian Fiech[2]

[1] Williams College, Williamstown, MA, USA
[2] Memorial University of Newfoundland, St. John's, Newfoundland, Canada

Abstract. We present the design and rationale of a new statically-typed object-oriented language, $\mathcal{LOOM}$. $\mathcal{LOOM}$ retains most of the features of the earlier language **PolyTOIL**. However the subtyping relation is dropped from $\mathcal{LOOM}$ in favor of the matching relation. "Hash types", which are defined in terms of matching, are introduced to provide some of the benefits of subtyping. These types can be used to provide support for heterogeneous data stuctures in $\mathcal{LOOM}$. $\mathcal{LOOM}$ is considerably simpler than **PolyTOIL**, yet is just as expressive. The type system for the language is decidable and provably type safe. The addition of modules to the language provides better control over information hiding and allows the provision of access like that of C++'s friends.

1 Introduction

Most statically-typed object-oriented programming languages, including C++ [ES90], Java [AG96], Object Pascal [Tes85], and Modula 3 [CDG+88], suffer from very rigid type systems which can block the easy expression of programmers' ideas, particularly in the definition of subclasses. Other statically-typed object-oriented languages, such as Eiffel [Mey92] and Beta [MMMP90], are more flexible, but require run-time or link-time checks in order to eliminate holes in the type system. One important goal of our previous work in designing static type systems for object-oriented languages has been to design flexible, yet safe, static type systems (see for example [Bru93, Bru94, BSvG95]). In this paper we propose a fairly radical departure from previous statically-typed object-oriented languages by dropping subtyping in favor of a relation called matching and a new type constructor related to matching.

It has become clear in the last several years that the concepts of inheritance and subtyping in object-oriented languages are not at all the same. (See [Sny86] for early hints, and [CHC90, AvdL90, LP91] for more definitive treatments of the topic.) In earlier papers [Bru93, Bru94, BSvG95], we introduced the notion of *matching*, a relation between object types which is more general than subtyping.

* Bruce and Petersen's research was partially supported by NSF grant CCR-9424123. Fiech's research was partially supported by NSERC grant OGP0170497.

** Current address: School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA

A history of matching can be found in [BSvG95]. See also [AC95] for a discussion of explanations of matching in terms of F-bounded and higher-order subtyping.

An important feature of matching is that if two classes are subclasses, then the corresponding object types are in the matching relation. However, these types are not generally in the subtype relation if the language includes a type expression, *MyType*, standing for the type of *self*. Since *MyType* is extremely valuable, we end up with a mismatch between subtyping and inheritance.

The language PolyTOIL [BSvG95] supports both subtyping and matching. It also supports bounded polymorphism where the bounds on type parameters are given in terms of the matching relation. Unfortunately, the relations of matching and subtyping are similar enough that it might be difficult for a programmer to keep their differences straight. As a result we began considering the possibility of simplifying the language by dropping one or the other.

It was clear that matching was absolutely necessary in PolyTOIL in order to express the type-checking rules for subclasses, as well as to provide constraints for polymorphic operations and data structures. Subtyping was too restrictive to serve either of these purposes. Moreover, as we looked at the sample programs we had written in PolyTOIL, we noticed that we rarely used subtyping. In the presence of *MyType* and match-bounded polymorphism, we almost never needed to explicitly change the types of methods in subclasses.

With this in mind, we decided to be bold and investigate the results of eliminating subtyping in favor of matching. While this caused no problems with the inheritance mechanism, we did discover a few situations where we found it difficult to do without subtyping. To deal with this we introduced a new type constructor, $\#T$, which represents elements whose types match the type T. With this addition, we found that we had a language which was essentially as expressive as **PolyTOIL**, yet significantly less complex.

In the next section we review briefly the main features of **PolyTOIL**. In section 3 we introduce the language $\mathcal{LOOM}$. We pay special attention to the introduction of a new type constructor which can be applied to object types to form types which have properties similar to subtypes (but are less restrictive). In the following section we provide a brief overview of the technical results used to justify the use of the language: type-checking is decidable and the language is provably type-safe. We provide a brief overview of $\mathcal{LOOM}$'s module system in section 5. Finally in section 6 we summarize and evaluate the features of the language.

2 Introduction to PolyTOIL and Matching

In [BSvG95] we introduced **PolyTOIL**, a statically-typed object-oriented programming language which provides all of the usual constructs of class-based object-oriented languages. In particular, it supports objects which consist of instance variables and methods, and in which computation is based on message-sending. Reuse is supported via inheritance, subtyping, and a form of bounded polymorphism.

We refer the reader to [BSvG95] for the details of the language. We provide only a brief summary here. Relatively innovative features of the language include:

- The use of *MyType* in classes to refer to the type of *self*, the receiver of the message.
- The treatment of classes as first-class values of the language. They have types and can be both parameters to functions and returned as the results of functions.
- The separation of the inheritance and subtyping hierarchies, in order to avoid problems which can arise with so-called binary methods.
- The introduction of a new ordering called "matching", written $<\#$, which corresponds closely to type changes allowed in defining subclasses. Subtyping, written $<:$, is also supported.
- Support for a form of bounded polymorphism, called *match-bounded polymorphism* in which the bound is expressed in terms of the new matching ordering.

Within the scope of each object (or class), **PolyTOIL** defines a type identifier *MyType* that represents the type of the object itself. This type is "anchored" to the type of the object in which it appears, so that an appearance of *MyType* in the type of a method inherited from another class represents the type of an instance of the current class, rather than the parent class in which the method was defined.

Matching is a relation between types which is very similar to subtyping, but is designed to correspond better to the inheritance hierarchy of classes and objects. An object type τ *matches* an object type σ in **PolyTOIL**, if for every method $m_i\colon\sigma_i$ included in object type σ there is a corresponding method $m_i\colon\tau_i$ included in object type τ such that $\sigma_i <: \tau_i$ (where occurrences of the keyword *MyType* in the method types are treated as uninterpreted free variables).

The definition of subtyping for object types is similar to that for matching, but also requires that *MyType* only occur positively[3] in method types (see [BSvG95] for definitions and details). In particular, an object type with a method with parameter of type *MyType* cannot have any non-trivial subtypes. As a result, if a class has a method with a parameter of type *MyType*, then subclasses will not give rise to subtypes.

While $\sigma <: \tau$ implies that any element of type σ can be treated as being of type τ, the import of matching is weaker. If $\sigma <\# \tau$ then any message sent to an object of the type τ could also be sent to an object of type σ.

MyType and matching are crucial to the expressiveness of **PolyTOIL**. *MyType* is often used in expressing the types of methods whose types will need to change in subclasses. It appears frequently in association with *copy* or *clone*

[3] Roughly, an occurrence of an identifier is positive (or covariant) if it is on the left side of an even number of function types, and is negative (or contravariant) if it is on the left of an odd number of function types. In particular, if the return type of a function is *MyType*, then that occurrence of *MyType* is positive. On the other hand, if a parameter type of a function is *MyType*, then that occurrence is negative.

```
class Node(n: Integer)
   var
      val = n: Integer;
      next = nil: MyType;
   methods
      function getVal(): Integer  {return val.clone() }
      procedure setVal(newVal: Integer) {val := newVal.clone()}
      function getNext(): MyType {return next}
      procedure setNext(newNext:MyType) {next := newNext}
      procedure attachRight(newNext: MyType) {self.setNext(newNext)}
   end class;

 class DbleNode(n: Integer)
         inherits Node(n) modifying attachRight
   var
      prev = nil: MyType
   methods
      function getPrev():MyType {return prev}
      procedure setPrev(newPrev: MyType){prev := newPrev}
      procedure attachRight(newNext: MyType) {self.setNext(newNext);
           newNext.setPrev(self) }
 end class;
```

Fig. 1. Singly and doubly-linked node classes in **PolyTOIL**.

methods and so-called *binary* methods – methods whose parameter should always be the same type as the receiver. Bounded polymorphism based on matching proves to be a very useful tool in object-oriented programming – so much so that when programming in **PolyTOIL**, we found that it (in combination with the use of *MyType*) almost completely replaced the use of subtype polymorphism as a mechanism for producing generic code.

Figure 1 provides an example of *Node* and *DbleNode* class definitions. Notice that for *n: Integer*, class *DbleNode(n)* inherits from *Node(n)*. The occurrences of *MyType* in *Node* implicitly denote the type of objects generated by *Node*, while in *DbleNode* they denote the type of objects generated by *DbleNode*. It follows easily from the definition of matching in PolyTOIL that the types generated by these classes are in the matching relation. However they are not subtypes because *setNext* and *attachRight* have parameters of type *MyType*.

Nevertheless it is possible to define a polymorphic list class that, when instantiated with the type of objects generated by *Node*, will generate singly-linked lists, while if it is instantiated with the type of objects generated by *DbleNode* will generate doubly-linked lists. (See [BSvG95] for the code and details.) The fact that these types are not subtypes is not a handicap in using them in the ways intended.

These results suggested to us that subtyping might not be necessary at all.

At the same time, we began to feel that presenting programmers with both the subtyping and matching hierarchies on types in a single language might be confusing.

The paper [GM96] in ECOOP '96 presented a language TooL, which is more general 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 the language was too complex for programmers, and suggested dropping matching. Based on our experience, we prefer instead to give up subtyping. In the rest of this paper, we describe the language $\mathcal{LOOM}$, which we designed as a simpler replacement for **PolyTOIL**.

3 $\mathcal{LOOM}$: Core Language

As a result of our concerns about the complexity of **PolyTOIL**, we developed the language $\mathcal{LOOM}$. $\mathcal{LOOM}$ retains the syntax and match-bounded polymorphism of **PolyTOIL**, but completely abandons the subtyping relation in favor of a simplified version of matching. As a whole, $\mathcal{LOOM}$ is a much simpler language than its predecessor.

In abandoning subtyping, we also decided not to allow the types of inherited methods to be changed when overridden. While this may seem like a step backwards in the design of more flexible type systems, our experience indicates that the presence of the *MyType* construct (standing for the type of *self*) combined with match-bounded polymorphism provides sufficient flexibility for redefining methods in subclasses.

Because we wish to have matching correspond to the changes in types allowed in defining subclasses, we provide a more restricted definition of matching in $\mathcal{LOOM}$ compared to that of **PolyTOIL**. We write object types in $\mathcal{LOOM}$ in the form

```
ObjectType
   m1:T1;
   ...
   mn:Tn
end;
```

or more compactly, $ObjectType\{m_1\colon T_1;\ldots;m_n\colon T_n\}$.

Definition 1. Given object types $ObjectType\{m_1\colon T_1;\ldots;m_n\colon T_n\}$ and $ObjectType\{m_1\colon T_1;\ldots;m_k\colon T_k\}$, define

$$ObjectType\{m_1\colon T_1;\ldots;m_n\colon T_n\} <\# \; ObjectType\{m_1\colon T_1;\ldots;m_k\colon T_k\}$$

iff $k \leq n$. The relation $<\#$ is referred to as "matching".

In both **PolyTOIL** and [AC95], the matching relation allowed the types of corresponding methods to be subtypes. Since the corresponding types must now be the same, a matching type simply is an extension of the original. It might seem that this would significantly impact the expressiveness of the language. While there are some occasions where this restriction forces us to introduce type parameters, it is our experience that most of the time when we wish to have a change in the type of a method, the use of *MyType* provides the necessary change implicitly.

3.1 Hash Types Replace the Use of Subtyping

In object-oriented languages, subtyping allows a value of one type to masquerade as a value of a supertype. This permits the definition of data structures and operations that treat heterogeneous objects with common functionality. This is primarily used in two contexts. The first is in applying functions to parameters which are subtypes of their declared type. The second is for assignments to variables. While the first problem can be handled fairly elegantly with match-bounded polymorphism, the second is essentially impossible in **PolyTOIL** without subtype polymorphism.

In those places where we might have passed in a parameter that was a subtype of the declared type of the parameter, we could instead provide the function or procedure with an extra type parameter that was bounded (using matching) by the originally declared type of the parameter. If we had *procedure p(w: Window)* in **PolyTOIL** taking a parameter of type *Window*, then it could also be applied to parameters which were subtypes of *Window*. In $\mathcal{LOOM}$ we could rewrite it as *procedure* $p'(T <\# \, Window; w\colon T)$ with appropriately modified body. This procedure is actually now more general than the original because it can be applied to objects whose type merely matches that of *Window* rather than being a subtype. In particular, if *Window* contains a method with a parameter of type *MyType*, there are many types which will match it, but none which are proper subtypes.

This rewriting of programs has one major disadvantage – it introduces extra type parameters into many of the method definitions. One of the advantages of subtyping is that it is possible to write one simple function definition with a parameter of a fixed type, yet apply it to arguments of many more types. We address this problem below.

The use of subtyping in assignment statements occurs frequently when defining heterogeneous data structures. In a language with subtyping, we can relatively easily create a data structure in which all data values are subtypes of some fixed type. For instance, in programming a graphic user interface one might want to maintain a heterogeneous list of windows. From the point of view of maintaining the list, one need only require that each list element supports certain window operations. This can be captured most naturally by assuming each has a type which matches some general window type, rather than using subtyping. (After all, some of the methods might include negative occurrences of *MyType*.) We wish to be able to handle this in our language.

In implementing these heterogeneous data structures, we often have an instance variable to which we wish to assign the values to be stored in the structure. Generally it is assumed that the type of the value to be stored is a subtype of the declared type of the variable. However, as above, we have found that assuming it matches the declared type is often a better description of what is actually needed. Unfortunately, we know no way of modeling the flexibility of subtyping for variable assignments using match-bounded polymorphism.

To solve these problems, we introduce a new type, $\#\tau$, for τ an object type. If a variable or formal parameter is declared to have type $\#\tau$ then it will be able to hold values of any type matching τ.[4]

A simple example should illustrate the use of hash types. Suppose we have a class in **PolyTOIL** with an instance variable *aWindow* of type *Window*, and a method

```
procedure setWindow(newWindow:Window){aWindow := newWindow}
```

We can rewrite this in $\mathcal{LOOM}$ by changing the type of *aWindow* to be $\#Window$ and replacing the method by

```
procedure setWindow(newWindow:#Window){aWindow := newWindow}
```

Now we can pass a variable of any type matching *Window* to the method. Note that if *Window* had a binary method, then it has no subtypes, so only objects of type *Window* could be passed in to the original *setWindow*. However, there will be many types matching *Window*, so the revised method may be more flexible than the original.

The type-checking rules *Weakening* and *Subsump* in Appendix C state the essential properties of these new types: If e has type τ and $\tau <\!\# \sigma$, then e also has type $\#\sigma$. We refer to types of the form $\#\tau$ as "hash" types, in reference to the constructor.

We can now rewrite the procedure declaration above nearly as simply as with subtyping by writing *procedure* $p'(w\colon \#Window)$ in place of the longer polymorphic definition above. Similarly we can declare a variable to have type $\#\tau$ if it is to hold values with types that match τ.

3.2 A Heterogeneous Ordered List Program in $\mathcal{LOOM}$

Before presenting the syntax and type-checking rules of $\mathcal{LOOM}$ formally, we provide an extended example of a heterogeneous list program. We begin by defining parameterized singly-linked and doubly-linked nodes in $\mathcal{LOOM}$, and work up to defining a class which can be instantiated to provide either singly-linked or doubly-linked heterogeneous ordered lists.

In Figure 2 we provide examples of parameterized *HetNode* and *HetDoubleNode* classes in $\mathcal{LOOM}$. Unlike the earlier examples of Nodes from **PolyTOIL**, they are now parameterized by the type of value stored in the node as well as

[4] Some readers may find it useful to think of $\#\tau$ as an abbreviation for the type $\exists t <\!\# \tau.t$ of the second order lambda calculus. (See [CW85, MP88].)

```
class HetNode(T <# Object; v: #T)
   var
      value = v: #T;
      next = nil: MyType;
   methods
      function getValue(): #T  {return value.clone() }
      procedure setValue(newValue: #T) {value := newValue.clone()}
      function getNext(): MyType {return next}
      procedure setNext(newNext:MyType) {next := newNext}
      procedure attachRight(newNext: MyType) {self.setNext(newNext)}
   end class;

 class HetDbleNode(T <# Object; v: #T)
         inherits HetNode(T,v) modifying attachRight
   var
      prev = nil: MyType
   methods
      function getPrev():MyType {return prev}
      procedure setPrev(newPrev: MyType){prev := newPrev}
      procedure attachRight(newNext: MyType) {self.setNext(newNext);
           newNext.setPrev(self) }
 end class;
```

Fig. 2. Polymorphic singly and doubly-linked node classes.

the initial value for that node. The upper bound *Object* on the type parameter T is a built-in type which every object type matches. (It is similar to class *Object* of Java.) Because the instance variable *value* is declared to have type $\#T$, it can hold a value of any type matching T. Similarly, the methods *getValue* and *setValue* use hash types. On the other hand, the instance variables *next* and *prev* can only hold values with type exactly *MyType*. Thus the *next* field of an object generated by class *HetNode* can only hold nodes, while the corresponding field from class *HetDbleNode* can only hold doubly-linked nodes. Note that for fixed U $<\#$ *Object* and u: $\#U$, *HetDbleNode(U,u)* inherits from *HetNode(U,u)*.

We can create new objects from the *HetNode* class by first supplying the parameterized class with actual parameters and then applying the *new* operation to it. Thus if U is an object type and u:$\#U$, then *new HetNode(U,u)* results in the creation of a new node object whose *value* field is initialized to u.

The type of objects is written using the *ObjectType* constructor. The type functions *HetNodeType* and *HetDbleNodeType* defined in Figure 3 describe the types of objects generated by the classes *HetNode* and *HetDbleNode*.

The keyword *include* used in the definition of *HetDbleNodeType* indicates that all methods of *HetNodeType* are included in *HetDbleNodeType*, as well as the new methods declared there. This may be thought of as a form of inheritance for types. It is modelled on a similar construct in Rapide [KLM94a], and is

```
HetNodeType(T <# Object) = ObjectType
               getValue: Func():#T;
               setValue: Proc(#T);
               getNext: Func():MyType;
               setNext: Proc(MyType)
               attachRight: Proc(MyType)
            end ObjectType;

HetDbleNodeType (T <# Object) = ObjectType include HetNodeType
               getPrev: Func():MyType;
               setPrev: Proc(MyType);
            end ObjectType;
```

Fig. 3. Types for singly and doubly-linked node classes.

essentially the same as the use of *extends* on interfaces in Java [AG96].

It follows from the definition that if $U \mathrel{<\#} Object$, then *HetDbleNodeType(U)* $<\#$ *HetNodeType(U)*. Of course, there is an *implicit* change of type resulting from the occurrences of *MyType* in the types of *getNext*, *setNext*, and *attachRight*, but the definition of matching is sensitive only to *explicit* changes in types.

The use of *MyType* ensures that instance variable, parameter, and return types change appropriately when new classes are defined by inheritance, and new types are defined using *include*. The type checking rule *Msg* in Appendix C specifies that when a message is sent to an object whose corresponding method involves the type *MyType*, all occurrences of *MyType* in the method type are replaced by the type of the receiver. For example, if a *setNext* message is sent to an object, the actual parameter must have the same type as the receiver. In particular, if *sn: HetNodeType(U)* for some fixed type U, then the type of parameter *newActualNext* in *sn.setNext(newActualNext)* must also be *HetNodeType(U)* for the message send to be type correct. On the other hand, if *dn: HetDbleNodeType(U)*, then the type of *newActualNext* in *dn.setNext(newActualNext)* must be *HetDbleNodeType(U)*.

```
OrdEltType = ObjectType
                gt, eq: Func(#OrdEltType):Boolean;
                ...
            end ObjectType;

OrdNodeType = HetNodeType(OrdEltType);

OrdDbleNodeType = HetDbleNodeType(OrdEltType)
```

Fig. 4. Node types for ordered elements.

```
class OrdList(N <# OrdNodeType)
  var
     head = nil: N;
  methods
     function find(match:#OrdEltType): Boolean
       var
         current: N;
       { current := head;
         while (current != nil) & match.gt(current.getValue()) do
           current := current.get_next()
         end;
         return (current != nil) & (current.getValue()).eq(match))}

     procedure addNode(newNode:N)
      ...
 end;

OrdListType(N <# OrdNodeType) = ObjectType  -- parameterized type
      find: Func(#OrdEltType): Boolean;
      addNode: Proc(N);
  end ObjectType;
```

Fig. 5. Heterogeneous linked list.

We define *OrdEltType* in Figure 4 to be an object type which includes operators which allow comparisons with elements of any type matching *OrdEltType*. We wish to construct a heterogeneous list, each of whose elements has a type matching *OrdEltType*. The individual nodes holding those values will have type *OrdNodeType* if we wish to have a singly-linked list, or type *OrdDbleNodeType* if we wish to have a doubly-linked list (see Figure 4). However, because *OrdDbleNodeType* $<\#$ *OrdNodeType*, we can take advantage of the polymorphism by creating a list class which is parameterized so that the type of its nodes is a fixed type which matches *OrdNodeType*.

In Figure 5 we provide the definition of a parameterized class which generates heterogeneous lists of elements whose types all match *OrdEltType*. The figure also includes the type function *OrdListType* which describes the objects generated by the class.

Because *OrdList* can be applied to any type which matches *OrdNodeType*, we can easily instantiate the lists to be either singly or doubly-linked. Thus

slist := new OrdList(OrdNodeType)

creates a new singly-linked heterogeneous list, while

dlist := new OrdList(OrdDbleNodeType)

creates a new doubly-linked heterogeneous list. If *elt* has any type which matches *OrdEltType* then *slist.find(elt)*, *slist.addNode(new Node(OrdEltType, elt))*, *dlist.find(elt)*, and *dlist.addNode(new DbleNode(OrdEltType, elt))* are correctly typed message sends which look up or add new elements to singly or doubly-linked lists.

3.3 Type Checking and Hash Types

The rules for forming legal type expressions and defining matching in $\mathcal{LOOM}$ are given in Appendix B, while the most important type-checking rules can be found in Appendix C.

The type-checking rules for classes are in a slightly different style from those in [BSvG95]. This slight variation makes the proof of type-safety somewhat simpler. The keyword *self* is of type *MyType*, and represents the object from the outside. It is used as in the examples given earlier to send messages to the object executing the code, or it can be used to pass the object as a parameter to a method of another object. (The keyword *self* may be omitted in terms of the form *self.m()* in the implemented $\mathcal{LOOM}$ interpreter, as the interpreter can infer where it needs to occur.) The keyword *selfVar* represents the record of instance variables of the object, which are only visible within the object's methods. In the examples given earlier in this paper, if v is an instance variable, we have chosen to simply write v rather than *selfVar.v*. Again the interpreter will infer when this is needed and insert it if it is omitted.

The addition of hash types provides most of the original functionality of subtype polymorphism. It allows us to write heterogeneous data structures and to write functions that operate on all objects whose type matches their parameter. However there are some new wrinkles in type checking that should be noted.

As stated earlier, by the *Weakening* and *Subsumption* type-checking rules, it is possible for elements of a fixed type σ to be assigned to a variable with type $\#\tau$ for $\sigma <\!\!\# \, \tau$, but not vice-versa. Similar rules hold for the correspondence between formal and actual parameters of functions and procedures.

We discussed above the type-checking rule for sending messages, *Msg*, which appears in Appendix C. For that rule, we needed to know the exact type of the receiver. But what happens if all we know about the receiver o is that it has a type of the form $\#\sigma$? If the method to be applied has a type τ in which all occurrences of *MyType* are positive (e.g., *MyType* does not occur as the type of a parameter), then one can apply the same typing rule. That is, the type of $o.m$ is $\tau[\#\sigma/MyType]$. (See rule *Msg#* in the appendix.) It can be shown via an inductive proof that this rule is sound. However, if the method type τ includes a negative (contravariant) occurrence of *MyType*, then the rule does not hold.

A concrete example should help illustrate the reasons for the failure. Suppose *aNode* has type $\#OrdNodeType$. Then *aNode.attachRight(bNode)* will not be well-typed. The value of *aNode* at run-time might be of any type matching *OrdNodeType*. If the value held in *aNode* actually has type *OrdNodeType* at run-time, then the type of the parameter *bNode* must be *OrdNodeType*. On the other hand, if the type of the value held in *aNode* is actually *OrdDbleNodeType* at run-time, then the type of the parameter *bNode* must instead be *OrdDbleNodeType*. Since we cannot determine the type of the value statically, we cannot determine which is the correct type for the parameter. (Note that having an actual parameter with type $\#OrdNodeType$ makes the situation even worse!)

Thus if the receiver of a message has a hash type, we cannot send it a binary message, or indeed any message whose type involves an occurrence of *MyType* in a contravariant position.

A practical consequence of this is that if we have a heterogeneous data structure, as is the case in Figure 5 above, we can only send non-binary messages to its elements (at least in the absence of a type-case statement – or equivalent – which allows the run-time checking of types). One might suspect that this is the reason why the type *OrdEltType* in the program has no binary methods. In fact, this simply results from thinking through the logical design of these methods. If *ge* and *eq* had types of the form *func(#MyType):Boolean*, then there would be no way of comparing successive elements of the list, since their values could be of incomparable types matching *OrdEltType*. In that case one could not send a comparison method to one using the other as a parameter. Thus it would be a logical mistake to try to use a binary method there.

It is also worth noting here that we would be stuck in this case in a language supporting subtyping as well. If the type τ of elements in the list had a binary method, then no proper subtypes of τ would exist, forcing the list to be homogeneous. In $\mathcal{LOOM}$, such binary methods might exist in τ, as long as they aren't used in the methods of the heterogeneous data structure.

Finally in many cases where a binary method needs to be sent to a parameter x with a hash type of the form $\#\tau$, the function declaration can be rewritten to take an explicit type parameter $t \lessdot\!\# \, \tau$ and parameter $x\colon t$. In this way we have an explicit name for the type of x, and can provide an actual parameter for a binary method with a parameter which is also of type t.

Thus while this restriction on typing message sends corresponding to binary methods seems to be a limitation, the limitations are actually less severe than in languages with subtyping.

4 Decidability, Natural Semantics, and Type Safety

In this section we outline some important results for $\mathcal{LOOM}$ whose details will be provided elsewhere. These include the decidability of type checking for $\mathcal{LOOM}$, the provision of natural semantics, and the proof of type safety.

Pierce's [Pie92] results on the undecidability of subtyping in the second-order bounded polymorphic lambda calculus has caused designers of languages supporting bounded polymorphism to be concerned about whether their type systems are undecidable. While $\mathcal{LOOM}$ does not support subtyping, one might be concerned that match-bounded polymorphism could lead to the same difficulty. Luckily this turns out not to be the case. In fact the determination of whether two types match is relatively straightforward since one object type matches another only if the first type contains all of the methods (with corresponding types) of the second. As usual it is straightforward to replace the general transitivity rule for matching with a restricted rule which is computationally easy to deal with (and in fact we have included only this restricted rule in Appendix B).

The algorithm for type checking terms is fairly straightforward. As with **PolyTOIL**, the complexity of type-checking is non-polynomial in the worst case, since in contrived families of examples the type of a term can grow exponentially fast in the length of the term. In practice we find type checking in $\mathcal{LOOM}$ to be acceptably fast (at most quadratic in the size of the type).

In order to prove the type safety of $\mathcal{LOOM}$, we need to show that the type system is consistent with a semantics for the language. We can define a natural semantics for $\mathcal{LOOM}$ similar to that provided for **PolyTOIL** [BSvG95]. Once defined we can prove:

1. If a term of $\mathcal{LOOM}$ has type τ (possibly under some assumptions on free term and type variables) and if the evaluation of the term halts (when started with an environment and state that are consistent with the assumptions), then the value will also have type τ.[5]
2. If a term is well-typed then the evaluation will not get "stuck". That is, it will not get to a point in the evaluation where the partial result is not a designated "reduced value", yet no computation rule applies.

The proofs of these results are similar to those in **PolyTOIL**, but are simpler because of the lack of subtyping. The only terms which require extra work are message sends to hash types, but a proof by induction on the structure of terms suffices to show that the type-checking rule is sound. Needless to say, the assumption that the method type includes only positive occurrences of *MyType* is crucial to the proof.

Details of the semantics, formal statement of the theorems, and the proofs will be given in an extended form of this paper.

5 Adding Modules to LOOM

As programmers and language designers have gotten more experience with object-oriented languages, it has become clearer that class boundaries are not always the correct abstraction layer for large programs. Interestingly it is the hybrid languages which have grown from ADT-style languages which have often provided the best support for this modularity. Thus Ada 95 [Int95] and Modula-3 [CDG$^+$88] have introduced module structures which are distinct from classes. Pure object-oriented languages like Smalltalk and Eiffel have generally chosen to identify classes with modules, though newer languages like Theta [DGLM94] (which can be seen as a descendant of the ADT-style language, Clu) and Java [AG96] have also chosen to provide modules (called packages in Java).

[5] If τ involves free type variables then the type of the value will have a type obtained by replacing the free type variables of τ by corresponding values from the environment.

Modules provide three important functions in programming languages:

1. They provide a way of organizing code into manageable pieces and help with name-space management by only exporting names that are needed elsewhere.
2. They provide important abstraction barriers, lessening the dependence of a unit on the implementation details of another unit.
3. They provide support for separate compilation, aiding the programmer in debugging large programs as well as making it possible to provide reusable code in libraries in such a way that the original source code need not be revealed.

With strong language support for modules it should be possible to change the implementation of a type without changing its public interface and without requiring recompilation of other modules which import it. See [Jon96] for a more detailed discussion of modules.

We have chosen to follow the lead of languages like those in the Modula and Ada families and provide module interfaces which can be completely separate from the module implementation (e.g., the interface can be compiled before the implementation is written). A key issue is to provide a mechanism for revealing in the interface as much or as little as desired about the details of the implementation of a type.

In ADT-style languages it is typical for type definitions in interfaces to consist either of *manifest* types, in which all details of the type definition are revealed, or *opaque* types, in which only the name of the type is revealed. However in ADT-style languages, operations are defined externally from the type itself. Thus an opaque type will be accompanied by a collection of names of constants, functions, and procedures related to it that are to be exported (though their implementations are typically hidden).

In object-oriented languages, however, object types include the names and types of their methods. In ADT-style languages it is easy to reveal only certain operations, leaving others hidden in the implementation module. We can emulate this in object-oriented languages by providing partial revelations of object types as is done in Modula-3. That is, for each object type we only list those methods which we wish to publicly advertise. This can be expressed using the matching relation from $\mathcal{LOOM}$.

The following example is similar to one presented in [PT93]. If we declare

```
IntSetType <# ObjectType
                add: proc(Integer);
                remove: proc(Integer);
                contains: func(Integer):Boolean;
                intersect: proc(MyType)
              end;
```

in an interface, then we know that IntSetType has methods *add*, *remove*, *contains*, and *intersect* with the appropriate types. As before, we note that subtyping would not be an appropriate relation (even if it was supported in $\mathcal{LOOM}$)

because the fact that *intersect* has a parameter of type *MyType* implies that the given object type has no proper subtypes!

$\mathcal{LOOM}$, like **PolyTOIL**, allows the programmer to label methods as either Visible or Hidden, depending on whether or not we wish them to be available outside of the object's own methods.[6] Also an object's instance variables are not accessible outside the object's methods. Why then do we need the extra mechanism of partial revelations? The reason is that some objects might need to provide extra methods which are only accessible to a limited number of other types of objects. This is exactly the idea behind C++'s friends and is similar to the provision of different levels of access in Ada 95 and Java.

Let us continue with the example above of sets of integers. In order to implement the intersection operation efficiently we may need to have more access to the representation of the sets. Appendix A contains an example of a collection of $\mathcal{LOOM}$ modules which supports the efficient implementation of integer sets as ordered lists.

There are several points worthy of remark in this example. First notice that *OrdListType* is a manifest type. The user has full access to the type and to the class *OrdListClass* which generates objects of that type. However the type *IntSetType* is only partially revealed. Inside the implementation module it is defined as an extension of *OrdListType* (and hence matches *OrdListType*), but that information is not available outside of the implementation module. Hence one may not deduce in any other module that $IntSetType <\# OrdListType$. This represents good programming practice since one should not in general be able to use list operations on sets. Notice also that because *OrdListType* is not exported by *Interface SetOfInt*, the name space is not cluttered up with details about lists.

A second important point about this example is that our (efficient) implementation of *intersect* depends on being able to use the fact that *IntSetType* is implemented as an ordered list. If, for example, all of the list operations (such as *first*, *next*, and *deleteCur*) were hidden methods, we would not be able to send the corresponding messages to the parameter *other*. If there were methods of other classes which also needed access to these non-exported methods, we would simply include their definitions in the same module. Thus we can achieve the effect of C++'s friends without letting the non-exported methods escape the module boundary.

This technique for providing information hiding while providing access by operations at the same level of abstraction is similar to that suggested in [PT93] and [KLM94b], where bounded existential quantifiers based on subtyping are used to perform the information hiding.

In summary, our implementation of modules in $\mathcal{LOOM}$ allows the programmer to specify how much information about an object type is revealed to other program units. While it is possible to have a totally opaque type, this should

[6] For simplicity, we have not labeled the methods in the earlier examples, though all would be labeled as *Visible*. Our use of *Visible* corresponds to C++ and Java's *public*, while *Hidden* corresponds to their *protected*.

be relatively rare with the object-oriented style of programming. The user can reveal all of the details of a type definition by providing a *manifest* type, or may reveal only a portion of the type by providing a matching bound on the type. Function, procedure, class, and other constant declarations can also be exported by placing them in a module interface. Our module design satisfies all of the goals listed at the beginning of this section.

6 Evaluation of LOOM

One of the most difficult decisions to make in designing a programming language is not so much what features should be included, but rather which features can (or should) be excluded. Based on our previous work, we concluded that subtyping was a feature that could be excluded from object-oriented languages. In this paper we presented the language $\mathcal{LOOM}$, which dropped subtyping in favor of matching, and added hash types.

Aside from subtyping, features of $\mathcal{LOOM}$ include all of those in our earlier language **PolyTOIL**. Because of the lack of subtyping, we decided not to allow types of methods to be changed when defining subclasses.[7] Similarly, matching now requires corresponding method types to be the same, rather than subtypes. To replace other uses of subtyping we introduced a new type constructor which can be applied to object types to obtain "hash types". With the addition of the hash types, the elimination of subtyping has very little impact on the expressiveness of the language.

Another advantage of $\mathcal{LOOM}$ is that the matching relation is relatively trivial, only allowing the addition of new methods to object types, as compared to subtyping, which is defined on all types and is often quite complex. The covariant and contravariant rules for subtyping functions are a good example of the complexity of subtyping that we can simply ignore in $\mathcal{LOOM}$.

Binary methods have been hard to express in most statically-typed object-oriented programming languages because they seem to require a *MyType* term representing the type of *self*, yet *MyType* and subtyping have not interacted well. See [BCC+96] for an exposition of the difficulties of handling binary methods in statically-typed object-oriented languages. In that paper, matching is suggested as one way of getting around these difficulties, but it is also criticized as not providing support for heterogeneous data structures. In this paper we have shown how to provide that support by the use of hash types.

One possible disadvantage of $\mathcal{LOOM}$ is that the programmer must explicitly mark the types of variables and formal parameters which are allowed to hold values of more than one type by declaring them with hash types. Variables and parameters with non-hash types are only allowed to hold values of the type with which they are declared. The programmer is required to "plan ahead" which slots are allowed to hold values which match because the typing rules for message sending are different for hash types than for regular types. It is worth noting in

[7] An early version of $\mathcal{LOOM}$ did allow types to be changed, but we decided that the gains were not worth the added complexity.

this regard that Ada 95 [Int95] also requires the programmer to mark the types of variables and parameters with the "*class*" suffix if values which are subclasses of the declared type may be used.

A complication of $\mathcal{LOOM}$ compared to **PolyTOIL** is that binary messages may not be sent to hash types. As discussed in the previous section, this has little practical impact since binary methods are generally not appropriate for heterogeneous data, and match-bounded polymorphism can usually be used in place of hash types where binary methods are needed.

We noted that the type-checking algorithm for $\mathcal{LOOM}$ is decidable, with low complexity in practice. Also, the static type system for $\mathcal{LOOM}$ guarantees that no type errors will occur during the execution of type-checked programs, so $\mathcal{LOOM}$ is type-safe.

$\mathcal{LOOM}$ also provides significant features for information hiding. All instance variables are hidden from clients, while methods are marked as being either visible or hidden in order to determine their visibility to clients. All instance variables and methods are visible to inheritors. We described briefly the module facility in the language which provides for separate compilation, restriction of the scope of names, and information hiding. Like Modula-3, it provides extra support for information hiding by the use of partial revelations of types. This fine control over information hiding can be used to provide objects defined in the module access to more features than those outside, providing a feature similar to C++'s friends. This also allows more efficient implementation of binary methods.

We are currently examining mechanisms to allow modules to support multiple interfaces. This would make it possible to provide different views of a module to different clients. For instance, a client wishing to define a subclass of a class in a module will need much more detailed information about the superclass than a client which simply wishes to be able to generate objects from the class and use them.

We have implemented an interpreter for $\mathcal{LOOM}$, which is based on the natural semantics of the language. In spite of the seemingly radical step of eliminating subtyping, our experience is that $\mathcal{LOOM}$ provides a conceptually simpler language than **PolyTOIL**, yet provides essentially the same expressiveness as the earlier language. It may be time to rethink the notion that subtyping is an essential part of object-oriented languages.

Acknowledgements: The $\mathcal{LOOM}$ language design is due to Bruce and Petersen. A more complete description of the language design, type-checing rules, natural semantics, and the analysis of complexity of type-checking can be found in Petersen's honors thesis [Pet96]. The proof of type safety was done by Fiech with the assistance of Bruce, and was based on a similar proof for **PolyTOIL**. *The $\mathcal{LOOM}$ implementation was primarily accomplished by Petersen with assistance from Hilary Browne, and was based on the* **PolyTOIL** *interpreter written by Robert van Gent and Angela Schuett, with assistance from Petersen and Jasper Rosenberg.*

References

[AC95] Martin Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167, 1995.

[AG96] Ken Arnold and James Gosling. *Java.* Addison Wesley, 1996.

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA-ECOOP '90 Proceedings*, pages 161–168. ACM SIGPLAN Notices,25(10), October 1990.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object-Oriented Systems*, 1996. to appear.

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

[Bru94] K. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994. An earlier version of this paper appeared in the 1993 POPL Proceedings.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[CDG+88] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC systems Research Center, 1988.

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

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

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

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

[GM96] Andreas Gawecki and Florian Matthes. Integrating subtyping, matching and type quantification: A practical perspective. In *ECOOP '96*, pages 26–47. LNCS 1098, Springer-Verlag, 1996.

[Int95] Intermetrics. *Ada 95 Reference Manual, version 6.0.* 1995.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *23rd ACM Symp. Principles of Programming Languages*, pages 68–78, 1996.

[KLM94a] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *21st ACM Symp. Principles of Programming Languages*, pages 138–150, 1994.

[KLM94b] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium of Principles of Programming Languages, Portland, Oregon*, pages 138–150. Association for Computing Machinery, January 1994.

[LP91] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, pages 57–62, January 1991.

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

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

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[Pet96] Leaf Petersen. *A module system for LOOM.* Williams College Senior Honors Thesis, 1996.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proc 19th ACM Symp. Principles of Programming Languages*, pages 305–315, 1992.

[PT93] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, 1993.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. 1st ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

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

A Sample LOOM Program for Integer Sets Using Modules

A row of "*"s indicates a module boundary. The intersection method of SetOfInt is destructive – the receiver of the message is updated to hold the value of the intersection.

```
Interface IntOrdList;

type

   OrdListType = ObjectType
      first: proc();
      next: proc();
      off: func():Boolean; -- is current elt off end of list?
      add: proc(Integer);
      deleteCur: proc();  -- current is next elt after deleteCur
      contains: func(Integer):Boolean;
      getCur: func():Integer
   end;

   OrdListClassType = ClassType ... end;
      -- class types include instance variable and method types
```

```
const

   OrdListClass: OrdListClassType;

end;

****************************************************

Interface SetOfInt;

type

   IntSetType <# ObjectType
      add: proc(Integer);
      remove: proc(Integer);
      contains: func(Integer):Boolean;
      intersect: proc(MyType)
   end;

const

   function newSet(): IntSetType;

end; -- Interface SetOfInt

****************************************************

Module Implements SetOfInt import IntOrdList;
-- SetOfInt is implemented as a specialized ordered list in order
-- to make the methods find and intersect more efficient.

type

   IntSetType = ObjectType include IntOrdList::OrdListType
      remove :proc(Integer);
      intersect: proc(MyType)
   end;

   ListSetClassType = ClassType
          include IntOrdList::OrdListClassType;
      methods visible
         remove :proc(Integer);
         intersect: proc(MyType)
      end;
```

```
const

   ListSetClass = class inherit IntOrdList::OrdListClass
      methods visible
         procedure remove(elt:Integer) is
            begin
               if find(elt) then deleteCur()
            end;

         procedure intersect(other:MyType) is
                                       -- destructive intersection
            begin
               first();
               other.first();
               while (not off()) and (not other.off) do
                  if getCur() < other.getCur() then
                     deleteCur()
                  else if getCur() > other.getCur then
                        other.next()
                     else
                        next();
                        other.next()
                     end -- else if
                  end -- if
               end -- while
               while not off() do
                  deleteCur()
               end -- while
            end -- function
         end;  -- class

   function newSet():IntSetType;
      begin
         return new(ListSetClass)
      end;

end -- Module
```

B Type Formation and Matching Rules

There are rules which determine which type expressions are well-formed. While there is not room to include these in this extended abstract, the well-formed type expressions are the following: a type variable or constant, a function type of the form *Func*(σ):τ, a polymorphic function type *Func*$(s <\# \sigma)$:τ where σ is an object type or type variable, a regular or polymorphic procedure type, a record type of the form $\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\}$, an object type of the form *ObjectType* τ (for τ a record type), an expression of the form *ClassType*(σ,τ) (for σ and τ record types) which represents the type of a class whose instance variables have type σ and methods have type τ, or an expression of the form $\#\tau$ (where τ is an object type or variable).

Reference types are generated automatically by the system from variable declarations. The type of a variable holding values of type τ is written as *ref* τ.

We say that $\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\}$ *extends* $\{m_1{:}\,\tau_1;\ldots;m_k{:}\,\tau_k\}$, if $n \geq k$. Matching is only defined on object types and variables.

$$Var(<\!\#) \qquad C \cup \{t <\!\# \tau\} \vdash t <\!\# \tau,$$

$$Refl(<\!\#) \qquad \frac{C \vdash \tau <\!\# \mathit{Object}}{C \vdash \tau <\!\# \tau},$$

$$Trans(<\!\#) \qquad \frac{C \vdash \tau <\!\# \gamma}{C \cup \{t <\!\# \tau\} \vdash t <\!\# \gamma},$$

$$ObjectType(<\!\#) \qquad \frac{\tau \ \mathit{extends}\ \tau'}{C \vdash \mathit{ObjectType}\ \tau <\!\# \mathit{ObjectType}\ \tau'},$$

C Selected Type-checking Rules for $\mathcal{LOOM}$

C is a collection of type constraints of the form $t <\!\# \tau$, while E is a collection of type assignments to identifiers. We have omitted many of the rules which are the same as for PolyTOIL. In particular we have omitted the rules for declarations and most commands aside from assignment.

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

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

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

where σ may be of the form $\#\gamma$.

$$BdPolyFunc \frac{C \cup \{t <\!\# \gamma\}, E \ \vdash \mathit{Block}{:}\, \tau}{C, E \vdash \text{function } (t <\!\# \gamma)\ \mathit{Block}{:}\ (\mathit{Func}\ (t <\!\# \gamma){:}\, \tau)}$$

FuncAppl
$$\frac{C, E \vdash f\colon Func(\sigma)\colon \tau \qquad C, E \vdash M\colon \sigma}{C, E \vdash f(M)\colon \tau}$$

BdPolyFuncAppl
$$\frac{C, E \vdash f\colon Func(t \mathrel{<\!\!\#} \gamma)\colon \tau \qquad C \vdash \sigma \mathrel{<\!\!\#} \gamma}{C, E \vdash f[\sigma]\colon \tau[\sigma/t]}$$

Record
$$\frac{C, E \vdash a_i\colon \sigma_i,\ \text{for } 1 \le i \le n}{C, E \vdash \{v_1 = a_1; \ldots; v_n = a_n\}\colon \{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}}$$

Proj
$$\frac{C, E \vdash a\colon T,\ \ C \vdash T \ \mathit{extends}\ \{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}}{C, E \vdash a.v_i\colon \sigma_i} \text{ if } 1 \le i \le n$$

Class
$$\frac{C^{IV}, E \vdash a\colon \sigma, \quad C^{METH}, E^{METH} \vdash e\colon \tau}{C, E \vdash \text{class}(a, e)\colon \mathit{ClassType}(\sigma, \tau)}$$

where $C^{IV} = C \cup \{\mathit{MyType} \mathrel{<\!\!\#} \mathit{ObjectType}\ \tau\}$,
$C^{METH} = C^{IV} \cup \{\mathit{SelfVarType}\ \mathit{extends}\ \mathit{MkRef}(\sigma)\}$,
$E^{METH} = E \cup \{\mathit{self}\colon \mathit{MyType}, \mathit{selfVar}\colon \mathit{SelfVarType}\}$
$\mathit{MkRef}\{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\} = \{v_1\colon \mathit{ref}\ \sigma_1; \ldots; v_n\colon \mathit{ref}\ \sigma_n\}$
Neither *MyType* nor *SelfVarType* may occur free in C or E.
σ and τ must both be record types,
while the components of τ must be function types.

Object
$$\frac{C, E \vdash c\colon \mathit{ClassType}(\sigma, \tau)}{C, E \vdash \text{new } c\colon \mathit{ObjectType}\ \tau}$$

Msg
$$\frac{C, E \vdash o\colon \gamma, \quad C \vdash \gamma \mathrel{<\!\!\#} \mathit{ObjectType}\{m\colon \tau\}}{C, E \vdash o \Leftarrow m\colon \tau[\gamma/\mathit{MyType}]}$$

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

Only if all occurrences of *MyType* in τ_i are positive.

Inherits
$$\frac{\begin{array}{c} C, E \vdash c\colon \mathit{ClassType}(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}), \\ C^{IV}, E \vdash a_{m+1}\colon \sigma_{m+1}, \quad C^{IV}, E \vdash a'_1\colon \sigma_1, \\ C^{METH}, E^{METH} \vdash e_{n+1}\colon \tau_{n+1}, \quad C^{METH}, E^{METH} \vdash e'_1\colon \tau_1 \end{array}}{\begin{array}{l} C, E \vdash \text{class inherit } c \text{ modifying } v_1, m_1; \\ \qquad (\{v_1 = a'_1\colon \sigma_1, v_{m+1} = a_{m+1}\colon \sigma_{m+1}\}, \\ \qquad \{m_1 = e'_1\colon \tau_1, m_{n+1} = e_{n+1}\colon \tau_{n+1}\})\colon \\ \qquad\qquad \mathit{ClassType}(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}, \\ \qquad\qquad\qquad \{m_1\colon \tau_1; m_2\colon \tau_2; \ldots; m_{n+1}\colon \tau_{n+1}\}) \end{array}}$$

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

$Weakening$ $$\frac{C, E \vdash e{:}\,\tau}{C, E \vdash e{:}\,\#\tau}$$

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

This article was processed using the LaTeX macro package with LLNCS style.