7 Adding MyType to object-oriented languages

In the previous section we saw how useful Eiffel's `like Current` construct could be in helping programmer expressiveness. The main problem was that because it resulted in covariant changes to method parameter types and instance variables, it could lead to type insecurities.

In this section we go back to our encoding of classes to examine in more detail the problems with typing `self` first discussed in section 4.3. This will lead us to introduce a keyword, `MyType`, to stand for the type of `self`. It will be equivalent to Eiffel's `like Current`, but we will see how to statically type-check this construct in a way that guarantees type safety. The interesting point here is that understanding our previous encoding of objects and classes will lead us to a statically type-safe extension of object-oriented languages that will allow this style of programming. We will also be able to read off of the encoding the type-checking rules for this enriched language.

Let's take a brief side-track to discuss the use of class names in object oriented languages. Most familiar object-oriented languages use class names to stand for the names of classes, their constructors (e.g., as in `new C()`), and also as the types of objects generated by the class. We won't actually worry about constructors here, but we will take pains to distinguish between a class and the type of objects generated by the class. As a useful naming default, if `C` is the name of a class, then we will usually use `CType` to stand for the type of objects generated by the class. In our examples we will tend to use only interfaces as types in order to accentuate this difference.

Recall that our new translation of `C` = `class(inst,meth)` is

$$\Lambda(\mathtt{M'} <: \mathtt{M}).\ \Lambda(\mathtt{IV'} <: \mathtt{IV}^{\mathtt{ref}}).\ \langle \mathtt{inst},\ \lambda(\mathtt{self{:}VOT(IV',M')}).\mathtt{meth}\rangle$$

with type

$$\mathtt{ClassType(IV,M)} = \forall \mathtt{M'} <: \mathtt{M}.\ \forall \mathtt{IV'} <: \mathtt{IV}^{\mathtt{ref}}.\ \mathtt{IV} \times (\mathtt{VOT(IV',M')} \rightarrow \mathtt{M})$$

Using the notation suggested above for the types of objects generated by a class, the type of objects generated by `C` is `CType` = `OT(M)`.

In type-checking the method suite, `meth`, we assume that `self` has type `VOT(IV',M')` for $\mathtt{M'} <: \mathtt{M}$ and $\mathtt{IV'} <: \mathtt{IV}^{\mathtt{ref}}$. As we mentioned earlier, we can "close" `self` with `close[IV',M'](self)` to get an element of type `OT(M')`. In section 4.3 we noted that we could then use subsumption and the fact that $\mathtt{OT(M')} <: \mathtt{OT(M)} = \mathtt{CType}$ to deduce that `close[IV',M'](self)` has type `CType`.

Consider the following interface and class:

```
interface CType{
   returnMe():CType;
}

class C implements CType{
   x:int = 0;
   returnMe():CType {return self}
}
```

Because of the remarks above, the method `returnMe` in `C` can safely be given the return type of `CType`.

However, as we noted earlier, this results in the loss of typing information in subclasses. In particular if `D` is a subclass of `C` which inherits `returnMe` (i.e., does not override it), then `returnMe` still has return type `CType` rather than the more accurate `DType`. If we override `returnMe`, we are allowed to change the type in a covariant way (at least in some languages – we know it will be safe), but if we attempt to call `super.returnMe()` in the redefinition in `D`, we run into the same problems.

7.1 Introducing `MyType`

As we saw above, once we closed `self`, we obtained an element of type `OT(M')`. This is the most reasonable type to use for the return type of `returnMe`, as we don't want the outside world to be able to get access to the object's instance variables.

Unfortunately, the programmer writing a class does not have access to `OT(M')`, since `M'` only occurs in the translation of the class. Let's help out the programmer by creating a name for it. Since `OT(M')` is the best approximation for the type of `self`, let's give it the name `MyType`. Thus a programmer can write:

```
class C {
   x:int = 0;
   returnMe():MyType {return self}
}
```

Intuitively, if `cobj` is an object created from `C` (i.e., with type `CType`), then `cobj.returnMe()` will return an object of type `CType`.

However, the value of this shows more clearly when we define subclasses.

```
class D inherit C {
   ...
}
```

If `dobj` is an object created from `D` (i.e., with type `DType`), then `dobj.returnMe()` will return an object of type `DType`.

The keyword `MyType` will play the same role in our language as `like Current` does in Eiffel. For example, if we have a method `Clone` with signature `Clone():MyType` defined in the "top" class (e.g., class `Object` in Java), then it will be available in all subclasses and will always return an object whose static type is the same as the receiver.

Moreover, we can now write some of the same useful programs as we saw in Eiffel. Here is a simplified version of Eiffel's `Linkable` example:

```
class Node{
   item: int = 0;
   next: MyType = null;  -- Right neighbor

   setNext (other: MyType): void { next := other;}
   getNext (): MyType { return next;}
   setItem (newItem: int): void { item := newItem;}
   getItem ():int  { return item;}
}

class DbleNode inherit Node modifying setNext {
   previous: MyType = null;  -- Right neighbor

   setPrevious (other: MyType): void { previous := other;}
```

```
    getPrevious (): MyType { return previous;}
    setNext (other: MyType): void {
        next := other;
        if (other != null) then other.setPrevious(self);}
}
```

But what about type safety? We saw in the last section that assuming `DbleNodeType` <: `NodeType` resulted in a type error. We certainly don't want to have that happen to us. We will avoid that error by *not* assuming that subclasses always generate subtypes.

Let's take a closer look at the types of these classes and the objects that they generate. When we looked at the meanings of methods, we pulled `self` out as a parameter because it varied as we went from superclass to subclass. Similarly, we will pull out `MyType` from the types or interfaces of the instance variables and methods. Thus will write the interfaces of the instance variables and method suites of `Node` and `DbleNode` as parameterized types:

```
NodeIV(MT) = { item: int; next: MT}
NodeM(MT) =  { setNext: MT → void; getNext: void → MT;
               setItem: int → void; getItem: void → int }

DbleNodeIV(MT) = { item: int; next: MT; previous: MT}
DbleNodeM(MT) =  { setNext: MT → void; getNext: void → MT;
                   setPrevious: MT → void; getPrevious: void → MT;
                   setItem: int → void; getItem:void → int }
```

We can define subtyping between functions from types to types pointwise. That is, `F` <: `G` iff for all types `T`, `F(T)` <: `G(T)`. Looking at the definitions above, we see that $\texttt{DbleNodeIV}^{\texttt{ref}}$ <: $\texttt{NodeIV}^{\texttt{ref}}$ and `DbleNodeM` <: `NodeM`. In fact, if we maintain the restrictions on not allowing (explicit) changes to the types of instance variables, and only allowing method types to be replaced by subtypes in subclasses, we will always have these relations between the corresponding types of instance variables and of method suites.

We now must make sense of `MyType` in objects. This is relatively straightforward. If the object is generated by `Node`, and thus has type `NodeType`, the meaning of `MyType` in that object will just be `NodeType`. Similarly if an object has type `DbleNodeType`, then the meaning of `MyType` in that object will be `DbleNodeType`.

At this point you may be wondering why we don't just use `NodeType` *or* `DbleNodeType` *instead of* `MyType`*. Remember the reason is so that we end up with an accurate type of* `self` *that changes automatically in subclasses!*

Given the above, it shouldn't come as a surprise that the meaning of `MyType` will be determined by a fixed-point. We will use the notation μ`T. F(T)` for the fixed point of type function `F`. In particular, if `fixF` = μ`T. F(T)`, then `fixF` = `F(fixF)`. This is exactly the same equation satisfied by fixed points of regular functions. We won't bother to make up a special **Y** operator for type functions, but will just use the μ notation.

If `M(MyType)` is the type of the method suite of a class `C` and `IV(MyType)` is the type of its collection of initial values of instance variables, then we define:

```
OT(M) = μX. ∃Y. Y × (Y → M(X))
      = ∃Y. Y × (Y → M(OT(M)))
```

Notice that the meaning of `MyType` is set by the fixed point operator to be exactly `OT(M)`, as expected.

The types of objects "from the inside" are exactly as before, except that all occurrences of MyType are replaced by OT(M).

$$\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M) = IV}^{\texttt{ref}}\texttt{(OT(M))} \times \texttt{(IV}^{\texttt{ref}}\texttt{(OT(M))} \to \texttt{M(OT(M)))}$$

The meaning of MyType is exactly the same as in the definition of OT(M) (namely the meaning *is* OT(M)), the only thing that is different from the definition of OT(M) is that the instance variables (whose types may mention MyType) are visible.

Classes are defined similarly to before. The only difference is that both the instance variables and methods may involve MyType.

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

which has type:

$$\texttt{ClassType(IV,M) = } \forall\texttt{M'} <: \texttt{M. } \forall\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}\texttt{. IV(OT(M'))} \times \texttt{(VOT(IV',M')} \to \texttt{M(OT(M')))}$$

Notice that self is treated as having type VOT(IV',M'), for $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$. Therefore the type of close[IV'(OT(M')),M'(OT(M'))](self) is OT(M'). Thus in both instance variables and methods, MyType is interpreted as OT(M').

Subclasses are defined exactly as before. We might be concerned that methods inherited in subclasses need not be guaranteed to be type-safe, but the same argument as before shows that if the methods are type safe in the superclass they are guaranteed to be type safe in the subclass. Interestingly, the use of MyType has no impact on that argument.

In the object, the meaning of MyType will be OT(M), so we will apply both IV and M to OT(M) in order to get the final types of instance variables and methods. As before, we create a new object by building a new collection of instance variables, $\texttt{inst}^{\texttt{ref}}$, whose initial values are given by the class, and by defining a corresponding collection of methods which only take those instance variables as parameters. Let

$$\texttt{meth'(ir:IV}^{\texttt{ref}}\texttt{(OT(M))) = meth[}\langle\texttt{ir,meth'}\rangle\texttt{/self]:IV}^{\texttt{ref}}\texttt{(OT(M))} \to \texttt{M(OT(M))}$$

Then,

$$\texttt{new class(inst,meth) = close[IV}^{\texttt{ref}}\texttt{(OT(M)),M(OT(M))] } \langle\texttt{inst}^{\texttt{ref}}\texttt{,meth'}\rangle\texttt{: OT(M)}$$

We send messages exactly as before.

Thus adding a MyType construct to our language was straightforward. We simply parameterized our instance variables and methods by a type variable that we intend to eventually specialize to MyType. In classes, the MyType slot is filled with OT(M'), which represents possible values of MyType for subclasses. When we form an object, IV' gets specialized to $\texttt{IV}^{\texttt{ref}}$ and M' gets specialized to M, resulting in the meaning of MyType being fixed as OT(M). Thus the meaning of MyType in the object is exactly the type of the object, as desired.

Let's return to our example with the Node class. The type of the objects generated by Node is written NodeType, which we now know is defined as:

$$\begin{aligned}\texttt{NodeType} &= \texttt{OT(NodeM)}\\ &= \mu\texttt{X. } \exists\texttt{Y. Y} \times \texttt{(Y} \to \texttt{NodeM(X))}\\ &= \exists\texttt{Y. Y} \times \texttt{(Y} \to \texttt{NodeM(NodeType))}\end{aligned}$$

Now look at the type DbleNodeType of objects generated by the subclass DbleNode.

```
DbleNodeType = OT(DbleNodeM)
             = μX. ∃Y. Y × (Y → DbleNodeM(X))
             = ∃Y. Y × (Y →
             DbleNodeM(DbleNodeType))
```

Is `DbleNodeType` <: `NodeType`? It is easy to show using the subtyping rules for existentials, pairs, and functions that `∃Y. Y × (Y → MS')` <: `∃Y. Y × (Y → MS)` iff `MS'` <: `MS`. Therefore, `DbleNodeType` <: `NodeType` iff `DbleNodeM(DbleNodeType)` <: `NodeM(NodeType)`.

Because `DbleNodeM` <: `NodeM` as functions from types to types, for every field of `NodeM` there is a corresponding field of `DbleNodeM`. The problem is with the subtyping of individual fields (methods). Suppose, for the purposes of finding a contradiction, that `DbleNodeType` <: `NodeType`, and take a look at the type of the method `setNext` in each of the method suites. In the source language, in both classes the type of the method is `MyType` → `void`. In the encoding, this translates to `NodeType` → `void` in the `setNext` field of `NodeM(NodeType)` and `DbleNodeType` → `void` in the corresponding field of `DbleNodeM(DbleNodeType)`.

There is no difficulty with the range types of these function types since they are identical. But we need the domains of the types to be in a contravariant relation if the function types are to be subtypes. Thus we need `NodeType` <: `DbleNodeType`, but we assumed we had the opposite (and they are certainly not the same!). This contradiction shows that `DbleNodeType` is not a subtype of `NodeType`. A more direct proof can be given that essentially emulates the proof we gave that Eiffel is not type safe.

What went wrong here? The difficulty is that if `MyType` occurs in a negative or contravariant position (e.g., as the type of a subscript), then subtyping will fail. This failure is a result of the fact that the meaning of `MyType` implicitly changes when we create a subclass. Occurrences of `MyType` as return types cause no difficulty, only its use as a parameter type in methods.

Thus we have gained a more expressive type system at the cost of losing the guarantee that subclasses always generate subtypes. However, if we restrict the use of `MyType` to only positive positions, then we retain the guarantee that subclasses generate subtypes. The language Trellis/Owl, designed in the mid-1980's imposed exactly this restriction on the use of `MyType`.

7.2 The matching relation

We noted earlier that if `C'` with type `ClassType(IV',M')` is a subclass of `C` with type `ClassType(IV,M)`, then $\mathtt{IV'}^{\mathtt{ref}} <: \mathtt{IV}^{\mathtt{ref}}$ and `M'` <: `M`. The types of objects generated by the classes are `CType'` = `OT(M')` and `CType` = `OT(M)`. As we have just seen, if `OT(M)` includes a negative occurrence of `MyType`, then these will not be subtypes. However, we will find it useful to characterize their relationship by giving it a name.

We say `OT(M')` *matches* `OT(M)` (written `OT(M')` <# `OT(M)`) iff `M'` <: `M`. Thus two object types are in the matching relation if the smaller of the two has at least the methods of the larger, and corresponding method types are in the subtype relation when `MyType` is treated as an unknown type variable.

By contrast, `OT(M')` <# `OT(M)` iff `M'` <: `M` *and* `M` contains no use of `MyType` in a negative position, i.e., it has no occurrence of `MyType` as a parameter type. Thus if two object types are subtypes then they match, but not vice-versa.

By the definition of matching, if `C'` is a subclass of `C`, then their types must be in the matching relation. In particular, `DbleNodeType` <# `NodeType`. Thus, while the types of objects generated by subclasses need not be subtypes, they must be in the matching relation.

Matching provides information about what methods are available and their types. Let

```
Comparable = OT({equals, lessThan, greaterThan: MyType → boolean})
```

Then `Comparable` has no subtypes because of the occurrence of `MyType` as a parameter type. Suppose we know that `obj` has type `T`, where `T <# Comparable`. Then `obj` has methods including `equals`, `lessThan`, and `greaterThan`. We also know that all of these methods take parameters of type `T` (because that is the meaning of `MyType` for objects of type `T`), and return values of type `boolean`.

While matching provides information on what methods an object supports, it does not guarantee that objects of one type can "masquerade" as objects of another type. The stronger notion of subtyping is necessary in order to provide that assurance. Interestingly, much of the time only information on what methods can be sent to an object (and their types) is needed. This is especially the case for bounded polymorphism.

Let

```
class OrderedList[Elt <# Comparable]{
   insert(item:Elt):void {...}
   removeFirst():Elt{...}
   empty():boolean {...}
   search(o:Elt):int {...}
}
```

We give the name *match-bounded polymorphism* to this use of matching to constrain polymorphism.

Inside the methods of `OrderedList`, we are guaranteed that we can send the messages `equals`, `lessThan`, and `greaterThan` to objects of type `Elt`. This information about messages and their types is exactly what we needed to write this class. Moreover, this way of expressing the constraint is more straightforward and easier to understand than using F-bounded polymorphism.

In distinction to the use of F-bounded polymorphism, if the type of objects generated by class `C` matches a constraint, then objects generated by any subclass will also satisfy the same constraint. Let `C` be a class which generates objects of type `CType`, and suppose `CType <# BdType` for some object type, `BdType`. Let `D` be a subclass of `C`.

Then the type of objects generated from `D`, `DType`, matches `CType`. But `DType <# CType` and `CType <# BdType` implies that `DType <# BdType`. Thus if the type of objects generated by a class satisfies a match-bound, then the type of objects generated by any subclass will satisfy the same match-bound.

In summary, by including the type `MyType` we have lost the guarantee that subclasses always generate subtypes, but we have gained considerably in the expressiveness of the language. It is now easier to define subclasses (*e.g.*, the definition of `DbleNode` as a subclass of `Node`), and it is easier to express bounded polymorphism. When `MyType` occurs only in the return types of procedures, we retain all of the advantages of the simpler languages discussed in previous sections, including the fact that subclasses always generate subtypes.

7.3 Type-checking with `MyType`

In the previous subsection we noted that if the method suite of a class type-checked under our encoding, then it was guaranteed to be type-safe in all subclasses. However, our encoding used bounded type quantifiers representing the type interfaces of the instance variable and method suites. In the real world, programmers would not be happy having to translate their classes into that notation in order to understand whether or not their method bodies would type check. In this section we look for a way to explain the type-checking rules to programmers.

To make our work a bit easier, we will separate the uses of `self` for getting access to instance variables and as a target for methods. Thus we replace `self.x` by `selfinst.x`, if `x` is an instance variable. Similarly replace `self.m(...)` by `selfmeth.m(...)`, if `m` is a method. For simplicity, we'll also add `value` qualifiers to variables when we need their value.

For example, here is code for our earlier `Node` class where we've added a method `bump` to increase the contents of the `item` field by one in order to demonstrate how to handle sending a message to `self`.

```
class BNode{
   item: int = o;
   next: MyType = null;  -- Right neighbor

   setItem (newItem: int): void { selfinst.item := newItem;}
   getItem ():int  { return value selfinst.item;}
   setNext (other: MyType): void { selfinst.next := other;}
   getNext (): MyType { return value selfinst.next;}
   bump():void { selfmeth.setItem(selfmeth.getItem()+1);}
}
```

Its interface is

```
interface BNodeType{
   setItem (newItem: int): void;
   getItem ():int;
   setNext (other: MyType): void;
   getNext (): MyType;
   bump():void;
}
```

While this is annotated a bit more than one would normally find in a programming language, it would be easy for the compiler to add these annotations when parsing the class.

Encoding the class results in a term of the form:

$$\Lambda(\texttt{M'} <: \texttt{M}).\ \Lambda(\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}).\ \langle \texttt{inst},\ \lambda(\texttt{self:VOT(IV',M')}).\texttt{meth}\rangle$$

where

$$\texttt{VOT(IV',M')} = \texttt{IV'(OT(M'))} \times (\texttt{IV'(OT(M'))} \rightarrow \texttt{M'(OT(M'))})$$

Recall that occurrences of `OT(M')` in the encoding correspond to our use of `MyType`. We are given that $\texttt{M'} <: \texttt{M}$, so by the definition of matching, $\texttt{OT(M')} <\!\# \ \texttt{OT(M)}$. Thus we may assume that $\texttt{MyType} <\!\# \ \texttt{OT(M)}$ in type-checking `inst` and `meth`. In the case of type-checking the class `BNode` we may assume $\texttt{MyType} <\!\# \ \texttt{BNodeType}$

We use the fact that the type of `self` in the encoding is `VOT(IV',M')` to determine how to type-check the class. From the definition of `VOT(IV',M')`, we see that the type of `selfinst` is the first component of the meaning of the "visible" type, `IV'(OT(M'))`. Because $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$, and no depth subtyping is possible for reference types, we are guaranteed that if `x` is an instance variable declared in $\texttt{IV}^{\texttt{ref}}$, then its type will be the same in `IV'`. Thus in type-checking `BNode`, we can assume that `selfinst`.item has type int and `selfinst`.next has type `MyType`.

For message sends to `selfmeth`, we will actually be a bit sneaky. Rather than interpreting `selfmeth` as the method suite of `self` in type $\texttt{VOT(IV'}^{\texttt{ref}}\texttt{,M')}$, we instead interpret it as `close[IV'`$^{\texttt{ref}}$`,M'](self)`, which has type `(OT(M'))` = `MyType`. We can get away with this slight of hand because the results of sending a message to `close[IV'`$^{\texttt{ref}}$`,M'](self)` is exactly the same as sending it to `self`.

Exercise 7.1 *Check the encoding yourself, remembering the two definitions of message send, which depend on whether the type of the receiver was a "visible" object type or a "packed" object type.*

While it seems a bit strange to pack something up with `close` and then immediately open it up again when sending the message, the advantage is that we can now write down the type of `selfmeth` as `MyType`, something which makes sense to the programmer. (After all, even with this strange encoding, an implementation could optimize and just execute the simpler encoding.)

Thus we will type check the methods of `BNode` under the assumption that `selfmeth` has type `MyType` and that `MyType`matches `BNodeType`.

We can write the formal type-checking rule for classes as follows:

$$\textit{Class} \qquad \frac{\begin{array}{c} C \cup \{\texttt{MyType} <\!\!\# \, \texttt{OT}(M)\}, E \vdash \texttt{inst: IV(MyType)} \\ C \cup \{\texttt{MyType} <\!\!\# \, \texttt{OT}(M), \texttt{IT} <: \texttt{IV}^{\texttt{ref}}(\texttt{MyType})\}, \\ E \cup \{\texttt{selfinst: IT}, \texttt{selfmeth: MyType}\} \vdash \texttt{meth: M}(\texttt{MyType}) \end{array}}{C, E \vdash \texttt{class(inst,meth): ClassType}(IV, M)}$$

$$\textit{New} \qquad \frac{C, E \vdash \texttt{C: ClassType}(IV, M)}{C, E \vdash \texttt{new C: OT}(M)}$$

Those rules are a bit hard to read because of the formalism, but concisely summarize the type-checking rules discussed in this subsection.

Finally we present the type checking rule for message sends:

$$\textit{Msg} \qquad \frac{C \vdash \texttt{U} <\!\!\# \, \texttt{OT}(\{\texttt{m: T}\}) \qquad C, E \vdash \texttt{obj: U}}{C, E \vdash \texttt{obj.m: T}[\texttt{MyType} \mapsto \texttt{U}]}$$

Put more informally, this rule states that if `obj` has type `U`, where `U` $<\!\!\#$ `OT({m: T})`, then the type of `obj.m` is the type `T` of method `m`, where all free occurrences of `MyType` in `T` have been replaced by the type `U` of the receiver. In other words, when you send a message to an object, all occurrences of `MyType` in the method type are replaced by the static type of the receiver. This matches the intuition that `MyType` represents the type of the object executing the method.

Notice that while these type-checking rules mention `OT(M)`, they do not require an understanding of the encoding in order to do the type-checking. We have already discussed the assumptions used in type-checking the class `BNode`. Suppose `bn` is an object with static type `BNodeType`. Then `bn.getNext()` returns a value of type `BNodeType`, while `bn.setNext(newNext)` is well-typed only if `newNext` has type `BNodeType`.

Further, if `CNodeType` $<\!\!\#$ `BNodeType`, and `cn` has static type `CNodeType`, then `cn.getNext()` and `cs.setNext(newNext)` respectively return a value of type `CNodeType` and take a parameter of type `CNodeType`.

Exercise 7.2 *Fill in the missing instance variables and code for class* `OrderedList` *in section 7.2, and show that each of the initial values for instance variables and method bodies is well-typed.*

In this section we showed how to add the keyword `MyType`, representing the type of `self`, to an object-oriented language in order to gain expressiveness and yet retain type-safety. We also added the matching relation to the language, and discovered that match-bounded polymorphism is more useful than plain bounded polymorphism, and is more compatible with the subclass relation than F-bounded polymorphism.

The price paid for these advantages is that subclasses will not always generate subtypes, though they do generate matching types, and the language needs to support both matching and subtyping, two quite similar relations. The research language PolyTOIL [BSvG95] is very similar to the language described in this section.

In [BFP97], we went a step further and designed a language with `MyType`, matching, and match-bounded polymorphism, but without subtyping. In order to recapture some of the benefits of subtyping, the language introduces "hash types", written `#T`. If an expression has type `T` then at run time, the value of the expression will have exactly the type `T`. However, if it has type `#T`, then at run time its value may be of any type which matches `T`.[3] Thus for each variable, parameter, or return type whether we wish it to have an exact or a hash type.

The main disadvantage of using hash types is that if an expression has a hash type then we may not send a binary message to it. For example, suppose that `nd` has type `#NodeType`. It is sensible that `nd.getNext()` returns a value of type `#NodeType`. However, what should the type of `newNext` be in `nd.setNext(newNext)`, if it is to be well-typed? If the value of `nd` at run time is `NodeType`, then the parameter should also be of type `NodeType`, while if the value of `nd` at run time is `DbleNodeType`, then the parameter should be of type `DbleNodeType` (and a type error will occur if the parameter instead has type `NodeType`). Because binary methods require the parameter to have the same type as the receiver, we cannot determine what the correct type of the parameter should be if we do not know the exact type of the receiver.

As discussed in [BFP97], this restriction is not a problem in practice because the use of `MyType` does not make sense from a modelling point of view when working with heterogeneous data structures, and these heterogeneous data structures are the main reasons for the use of hash types.

While initial experience with LOOM has been very positive, greater use of the language will be necessary in order to evaluate its success in balancing type safety, expressiveness, and simplicity.

References

[BFP97] 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.

[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.

[3] `#T` can be understood as representing the bounded existential type, $\exists \mathtt{t} \lessdot\!\# \, \mathtt{T.\ t}$.