Semantics-Driven Language Design: Statically Type-Safe Virtual Types in Object-Oriented Languages

Kim B. Bruce and Joseph C. Vanderwaart [1]

Department of Computer Science
Williams College
Williamstown, MA 01267 USA
kim@cs.williams.edu

Abstract

The virtual class [15] construct was first introduced in the language Beta to provide added expressiveness when used with inheritance. Unfortunately, the virtual class construct in Beta is not statically type-safe. In this paper we show how a generalization of the semantics of object-oriented languages with a `MyType` construct leads to a variant of virtual classes which needs no run-time checks. This results in an object-oriented language in which both parametric types and virtual classes (or types) are well-integrated, and which is statically type-safe.

Keywords: Language design, semantics, virtual classes, parametric polymorphism, static type checking

---

1 Introduction

In this paper we develop a statically safe variant of Beta's "virtual classes", a construct that, as originally designed and implemented, required dynamic type checking. Interestingly, the examination of the semantics of object-oriented languages with "`MyType`" led to the discovery of the type-safe variant and its typing rules.

While most object-oriented languages have chosen to support genericity by introducing parametric polymorphism, the object-oriented language Beta [13]

[1] Partially supported by NSF grants CCR-9424123 and CCR-9870253. This paper was prepared while the first author was a visiting professor at Princeton University, partially supported by NSF grant CCR-9625413. Thanks to members of the Princeton Computer Science Department for their hospitality.

has introduced a construct called virtual classes [14,15] to play a similar role. This construct has received renewed attention with the proposal by Thorup [24] to include a version of virtual classes in Java. (We prefer the term *virtual types* introduced by Thorup, and will use that terminology in the rest of the paper.)

Virtual types may be introduced in a class definition by including a type variable as one of the components of the class. The type variable is given an initial definition in the class, but that definition may be overridden in subclasses. Thus the "virtual classes" are to be thought of in the same way as C++'s virtual methods, in that each may be overridden in subclasses. Just as there are restriction on overriding methods, virtual types may only be overridden by replacing the initial value by a subtype or subclass. Replacing a type by a subtype (or a class by a subclass) is called a *covariant* change in type.

We illustrate virtual types/classes with an example adapted from Thorup's paper, written in a Java-like style. It is an implementation of the Observer pattern from [10].

```
class Observer
    typedef SType as Subject;
    typedef EventType as Object;
    void notify (SType subj, EventType e)...;
    ...


class Subject
    typedef OType as Observer;
    typedef EventType as Object;
    OType observers[];
    ...
    void notifyObservers(EventType e)
        int len = observers.length;
        for (int i = 0; i < len; i++)
            observers[i].notify(this,e);



class WindowObserver extends Observer
    typedef SType as WindowSubject;
    typedef EventType as WindowEvent;
    void notify (SType subj, EventType e)
        ... subj.windowMeth ...
```

```
    ...

class WindowSubject extends Subject
    typedef OType as WindowObserver;
    typedef EventType as WindowEvent;
void windowMeth(...)...;
    ...
```

The keyword `typedef` is used to introduce a virtual type definition as part of a class definition. The definition of virtual type `SType` as `Subject` in the class `Observer` is overridden with the subclass `WindowSubject` in class `WindowObserver`. Similarly, the definition of `EventType` as `Object` in class `Observer` is overridden with subclass `WindowEvent` in class `WindowObserver`. Virtual types are also overridden in defining the subclass `WindowSubject` of `Subject`.

In the program above, the first two classes, `Observer` and `Subject`, are mutually referential, while both depend on `EventType` (which is initially `Object`). The method `notify` of `Observer` may rely on properties of `Subject` (and `EventType`). Similarly, the methods of `Subject` rely on properties of `Observer`.

Both are then specialized simultaneously in order to deal with windows. The class `WindowObserver` needs to refer to `WindowSubject` rather than the less specialized `Subject` class because the method `windowMeth` is sent to parameter `subj` in the body of the overridden `notify` method. Similarly, we presume that `WindowSubject` needs to refer to `WindowObserver` rather than `Observer` in order for the bodies of some new or overridden methods to make sense. We also assume both need to refer to `WindowEvent`s rather than just plain `Event`s. Notice that the needed specializations result in classes being overridden by subclasses.

Bruce, Odersky, and Wadler [5] compared the expressiveness of parametric polymorphism and virtual types. As noted there, it is quite difficult to express the type of parallel specialization in the above example with parametric polymorphism, even if F-bounded or match-bounded polymorphism is used.

While the primary focus in this paper is supporting the use of virtual types for parallel specialization, we note that virtual types may also be used to specialize types in a single class. For example, both Thorup [24] and Igarashi and Pierce [12] define class `Animal` with virtual type `FoodType` initially defined to be type `Food`. A subclass `Cow` is defined in which the virtual type `FoodType` is overridden to be `Grass`.

We tend to prefer to write examples like this by making `Animal` a parameterized class with type variable `FoodType`. Class `Cow` can then be defined as an extension of the results of instantiating `Animal` with the class `Grass`.

However an advantage of writing these as virtual classes in Beta is that a variable of type `Animal` can hold objects of type `Cow` or other extensions of `Animal`. While there are type safety problems with this, it may nevertheless be useful to construct such heterogeneous collections. We discuss a statically type safe (but somewhat restricted) way of supporting this usage of virtual types in section 8.

While it appears that virtual types increase the expressiveness of object-oriented languages, it has not been at all clear how they could be supported in a statically type-safe way.

Bruce, Odersky, and Wadler [5] also proposed an extension of Java which provides a construct similar to virtual types yet is statically type-safe. No justification for the safety of that mechanism was included there. In this paper we explain how such an extension to an object-oriented language can be explained by an underlying translational semantics into the higher-order bounded polymorphic lambda calculus. Type-checking rules for the object-oriented language can then be read off from the type-checking rules for the underlying lambda calculus. In this paper we base the extension on our language $\mathcal{LOOM}$. The paper [5] includes a version more appropriate for extending Java.

This paper is organized as follows. In section 2 we discuss the problems with static type-checking virtual types. In section 3 we introduce the type expression `MyType` and the matching relation as used in our language $\mathcal{LOOM}$. This will lead us in section 4 to a generalization of `MyType` that provides support for a statically type-safe version of virtual types. In section 5 we discuss the semantics of `MyType` and virtual types in terms of a translation to $F_{\leq}$. In the following section we continue by showing how to encode classes and the generation of new objects in $F_{\leq}$. In section 7 the type-checking rules for `MyType` and virtual types are explained in terms of their encodings. In section 8 we discuss uses of virtual types in creating heterogeneous collections. Finally, we compare our work with other recent work on virtual types and conclude.

We presume the reader is familiar with the representation of type parameters in the polymorphic lambda calculus [11,22], and with existential types [20]. Mitchell [18] and [19] are good references for these topics.

2 Static typing problems with virtual types

It is well-known (see [9], for example) that the covariant redefinition of instance variable and method parameter types in subclasses can result in the loss of type safety in programs, but this flexibility is often exactly what is desired. In fact, the language Eiffel [16] has a construct (anchored types) which provides very similar capabilities, with the arguments for allowing such covariant changes based on its usefulness.

The simultaneous covariant redefinitions of `SType`, `OType`, and `EventType` allowed by Beta are very useful for the program in the introduction, but the following code shows why virtual types can lead to the failure of static type safety.

```
Observer obs = new WindowObserver();
   // allowed since WindowObserver is a subclass of Observer
Subject subj = new Subject();
...
obs.notify(subj);
```

Statically, it appears that there is no problem with the message send since `obs` has type `Observer` and the method `notify` of type `Object` requires a parameter of type `Subject`. However, because `obs` actually holds an object of class `WindowObserver`, the method executed for `notify` expects a parameter of type `WindowSubject`, and its body will include the message send of `windowMeth` to `subj`. Because `Subject` has no such method (it is first introduced in `WindowSubject`), a run-time error results.

Recognizing this possibility, Beta inserts a check at run-time with such a message send in order to ensure that the parameter is consistent with the actual method executed at run-time. Thorup proposes a similar check by inserting dynamic type checks in his proposed extension of Java. The Beta designers and Thorup argue that many of these dynamic checks can be eliminated by the compiler. We return to this issue later.

A similar program may be written in Eiffel, and its execution causes a run-time error with current compilers. Proposals for a "system validity check" [16] and, more recently, for "no polymorphic CAT-calls" [17] have been put forward by Bertrand Meyer, the designer of Eiffel, to fix this problem. To our knowledge, none of these have been implemented in existing compilers, so current implementations of Eiffel remain type-unsafe.

An important goal of this paper is to identify statically type-safe type-checking rules that will rule out type-unsafe code such as that above, while supporting the type-safe uses of virtual types.

3 The `MyType` type expression in $\mathcal{LOOM}$

Our encoding of virtual types in the higher-order bounded polymorphic lambda calculus is based on the encoding of `MyType` in our language, $\mathcal{LOOM}$ [4]. In this section we provide a brief overview of the use of `MyType` in $\mathcal{LOOM}$.

$\mathcal{LOOM}$ is an object-oriented programming language which grew out of our earlier languages TOOPLE [6] and PolyTOIL [7]. $\mathcal{LOOM}$ is class-based and is provably type-safe. $\mathcal{LOOM}$ carefully distinguishes between classes and types in that a class cannot be used as a type for objects. Instead of using classes as types, there is an `ObjectType` type constructor which specifies the types of all public methods of objects generated from the class. These object types are similar to Java's interfaces in that distinct classes can generate objects of the same object type.

$\mathcal{LOOM}$ uses the reserved word `self` in the bodies of methods to refer to the object executing the method, and the reserved word `MyType` to refer to the type of `self`. The language Trellis/Owl [23] also includes a `MyType` construct, while Eiffel expresses it with the `like Current` construct. Because the meaning of `self` changes when inheriting code from a class to a subclass, it is useful to have the name `MyType` for its type, as it also changes meaning when moving to a subclass.

The following is an extract from a $\mathcal{LOOM}$ program[2]:

```
class Node
   var
      val = 0: Int;
      next = nil: MyType
   methods
      fun getVal(): Int {return val}
      proc setVal(nuVal: Int) {val := nuVal}
      fun getNext(): MyType {return next}
      proc setNext(nuNext: MyType){next := nuNext}
end;

NodeType = ObjectType
   {getVal: Func(): Int;
    setVal: Proc(Int);
    getNext: Func(): MyType;
    setNext: Proc(MyType)}
```

`NodeType` is the type of objects generated by the class `Node`. Thus the evaluation of the expression `new Node` results in the creation of an object of type `NodeType`. The instance variables of a class are hidden by default, while methods are visible by default. Methods may be declared to be `HIDDEN` which is similar to the `protected` declaration in Java and C++.

Since `MyType` represents the type of the object executing the method, it follows that if `n:NodeType` then `n.getNext()` should return a value of type `NodeType`. To be well-typed, the parameter `newNext` in `n.setnext(newNext)` should also have type `NodeType`. The typing rule for message sends will imply that if a method `m` has signature `S` in an object type `T`, and `obj` is an object of type `T`, then the type of `obj.m` is `S[T/MyType]`, the type obtained by replacing all free occurrences of `MyType` in `S` by `T`.

The value of having `MyType` is exhibited in the definition of subclasses:

[2] We have made minor changes to the syntax to make the code more compact

```
class DbleNode inherit Node modifying setNext
    var
       prev = nil: MyType
    methods
       fun getPrev(): MyType {return prev}
       proc setPrev(nuPrev: MyType) {prev := nuPrev}
       proc setNext(nuNext: MyType) {super.setNext(nuNext);
                                      nuNext.setPrev(self)}
end;

DbleNodeType = ObjectType include NodeType
   {getPrev: Func(): MyType;
    setPrev: Proc(MyType);}
```

In the above subclass definition we add a new instance variable and two methods as well as modify `setNext`. The term `super.setNext(nuNext)` indicates that the `setNext` method from the superclass (`Node`) should be executed. If `dn` has type `DbleNodeType` then, by the rules explained above, `dn.getNext()` has type `DbleNodeType`, and `dn.setNext(newNext)` takes a parameter of type `DbleNodeType`, exactly as desired. This would not be possible in a subclass if `NodeType` were used in place of the type `MyType` in declarations in `Node` and `NodeType`.

We say one object type *matches* another, written `S` $<\#$ `T`, iff every method in `T` also occurs in `S` with exactly the same signature as in `T`. Thus `S` is a width-only extension of `T`.[3] Because the matching relation is only defined on object types and not classes, it is technically independent of inheritance. However, if `C` is a class generating objects of type `CType`, and `SC` is a subclass of `C` generating objects of type `SCType`, then `SCType` $<\#$ `CType`.

The matching relation is weaker than subtyping. For example, `DbleNodeType` $<\#$ `NodeType`, but they are not subtypes. The covariant change to the type of the parameter to `setNext` destroys the subtype relation. We discuss subtyping, matching, and binary methods in more detail in section 8. The next section introduces the generalization of `MyType` to virtual types.

4 Type-safe virtual types

In the introduction we showed how virtual types could be used to model the Subject-Observer pattern, and further showed how they facilitate the definition of specializations of those classes to get `WindowSubject` and `WindowObserver`. However, we also showed that blindly using such specializations can result in

[3] It would be possible to also allow depth extensions of `T`, but our experience showed it was rarely necessary in the presence of `MyType` and parametric polymorphism. Thus we omit it in order to keep the language simpler.

type insecurities.

In this section we discuss an extension to $\mathcal{LOOM}$ which supports a construct very similar to virtual types, yet is statically type-safe. The key to this construct is a relatively simple generalization of $\mathcal{LOOM}$'s `MyType`.

We begin by introducing a keyword, `TypeGroup`, which can be used together with generalizations of the `MyType` notation to group together mutually recursive type definitions. It is used as follows:

```
SubjObservGrpTp = TypeGroup

  ObjectType (MyObserver)
    notify: proc(MySubject, MyEvent)
      ...
  end;

  ObjectType (MySubject)
    notifyObservers: proc(MyEvent)
      ...
  end;

  ObjectType (MyEvent) with EventType;
    -- EventType is defined externally

end; -- TypeGroup
```

`SubjObservGrpTp` packages together three mutually recursive types, which will be referred to inside the group as `MyObserver`, `MySubject`, and `MyEvent`. The names can be chosen arbitrarily, but were chosen here to suggest the analogy with `MyType`. The type group delimits the scope of the included names. Each type name is visible inside the other type definitions, so the definition of `MyObserver` uses both `MySubject` and `MyEvent` in the parameter type. We can refer to these types externally by appending the internal name with the name of the type group, *e.g.*, `SubjObservGrpTp.MyObserver`.

Classes which generate objects of these types can be defined individually or as part of a class group. We define a single class as follows:

```
class ObserverClass generates SubjObservGrpTp.MyObserver
      notify = procedure (subj:MySubject; e:MyEvent)...
          ...
end;
```

Normally, however, these classes corresponding to each of the types in the type group will be defined at the same time. In this case it is usually easier to define them with a class group as follows.

```
classgroup SubjObservGrp generates SubjObservGrpTp

   class ObserverClass generates MyObserver
         notify = procedure (subj:MySubject; e:MyEvent)...
              ...
   end;

   class SubjectClass generates MySubject
      var
         observers: arraytype[MyObserver];
              ...
      methods
         procedure notifyObservers(e:MyEvent)...
              ...
   end; -- class

   class EventClass generates MyEvent inherit OldEventClass...
end; -- classgroup
```

A class group declaration indicates for which type group the classes generate objects. Individual classes indicate for which type in the type class they generate objects. Thus it follows from the declarations in the class group that executing `new SubjObservGrp.ObserverClass` will result in an object of type `SubjObservGrpTp.MyObserver`. It causes no difficulty to have more than one class in a class group which generates objects of the same type. Thus we may have two different classes generating objects of type `MyObserver` (though this may be unusual).

Like `MyType`, the power of this notation only becomes clear in the definition of subclasses. We first specify the new type group:

```
WindowSubjObservGrpType = TypeGroup extends SubjObservGrpType

   ObjectType (MyObserver) with
      ... // new method declarations

   ObjectType (MySubject) with
      ... // new method declarations.

   ... // possible new types

   ObjectType (MyEvent) with WindowEventType; -- externally defined
end;
```

The idea is that all of the types change in parallel in each of the types in the type group. Just as the meaning of `MyType` changed in extensions of types and classes, the meanings of the internal names in the type group change in extensions. Because we wish several names to change in unison, we group them together in a type group.

Types with the same name as the type group being extended must match those of the "super" type group. This can be done automatically by using `with` clauses to add new declarations, or the types may be defined independently from the types in the super type group (as with the definition of `MyEvent` above). If a name is not repeated in the extended group then its definition is just "inherited" unchanged. It is also possible to add new types to a type group.

The matching relation for virtual types is defined on the entire type group rather than on individual types in the group. Type group `TG'` $<\!\#$ `TG` iff for each type `MT` in `TG`, there is a corresponding type with the same name in `TG'` such that every method in `TG.MT` also occurs in `TG'.MT` with exactly the same signature as in `TG.MT`. (Essentially, the types pointwise match.) Clearly a type group defined by extending another matches the original, but type groups may match even if one is not defined as an extension of the other.

Extending single classes implementing a type from a type group is straightforward. The following illustrates that class groups may be extended in a fashion similar to that of type groups.

```
ClassGroup WindowSubObsGrp generates WindowSubjObservGrpType
        inherit SubjObservGrp
    modifying ObserverClass, SubjectClass, EventClass

   class ObserverClass generates MyObserver
          inherit SubjObservGroup.ObserverClass
      ...    -- modified and new variables and methods
   ...  -- redefinitions of SubjectClass and EventClass
   ...  -- possible new classes
end;
```

When one ClassGroup inherits from another, all classes from the original are inherited, though one may override inherited classes by listing them in the `modifying` clause. These overridden classes can be defined to inherit from those in the "super" class group or may be defined from scratch as long as they generate the appropriate type. While we discuss type checking in more detail later, we note that if a type is extended in a "sub" type group, then the corresponding class in the inheriting ClassGroup will also have to be modified so that it generates objects of the new type. Otherwise it will generate a type error.

These constructs are very straightforward to use and, when type-checked with the appropriate rules, provide type-safe inheritance of classes whose types are mutually recursive. In the next section we discuss the semantics of `MyType` and the version of virtual types introduced in this section.

5 The semantics of `MyType` and virtual types

Bruce, Fiech, and Petersen [4] presented an operational semantics for $\mathcal{LOOM}$, and used that to prove the type safety of the language. In order to better motivate the type-checking rules for our constructs, we instead present a translation of the key constructs of $\mathcal{LOOM}$ into an extension of $F_{\leq}$, the higher order polymorphic lambda calculus with subtyping. We will then generalize this semantics in order to derive type-checking rules for the extension of $\mathcal{LOOM}$ with virtual types introduced in the last section.

The translation presented here extends the ORE semantics presented in [3] by supporting an imperative rather than a purely functional language, and by including a translation for classes as well as objects. Our translation will be informal since we have not actually defined a formal source language. As a result we will treat constructs that are similar to those found in object-oriented languages as abbreviations for terms of $F_{\leq}$.

5.1 Object types and `MyType`

To get started we first focus on defining object types in a language without `MyType`. Let `IV` be the type of the record of instance variables of an object and `Meth` be the type of the record of methods. We will treat methods as being parameterized by their record of instance variables. Objects are represented by pairs of a record of (updatable) instance variables and of the parameterized methods:

$$\mathtt{IV} \ \times \ (\mathtt{IV} \ \rightarrow \ \mathtt{Meth})$$

Alternatively, we could move the `IV` parameter inside the record of methods so that we have $\{\mathtt{...,mi:IV} \rightarrow \mathtt{Ti,...}\}$ (and in fact, our implementation works that way). The two are isomorphic, so we choose the above presentation for notational simplicity.

There are two problems left to solve:

(i) How can we hide the instance variables from outside the object?

(ii) How can we determine the meaning of `MyType`?

We can solve the first problem by using an existential quantifier to hide the record of instance variables, writing the type as

$$\exists\ \mathtt{Y}.\ \mathtt{Y} \ \times \ (\mathtt{Y} \ \rightarrow \ \mathtt{Meth})$$

From the outside we see that objects are pairs in which the second component may be applied to the first, but we have hidden the type of the first component.

To handle MyType we need to generalize the types of instance variables and methods to be parameterized by a type variable that will end up being interpreted as MyType.

Let I, M: Type $\rightarrow$ Type represent the types of functions that, when provided with the meaning of MyType, return the types of the records of instance variables and methods of an object. For example, objects generated by class Node in section 3 are represented by

```
NodeI(MT) = {val: Int; next: MT}
NodeM(MT) = {getVal:...; setVal:...; getNext:Func():MT;
             setNext: Proc(MT)}
```

We can now determine the type of MyType by defining it recursively to stand for the entire type. We let Obj(M) stand for the type of objects with method suite described by M.

```
Obj(M) = Rec(X) ∃Y. Y × (Y → M(X))
```

Because of the recursive definition, we can see that the meaning of MyType is Obj(M). If M' extends M then Obj(M') <# Obj(M), but it follows from the subtyping rules for recursive types [1] that Obj(M') <: Obj(M) fails unless X occurs only positively in M.

Once we are provided with the interfaces of the instance variables for a class which generates objects of the types, we can determine the type of these objects as seen from the inside of the class. We use the notation VisObj(I,M) for the type of objects with the instance variables visible.

```
VisObj(I,M) = I(Obj(M)) × (I(Obj(M)) → M(Obj(M)))
```

In this definition, we have replaced all occurrences of MyType by Obj(M).

We now have definitions of the type of an object from both the inside and the outside. We can convert from the inside view to the outside using close: VisObj(I,M) $\rightarrow$ Obj(M), defined as:

```
close(I,M)(o: VisObj(I,M)) = pack [I(Obj(M)),o] as Obj(M)
```

The close operation packs o so that appropriate occurrences of I(Obj(M)) in its type are replaced by an existentially bound variable to obtain an object of type Obj(M). Thus the two occurrences of I(Obj(M)) in VisObj(I,M) (for the type of instance variables and the parameter type of methods) are replaced by the existentially-bound type variable.

Messages are sent to objects by applying the second (method) component of the object to the first (instance variable) component and then extracting the desired method. The only complication is that objects have been "packed" in order to hide their representation. However, the "open" operation on existential types opens up the packaged object in such a way that it may be manipulated even without knowing the precise value of the hidden type.

Let `o: Obj(M)` then

```
o.m = open o as [IV,⟨ inst,methfun ⟩] in methfun(inst).m
```

In this notation, `IV` is the hidden type of instance variables, while `inst` and `methfun` are the record of instance variables and the function taking the record of instance variables to the record of methods. Once the second component has been applied to the first, the appropriate method is extracted from the record of methods.

Because the method body has access to the instance variables of `self`, there is no difficulty in evaluating or updating these variables. Message sends to `self` (or indeed any element of type `VisObj(I,M)` are handled similarly to message sends to object types except that they do not need to be opened before applying the methods to the instance variables.

Typing restrictions on existential types [20] will not allow methods which return something of type `VisObj(I,M)` to type check. Thus if we have a method returning `self`, for example, we must apply `close` to `self` before returning it. An object of type `Obj(M)` is then returned, which is legal.

5.2 Virtual types

The semantics of type groups is very similar to that of types with `MyType`. The main modification is that virtual types require mutually recursive types rather than a single recursion.

For notational simplicity, we assume that the type group defines two object types with interfaces `M1` and `M2`. Rather than being parameterized by `MyType`, each is parameterized by the pair consisting of the internal names `X1` and `X2` of both types being defined. We define these object types via mutual recursion:

```
Obj(M1, M2) = Rec(⟨ X1,X2 ⟩).⟨ ∃Y1.Y1 × (Y1 → M1(X1,X2)),
                               ∃Y2.Y2 × (Y2 → M2(X1,X2)) ⟩
```

The first object type is the first component of `Obj(M1, M2)`, while the other is obtained by taking the second component of the pair. Each depends on the other, as expected. Type groups with k types can be interpreted similarly using k-tuples.

For example, `SubjObservGrpTp` is interpreted by defining method interfaces `Obs(MyObserver,MySubject,MyEvent)`, `Sub(MyObserver,MySubject, MyEvent)`, and `Evt(MyObserver,MySubject,MyEvent)` (where `Evt` doesn't actually depend on `MyObserver` or `MySubject`), and then defining `SubjObservGrpTp` = `Obj(Obs,Sub,Evt)`.

We can interpret the internal view of the types of objects similarly. Let `OM` abbreviate `Obj(M1,M2)`. The type of instance variables of a class may depend on all types in the corresponding type group. Thus we will also write the interfaces of instance variables for each class in the group as depending on the tuple of all object types, OM. Because OM is already defined, the internal views of objects may be defined individually. The following is the internal type of objects generated by the class whose instance variable types are given by Ii and methods are given by Mi:

$$\texttt{VisObj(Ii,OM) = Ii(OM)} \times \texttt{(Ii(OM)} \to \texttt{Mi(OM))}$$

The close operation is defined similarly to that for objects with MyType.

$$\texttt{close}(\texttt{I}_i, \texttt{OM})(\texttt{o} : \texttt{VisObj}(\texttt{I}_i, \texttt{OM})) = \texttt{pack}[\texttt{I}_i(\texttt{Obj(OM)}), \texttt{o}] \ \texttt{as} \ (\texttt{Obj(OM)})_i$$

Message sending is interpreted exactly as for the simpler language with only MyType.

6 Classes

Because one of our goals is to write down type-checking rules for method definitions in classes, we also need to understand how classes are encoded in a higher-ordered polymorphic lambda calculus with subtyping.

As we define classes, we must be aware that methods from classes can be inherited in subclasses. Thus method definitions not only must make sense for objects generated from the class being defined, but also for all possible subclasses. We can be assured of this if the classes are parameterized by the possible types of the instance variables and methods in subclasses.

6.1 Encoding classes

Let J and J' be functions from types to types. We write J <: J' iff for all types, T, J(T) <: J'(T). In order to ensure type correctness of inherited methods, we will require that the method and instance variable interfaces of subclasses be subtypes of the corresponding interfaces of their superclasses.

There are several important differences between classes and objects (and correspondingly their types).

(i) Classes provide only initial values of instance variables, while objects hold the actual variables (references).

(ii) Methods in classes are parameterized by self, which represents both the methods and instance variables of the object executing the code. Methods in objects are parameterized only by the instance variables.

(iii) Methods in classes should remain type correct when inherited in subclasses.

We define

$$\begin{aligned}\texttt{Class(I,M)} = \forall \texttt{M'} <: \texttt{M}.\ \forall \texttt{I'} <: \texttt{I}^{\texttt{ref}}.\\ \texttt{I(Obj(M'))} \times \texttt{(VisObj(I',M')} \to \texttt{M(Obj(M')))}\end{aligned}$$

Thus if $\texttt{meth: } \forall\texttt{M'} <: \texttt{ M.}\forall\texttt{I'} <: \texttt{ I}^{\texttt{ref}}\texttt{.(VisObj(I',M')}\rightarrow\texttt{M(Obj(M')))}$ and $\texttt{inst: }\forall\texttt{M'}<:\texttt{M.I(Obj(M'))}$, we can define a class with methods given by `meth` and initial values of instance variables given by `inst` as

$$\texttt{class(inst,meth) = }\lambda\texttt{M'}<:\texttt{M. }\lambda\texttt{I'}<:\texttt{I}^{\texttt{ref}}\texttt{.<inst(M'), meth(M',I')>}$$

which has type `Class(I,M)`.

By part (i) above, the class provides an initial value $\texttt{v}_i$ for each field, $\texttt{x}_i$ of the record of instance variables, while the object itself contains a variable (or reference) of that type. The notation $\texttt{I}^{\text{ref}}$ indicates the corresponding record of references; if $\texttt{I(T)} = \{\ldots;\texttt{x}_i : \texttt{S}_i;\ldots\}$ then $\texttt{I}^{\text{ref}}\texttt{(T)} = \{\ldots;\texttt{x}_i : \texttt{ref S}_i;\ldots\}$. Thus `I(T)` represents the type of the initial values for instance *values*, while $\texttt{I}^{\text{ref}}\texttt{(T)}$ is the type of the record of instance *variables* (references). When we create new objects we will take the initial values provided by the class and build a record of variables initialized to those values.

By part (ii), all methods take a parameter of type `VisObj(I',M')`, which represents the type of `self` in an object generated by some subclass of the one being defined. Message sends must explicitly go through a `self` parameter because methods may be overridden in subclasses. If the definition of a message send of `m` to `self` in the body of some other method `n` were compiled in to the class definition, execution of the body of `n` would always use the original definition of `m` rather then the updated version.

To accommodate part (iii), classes are parameterized by all extensions of the types of the record of instance variables and methods. Because any subclass will have records of instance variables and methods whose types are subtypes of those of the original class, we should be able to instantiate the methods for those types.

Defining subclasses by adding new instance variables and methods, and by overriding existing methods, is pretty straightforward. If the superclass `c = class(inst,meth)` has type `Class(I,M)` then the type of any subclass will be of the form `Class(IS,MS)` for $\texttt{IS}^{\text{ref}} <: \texttt{I}^{\text{ref}}$ and $\texttt{MS} <: \texttt{M}$.[4]

Generalizing the type and class constructions for our richer language with virtual types is straightforward. While classes may be conveniently defined together in a class group, the classes themselves are generally not mutually recursive. They also may be defined individually, as long as we take care to handle the types occurring in them properly. Again, for notational simplicity we presume that the typegroup only contains two types. The type of the ith class is given by

$$\texttt{Class}(\texttt{I}_i, \texttt{M}_1, \texttt{M}_2) = \forall\texttt{M'} <: \langle\texttt{M}_1, \texttt{M}_2\rangle.\forall\texttt{I}'_i <: \texttt{I}_i{}^{\text{ref}}.$$

[4] We add the `ref` superscript here in case subtyping includes depth subtyping. Because the objects include instance variables rather than simply instance values, we need the subtyping which works with the variables. Subtyping for records of variables only includes record extension. As a result, changes in types of individual instance variables are not allowed.

$$\texttt{I}_i\texttt{(Obj(M'))} \times \texttt{(VisObj(I}_i'\texttt{,M')} \rightarrow \texttt{M}_i\texttt{(Obj(M'))}$$

The actual class and subclass definitions are as for the case with `MyType` except for the more general type parameters.

6.2 Creating objects

Creating objects from classes is a bit tricky because methods in classes are parameterized by `self`, which includes both methods and instance variables, while methods in objects are parameterized only by the instance variables. Because of space restrictions we provide only a brief sketch here. The process is identical for both the `MyType` language and virtual types.

Let `meth`: $\forall$`M'` $<:$ `M`.$\forall$`I'` $<:$ $\texttt{I}^{\texttt{ref}}$`.(VisObj(I',M')`$\rightarrow$`M(Obj(M')))` and `inst`: $\forall$`M'` $<:$ `M.I(Obj(M'))`, and

$$\texttt{class(inst,meth) = } \lambda\texttt{M'} <: \texttt{M. } \lambda\texttt{I'} <: \texttt{I}^{\texttt{ref}}\texttt{.<inst(M'), meth(M',I')>}$$

We can create a new object from class `c = class(inst,meth)` as follows. Specialize the polymorphic instance variables and methods to the types of the method suite and instance variables generated by the class

$$\texttt{inst' = (inst(M))}^{\texttt{ref}}\texttt{: I}^{\texttt{ref}}\texttt{(Obj(M))}$$
$$\texttt{meth' = meth(M,I}^{\texttt{ref}}\texttt{):VisObj(I}^{\texttt{ref}}\texttt{,M)} \rightarrow \texttt{M(Obj(M))}$$

The notation $\texttt{inst}^{\mathrm{ref}}$ indicates a record of *variables* whose initial values are given by the corresponding record of *values*, `inst`.

In order to adapt the method suite from the class to fit the object, we must "build-in" the methods, and have the methods depend only on the record of instance variables:

$$\begin{aligned}\texttt{methfun = } & \lambda\texttt{w:I}^{\texttt{ref}}\texttt{(Obj(M)).} \\ & \texttt{meth'(}\langle\texttt{ w,methfun }\rangle\texttt{):I}^{\texttt{ref}}\texttt{(Obj(M))} \rightarrow \texttt{M(Obj(M))}\end{aligned}$$

Thus `methfun` is obtained from `meth'` by setting `self` to be the pair formed with the provided record of instance variables and `methfun` itself as the method suite.

Finally define:

$$\texttt{new class(inst,meth) = close(I,M) } \langle\texttt{ inst',methfun }\rangle\texttt{: Obj(M)}$$

Each time the `new` command is executed, a new record of instance variables, `inst'`, is created, whereas the method suite `methfun` may be shared.

Constructing new objects from classes with virtual types is similar. The only substantial difference is that `meth'` is obtained by applying `meth` to the pair of $\texttt{M}_1$ and $\texttt{M}_2$ as well as $\texttt{I}_i$. The construction of `methfun` and the final object is exactly as with `MyType`.

7 Deriving type checking rules from the semantics

The definition of the meaning of classes given in the previous sections provides guidance for type-checking classes in our high-level language. Recall that in the definition of `class(inst,meth)` we required `inst`: $\forall$`M'` $<:$ `M.I(Obj(M'))` and `meth`: $\forall$`M'` $<:$ `M.`$\forall$`I'` $<:$ `I`$^{\texttt{ref}}$`.(VisObj(I',M')`$\rightarrow$`M(Obj(M')))`.

We could design a realistic source language by omitting the type function parameters `M'` and `I'`, writing `MyType` for `Obj(M')` and `SelfType` for `VisObj(I',M')`. The keyword `self` has type `SelfType` and the function `close` takes elements of type `SelfType` to `MyType`. The constraint that `M'` $<:$ `M` would translate into `MyType` $<\!\#$ `Obj(M)`

Using this terminology, we can write the type-checking rules for the language. Type-checking assertions are written in the form $C, E \vdash \texttt{e:T}$, where C is a collection of subtyping and matching constraints on type variables, and E is a collection of assumption about the types of term variables (see [4] for details). The following type-checking rule results directly from the terminology introduced above:

$$\frac{C^{IV}, E \vdash \texttt{inst:I(MyType)}, \quad C^{METH}, E^{METH} \vdash \texttt{meth:M(MyType)}}{C, E \vdash \texttt{class}(\texttt{inst}, \texttt{meth}) : \texttt{Class(I,M)}}$$

where

- $C^{IV} = C \cup \{\texttt{MyType} <\!\# \, \texttt{Obj(M)}\}$,
- $C^{METH} = C^{IV} \cup \{\texttt{SelfType} <\!\# \, \texttt{VisObj}(I, M)\}$,
- $E^{METH} = E \cup \{\texttt{self} : \texttt{SelfType}, \texttt{close:Func}(\texttt{SelfType}) : \texttt{MyType}\}$, and
- neither `MyType` nor `SelfType` occurs free in C or E.

A similar, but slightly more complex rule can be given for subclasses.

The rule for type-checking classes with virtual types is similar, and can again be read off from the definition of classes. We provide the definition for the case in which the type group, `TGP`, contains only two types, with method interfaces $\texttt{M}_1$ and $\texttt{M}_2$. As above, we write $\texttt{MyType}_i$ for $(\texttt{Obj(M'}_1\texttt{,M'}_2\texttt{)})_i$. We also write $\texttt{TGP.MyType}_i$ for the types $(\texttt{Obj(M}_1\texttt{,M}_2\texttt{)})_i$ defined in `TGP`. We write the type-checking rule for a class with instance variable interface $\texttt{I}_1$ and method suite $\texttt{M}_1$ which implements $\texttt{TGP.MT}_1$. As above, write $\texttt{SelfType}_1$ for $\texttt{VisObj(I'}_1\texttt{,Obj(M'}_1\texttt{,M'}_2\texttt{))}$.

$$\frac{\begin{array}{c} C^{IV}, E \vdash \texttt{inst}_1 : \texttt{I}_1(\texttt{MyType}_1, \texttt{MyType}_2), \\ C^{METH}, E^{METH} \vdash \texttt{meth}_1 : \texttt{M}_1(\texttt{MyType}_1, \texttt{MyType}_2) \end{array}}{C, E \vdash \texttt{class generates TGP.MyType}_1(\texttt{inst}_1, \texttt{meth}_1) : \texttt{Class}(\texttt{I}_1, \texttt{M}_1, \texttt{M}_2)}$$

where

- $C^{IV} = C \cup \{\texttt{MyType}_1 <\!\# \, \texttt{TGP.MyType}_1, \texttt{MyType}_2 <\!\# \, \texttt{TGP.MyType}_2\}$,
- $C^{METH} = C^{IV} \cup \{\texttt{SelfType}_1 <\!\# \, \texttt{VisObj}(I_1, M_1, M_2)\}$,

- $E^{METH} = E \cup \{\texttt{self} : \texttt{SelfType}_1, \texttt{close:Func(SelfType}_1\texttt{)} : \texttt{MyType}_1\}$, and
- none of $\texttt{MyType}_1$, $\texttt{MyType}_2$, or $\texttt{SelfType}_1$ occurs free in C or E.

Class groups can be treated as a notational abbreviation for a series of class definitions as above. The translation is straightforward.

The rules for subclassing are tedious to write down, but not particularly surprising. Because of the way we have type checked methods, inherited methods need not be type checked in subclasses. For class groups, one must ensure that the type of any class inherited (unchanged) also not have changed from the type group implemented by the superclass group.[5]

The type checking rule for message sends for the language with `MyType` also follows directly from our translation. If object `o` has type `T = Obj(M)`, for some `M`, then by the recursive definition

```
T = Obj(M) = Rec(X) ∃Y. Y × (Y → M(X))
  = ∃Y. Y × (Y → M(T))
```

the meaning of `MyType` in the types of methods is `T`. Thus if method `m` of `o` has type `U` involving `MyType`, then from the definition

```
o.m = open o as [I',⟨ inst,methfun ⟩] in methfun(inst).m
```

we see that

$$\frac{C, E \vdash \texttt{o} : \texttt{ObjectType}\ \{\texttt{...,m:U,...}\}}{C, E \vdash \texttt{o.m} : \texttt{U[T/MyType]}}$$

We actually need a slightly more general form of this rule for type-checking classes in order to handle terms whose types are only known up to a match-bound (like `close(self)` with type `MyType`).

$$\frac{C, E \vdash \texttt{o} : \texttt{T'}, \quad C \vdash \texttt{T'} <\!\# \,\texttt{ObjectType}\{\texttt{m} : \texttt{U}\}}{C, E \vdash \texttt{o.m} : \texttt{U[T'/MyType]}}$$

This more general rule follows easily from the definition of matching.

For the more general case of virtual types, this rule generalizes to

$$\frac{C, E \vdash \texttt{o} : \texttt{TPG'.MT}_1, \quad C \vdash \texttt{TPG'} <\!\# \,\texttt{TPG}}{C, E \vdash \texttt{o.m} : \texttt{U[TPG'.MT}_1\texttt{/MyType}_1, \texttt{TPG'.MT}_2\texttt{/MyType}_2]}$$

for $\texttt{TPG.MT}_1 = \texttt{ObjectType}\{\ldots, \texttt{m} : \texttt{U}, \ldots\}$.

The use of this rule can be seen in the following code:

```
SubjObsGrpType.MyObserver obs = new ObserverClass;
SubjObsGrpType.MySubject subj = new SubjectClass;
...
obs.notify(subj);
```

[5] This is more straightforward with single classes. The complication occurs here only because of the compressed notation.

This message send is well-typed because, by the above rule, `obs.notify` takes a parameter of type `SubjObsGrpType.MySubject`.

In summary, the semantics provided here suggest the type-checking rules for our extension of $\mathcal{LOOM}$ with virtual types. These semantics may be used to give a careful proof of the type safety of these rules.

8 Other uses of virtual types

As mentioned in the introduction, another use of virtual types is to specialize types in a single type or class. The following example (written in slightly different notation) appears in both Thorup [24] and Igarashi and Pierce [12].

```
AnimalGrpTp = TypeGroup
   ObjectType (MyAnimalType){eat: Proc(MyFoodType);
                             clone: Func()MyAnimalType;...}
   ObjectType (MyFoodType) with FoodType
end;

HerbivoreGrpTp = TypeGroup extends AnimalGrpTp
   ObjectType (MyFoodType) with PlantType
end;
```

The type group `AnimalGrpTp` represents animals that can eat anything of type `FoodType`. There likely are not many animals that can successfully eat all kinds of food, so most animals created will actually be from classes which implement animals from an extension of the type group. Above we provided the example of an extension for herbivores. `CowGrpTp` could be defined by specializing `HerbivoreGrpTp` so that `FoodType` is specialized to exactly the kinds of food eaten by cows. Classes generating objects of these types can be written and type-checked as described earlier.

One of the uses of virtual classes in Beta that we have not yet described is forming collections of objects arising from specializations of the virtual classes. Thus one might wish to form a zoo whose representation contains an array of animals obtained from specializations of `AnimalGrpTp.MyAnimalType`.

In traditional object-oriented language, one may obtain such a heterogeneous collection by defining an array whose type is a supertype of all objects to be inserted. However, in this case, the different animal types will not be subtypes of `AnimalGrpTp.MyAnimalType` because of the covariant specialization of `MyFoodType`, which appears as the type of the method `eat` of `MyAnimalType`. Beta allows defining a collection in the way suggested, but inserts run-time checks in order to ensure that the failure of subtyping does not result in run-time type errors (*e.g.*, sending the message `lion.eat(gazelle)` when a `lion` only eats meat). We can create such collections, yet guarantee static type safety by generalizing the "hash types" introduced in $\mathcal{LOOM}$.

Our experience with the language PolyTOIL, which contains `MyType` and parametric polymorphism, led us to believe that subtyping played a much less important role than matching in such a language. As a result we dropped subtyping altogether in the design of $\mathcal{LOOM}$, replacing implicit subtyping by explicit hash types. If `o: T` then the run-time value held in `o` will always be of exactly type `T`.

If `T` is an object type, then we let `#T` (read "hash `T`") represent the type of all values whose type matches `T` (see [4] for more details). In terms of our encoding in terms of $F_{\leq}$, we can interpret `#T` as $\exists$`t` $<\#$ `T. t`. If a variable has type `#T`, then it can be assigned a value of any type matching `T`. Thus hash types can be used to represent heterogeneous collections. The main difference between this and subtypes is that one must explicitly indicate with a hash type when the type is supposed to represent a heterogeneous collection, whereas that is implicitly the case with subtyping.

An important advantage of hash types over subtypes is that they can be used for collections of elements whose types fail to be subtypes of a single type, but are in the matching relation. For example, a variable of type `#Node` can hold elements of both type `NodeType` and `DbleNodeType`. We generalize this notion of hash types for type groups.

If `TPG` is a type group with included type `T`, then `#TPG.T` represents the type of all values whose type is of the form `TPG'.T` where `TPG'` $<\#$ `TPG`. As with hash types involving `MyType`, we interpret `#TPG.T` as $\exists$`t` $<\#$ `TPG. t.T`. Thus `#AnimalGrpTp.MyAnimalType` includes animals from all type groups matching `AnimalGrpTp`.

Earlier we saw how to type-check message sends involving `MyType` or virtual types. These are a bit more complex with hash types because of the interpretation involving existentials. We begin by reviewing what happens with `MyType`. Let `o:#T` where `T` is of the form `ObjectType{...,m:U,...}`.

We interpret a message send to `o` as:

```
o.m = open o as [T' <# T,o'] in o'.m
```

The actual object `o'` packed in `o` is guaranteed to have some type `T'` $<\#$ `T`, so `o'.m` is well-typed and has type `U[T'/MyType]`. A problem with typing the entire `open` expression is that the type of `o'.m` involves `T'` in those places that originally involved `MyType`. Unfortunately the typing rules for `open` expressions will not allow the such an expression to have a type involving `T'`, since the `T'` would then escape the scope of the `open` expression.

One way of avoiding this problem would be to promote all occurrences of `T'` in the result type to `#T`. However, it is easy to find examples where blindly promoting types would result in a type insecurity. For example, let `NodeType` be as defined in section 3. If `n: #Nodetype`, and we promote occurrences of the hidden type to #NodeType in typing message sends then n.setNext would be typed as taking a parameter of type #NodeType. However if n actually held a value of type DbleNodeType, then providing a parameter whose actual type is NodeType (and hence is also considered to have type #NodeType) would result in a type error. The error occurs because the code executed would be the method body from DbleNode, which requires sending a message setPrev to the parameter, which doesn't have a corresponding method.

However, it is possible to prove that if all occurrences of MyType in U occur positively, then the promotion is well-typed. (See [4] for a proof using a different semantics.). In other words, if o:#T, where T is of the form ObjectType{...,m:U,...}, and all occurrences of MyType in U are positive, then o.m: U[#T/MyType]. The general rule is:

$$Msg\# \quad \frac{C, E \vdash \mathtt{o} : \#\mathtt{ObjectType}\{\ldots; \mathtt{m}_i : \mathtt{T}_i; \ldots\}}{C, E \vdash \mathtt{o} \Leftarrow \mathtt{m}_i : \mathtt{T}_i[\#\mathtt{ObjectType}\{\ldots; \mathtt{m}_i : \mathtt{T}_i; \ldots\}/\mathtt{MyType}]}$$

if all occurrences of MyType in $\mathtt{T}_i$ are positive.

For example, if n: #Nodetype, then n.getNext() has type #NodeType, while n.setNext(newNext) is not well-typed no matter what the type of newNext. While this may seem limiting, it should be clear that if one does not know the exact type of n, there is no way to ensure statically that the type of the parameter will match that of the receiver.

We can generalize this typing rule to virtual types as follows:

$$Msg\# \quad \frac{C, E \vdash \mathtt{o} : \#\mathtt{TGP.MT}_i}{C, E \vdash \mathtt{o} \Leftarrow \mathtt{m} : \mathtt{T}[\#\mathtt{TGP.MT}_1/\mathtt{MT}_1, \ldots, \#\mathtt{TGP.MT}_n/\mathtt{MT}_n]}$$

if type group TGP consists of types $\mathtt{MT}_1$ through $\mathtt{MT}_n$, if $\mathtt{TGP.MT}_i$ has a method m with type T, and if $\mathtt{MT}_1$ through $\mathtt{MT}_n$ occur only positively in T.

Thus if a: #AnimalGrpTp.MyAnimalType, then at run-time, a may hold an animal from any group type matching AnimalGrpTp, including HerbivoreGrpTp and CowGrpType. The expression a.clone() is well-typed and has static type #AnimalGrpTp.MyAnimalType, but a.eat(someFood) will never be well-typed. Thus we may clone all of the animals in a zoo without difficulty, but if we wish to feed them all we we need to introduce a type-case statement into the language. Such a construct would allow us to test whether the run-time value is of a particular type (e.g., CowGrpType.MyAnimalType), and then use it with that type in expressions.

By using both exact and hash types, we gain added expressiveness in our language by allowing the representation of either homogeneous or heterogeneous collections. With exact types we may type check any message send to an object, while with hash types we can only successfully type a message send if all of the types from the receiver's type group occurring in the message type are positive.

9 Conclusion

In this paper we have presented a language construct similar to the virtual classes of Beta, but which can be statically typed. An encoding of this construct in $F_{\leq}$ was given, and the type-checking rules resulting from the encoding were discussed.

One aim of this paper was to show that a deep understanding of the semantics of languages (especially object-oriented languages) could lead to the development of statically type-safe language constructs which could increase the expressiveness of programming languages. The work described here resulted from discussions on the java-genericity mailing list that were provoked by the paper of Thorup [24], which proposed to add virtual types to Java [2]. Many of us wished to find a statically type-safe construct that would allow similar expressiveness. After struggling to find such a construct, the first author re-examined the semantics of object-oriented languages as originally presented in Bruce [6]. The generalization from simple recursion to mutual recursion provided the guidance as to how to devise a construct which would be statically type-safe and yet have similar expressiveness.

The proposed construct was explained in Bruce, Odersky, and Wadler [5] as a possible extension of Java, though the semantics were not discussed in that paper. In the same paper it was shown how to simulate virtual types with F-bounded polymorphism, though in a very complex way. Torgersen [25] also has a proposal for restricting virtual types (essentially with mechanisms to restrict further changes to virtual types or to restrict the receiver to being of an exact type) to ensure static type safety.

More recently, Igarashi and Pierce [12] have shown how to model the simple uses of virtual types discussed in section 8 by using dependent records with bounded and manifest type fields. This work can be seen as a formal foundation for a purely functional version of Torgersen's proposal, explaining why that proposal is type-safe. The authors claim that it should be easy to extend this encoding to mutually recursive types. Surprisingly, it appears difficult to extend this encoding to an imperative version of the language. The difficulty seems to result from the combination of dependent types and imperative features.

Rémy and Vouillon [21] point out that virtual types pose few problems in Ocaml because of its combination of the use of row variables and type inference. (The row variables in Ocaml make it easy to define a construct like `MyType`.) Indeed, type inference can make many typing problems easier as the user does not have to think about writing the types in programs. In Ocaml, the Subject-Observer example can be written by having both the subject and observer classes take type parameters. The subject class takes type parameters for the observer and event classes, while the observer class takes type parameters for the subject and event classes. Constraints written similarly to bounded quantification provide guarantees that objects with types given by the type parameters have the appropriate methods. As with our proposal, the classes are not defined mutually recursively (though we allow this grouping for convenience), while the corresponding objects types are defined mutually recursively.

The main difference between the proposals is that our new language construct allows us to omit the extra type parameters in the class definitions. We believe this to be an important difference in practice as extra type parameters intended to be instantiated with recursively defined types make class definitions harder to read. A disadvantage of their encoding is that because of the lack of support for hash types, it is not possible to directly support heterogeneous collections of elements created from descendants of AnimalGpTp.MyAnimalType. In its defense, the authors point out that if a new supertype is created which drops the methods with virtual types as parameters, then heterogeneous collections of objects with that type are roughly equivalent.

The construct described here and in [5] is flexible enough that it should be relatively straightforward to add to any statically-typed object-oriented language. For example it could be adapted to be used in Eiffel in such a way as to close the static type holes caused by allowing covariant changes in the types of instance variables and parameters of methods. In essence it provides a mechanism similar to Eiffel's "anchor" types, but with sufficient restrictions to ensure static type safety. It is our belief that a construct similar to this would be less restrictive to programmers than Meyer's recent "No polymorphic cat-calls" proposal [17] to fix the Eiffel type system.

References

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

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

[3] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Proceedings of TACS '97 (Theoretical Aspects of Computer Science), Springer LNCS 1281*, pages 415–438, 1997.

[4] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[5] Kim Bruce, Marin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP '98*, pages 523–549. LNCS 1445, Springer-Verlag, 1998.

[6] 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.

[7] 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.

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

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

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and reuse of object-oriented designs.* Addison-Wesley, 1994.

[11] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[12] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. to appear in *Informal FOOL 6 electronic proceedings*, 1998. Technical report, University of Pennsylvania, 1998.

[13] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, 1987.

[14] Bent Brun Kristensen, Ole Lehrmann Madsen and Birger Moller-Pedersen and Kristen Nygaard. Abstraction mechanisms in the BETA programming language. In *Conference Proceedings of the 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 285–298, 1983.

[15] O. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89 Proceedings*, pages 397–406. ACM SIGPLAN Notices,24(10), October 1989.

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

[17] B. Meyer. Static typing and other mysteries of life. Technical report, Interactive Software Engineering, Inc., 1995. Text of invited address to OOPSLA '95.

[18] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, 1990.

[19] John C. Mitchell. *Foundations for Programming Languages.* Foundations of Computing Series. MIT Press, 1996.

[20] 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.

[21] Didier Rémy and Jérôme Vouillon. On the (un)reality of virtual types. *INRIA technical report*, 16 pages, 1998.

[22] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag LNCS 19, 1974.

[23] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, pages 9–16. ACM SIGPLAN Notices,21(11), November 1986.

[24] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97*, pages 444–471. LNCS 1241, Springer-Verlag, 1997.

[25] Mads Torgersen. Virtual types *are* statically safe. In *Informal FOOL 5 electronic proceedings.* http://pauillac.inria.fr/~remy/fool/abstracts/5.html, 1997.