Understanding Object-Oriented Languages: Semantics and Types *(Continued)*

Kim B. Bruce
Princeton University & Williams College

December 14, 1998

3.3 Modeling subclasses

Now that we have presented our encoding of classes, we investigate how to encode subclasses. Before doing that, however, we'd best understand the legal changes to types of methods and instance variables that can be made in going from a superclass to a subclass.

3.3.1 Why restrict type changes in subclasses?

Why should there be any restrictions on redefining instance variables and methods in subclasses? The problem is that the type-safety of the methods of a class depends both on the types of the instance variables and on the types of the other methods.

Let class `C` have methods `m` and `n` as follows:

```
class C {
   ...
   m(): void {... self.n(arg) ...}
   n(farg:D): E {...}
}
```

In the above we assume that `arg` has type `D` and `self.n(arg)` occurs in a context which expects a value of type `E`.

Now suppose we define class `SC`:

```
class SC {
   n(farg:D'): E' {...}
```

Are there any restrictions on the changes we can make to the types of the arguments and return type of method `n` in the subclass?

Again, method `m` of the superclass uses `n` with an argument `arg` of type `D` in a context expecting a result of type `E`. Since `m` is inherited unchanged in `D`, when message `m` is sent to an object generated from `D`, it will execute the same code. In particular, it will generate a message send of `n` to `self`. However, the method `n` now has type `D'` $\rightarrow$ `E'`, rather than `D` $\rightarrow$ `E`.

If the application of `self.n(arg)` is still going to be legal, the new `n` must be applicable to arguments of type `D`. In other words, we must have `D` $<:$ `D'`. Similarly, the results of the new `n` must be usable in the same contexts as the old, so `E'` $<:$ `E`. Of course, this is just the rule we learned a while back for subtyping of function types: `D'` $\rightarrow$ `E'` $<:$ `D` $\rightarrow$ `E` iff `D` $<:$ `D'` and `E'` $<:$ `E`.

Thus, in defining subclasses, we may only change the return types of methods in a covariant way (e.g., by subtypes) and parameter types in a contravariant way. C++, for example, allows programmers to change the return types in methods in a covariant way. It is extremely difficult to come up with a compelling example of the use of contravariant changes in parameter types in subclasses. As a result, we know of no popular languages which allow parameter types to change.

Given the above reasoning, you will not be surprised that the rules for changing the types of instance variables in subclasses are exactly the same as for subtyping of variable or reference types. That is, no changes are allowed to the types of instance variables in going from a superclass to a subclass.

Exercise 3.1 *Write down an example similar to that above which shows that allowing covariant changes to the types of instance variables will result in type errors. That is, write a class that is well-typed, but a subclass which is not well-typed only due to the change to the type of an instance variable.*

Suppose class `C` has type `Class(IV,M)` and `C'` is a subclass with type `Class(IV',M')`. Recall that `IV` represents the type of the record of initial values of instance variables, while $\mathtt{IV}^{\mathtt{ref}}$ represents the type of a record of instance variables. Because `C'` is a subclass, `IV'` may add new instance variables, but may not change the types of existing instance variables. This is equivalent to saying that $\mathtt{IV'}^{\mathtt{ref}} <: \mathtt{IV}^{\mathtt{ref}}$, since reference types have no subtypes.

Similarly we may add new methods in `C'`, but overridden methods may only change return types in a covariant way and parameter types in a contravariant way. Thus $\mathtt{M'} <: \mathtt{M}$.

In summary, if class `C` has type `Class(IV,M)` and `C'` is a subclass with type `Class(IV',M')`, then $\mathtt{IV'}^{\mathtt{ref}} <: \mathtt{IV}^{\mathtt{ref}}$and $\mathtt{M'} <: \mathtt{M}$.

Notice that we have not *proved* that contravariant changes to parameter types and covariant changes to return types of methods in subclasses will result in subclasses free of type errors. We have only shown that other changes *can* result in type errors. Only when we have finished our encoding will we be able to make such an assertion.

3.3.2 A first try at encoding subclasses

We begin by using the simple encoding of the last section and seeing where we run into trouble. We use the notation $\mathtt{class\ inherit\ c\ (inst}_{sub}\mathtt{,meth}_{sub}\mathtt{)}$ to represent a subclass formed by inheriting the instance variables and methods from `c` and then adding and/or overriding with instance variables and methods from $\mathtt{inst}_{sub}$ and $\mathtt{meth}_{sub}$.

Suppose `c` = `class(inst,meth)`, and thus has encoding

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

To avoid type safety problems of the sort discussed in the previous section, we assume that if $\mathtt{inst}_{sub}$ redefines the initial value of an instance variable in `inst`, then they have the same type. If a method in `meth` is redefined in $\mathtt{meth}_{sub}$, then the redefined method must have a type which is a subtype of the original.

If `m` and `n` are records, then we write $\mathtt{m} \oplus \mathtt{n}$ for right dominant record combination. That is, if both `m` and `n` have a field with the same label `l` then the value from `n` is associated with `l` in the combination.

Let $\mathtt{inst}_{all} = \mathtt{inst} \oplus \mathtt{inst}_{sub}$ have type $\mathtt{IV}_{all}$. Similarly, let $\mathtt{meth}_{all} = \mathtt{meth} \oplus \mathtt{meth}_{sub}$ have type $\mathtt{M}_{all}$. Mimicking the encoding of classes from the previous section, one would anticipate that we could encode the subclass as follows:

$$\texttt{class inherit c (inst}_{sub}\texttt{,meth}_{sub}\texttt{)} = \langle \texttt{inst}_{all},\ \lambda(\texttt{self}: \texttt{VOT}(\texttt{IV}_{all}{}^{\texttt{ref}}, \texttt{M}_{all})).\texttt{meth}_{all}\rangle$$

Unfortunately, this simple encoding will not work for subclasses because of typing problems with inherited methods..

The problem is that when we type-checked the methods of `meth`, we presumed that the type of `self` was `VOT(I,M)`, whereas now we are using the inherited methods in a context where the type of `self` is $\texttt{VOT(IV}_{all}{}^{\texttt{ref}}\texttt{,M}_{all}\texttt{)}$. This would be fine if $\texttt{VOT(IV}_{all}{}^{\texttt{ref}}\texttt{,M}_{all}\texttt{)} <: \texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$, but that is NOT the case.

Recall that

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

Thus for $\texttt{VOT(IV}_{all}{}^{\text{ref}}\texttt{,M}_{all}\texttt{)} <: \texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$, we will need $\texttt{IV}_{all}{}^{\text{ref}} <: \texttt{IV}^{\texttt{ref}}$, and $\texttt{IV}_{all}{}^{\text{ref}} \rightarrow \texttt{M}_{all} <: \texttt{IV}^{\texttt{ref}} \rightarrow \texttt{M}$. The first is no problem since $\texttt{IV}_{all}{}^{\text{ref}}$ is simply an extension of $\texttt{IV}^{\texttt{ref}}$. Recall that subtyping for function types is covariant in the return type and contravariant in the domain type. The return types are in the subtype relation, as $\texttt{M}_{all}$is an extension of `M`, but the domains change in a covariant way, rather than contravariant. That is, we need $\texttt{IV}^{\texttt{ref}} <: \texttt{IV}_{all}{}^{\text{ref}}$ for the function types be subtypes, but we have the reverse.

The result is that the methods in `meth` may not be type safe when used in a context in which `self` has type $\texttt{VOT(IV}_{all}{}^{\text{ref}}\texttt{,M}_{all}\texttt{)}$. What does this imply about the type-safety of languages like Java and C++? Luckily, not very much.

Let's look at an example showing what can go wrong when you use a parameter, `self`, of type $\texttt{VOT(IV}_{all}{}^{\texttt{ref}}\texttt{,M}_{all}\texttt{)}$ in a context which is type-correct when `self` has type $\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$.

Let `oself` be a record of type {`j:Ref int`}. Define the class `c` as follows:

```
class C{
   j:int;
   m():void {⟨oself,snd(self)⟩.n()}
   n():int {val self.j}
}
```

The method `m` of `C` builds a new object of type $\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$ by replacing the first component of `self` by `oself`, and then sends it the message `n`. In other words, when method `n` is executed, it is with `self` interpreted by this new object whose instance variables are now represented by `oself`. All the method `n` of `C` does is to return the value of the instance variable `j`. This is no problem at all in `C` as `oself` has a field `j` of type `int`.

Now define subclass `SC` of `c` as follows:

```
class SC inherit c{
   k:int;
   n():int {val self.k}
}
```

Now we have a problem when method `m` is sent to an object generated from class `SC`. Since `m` is not overridden, the inherited code is run, resulting in sending a message `n` to an object whose method suite is the same (i.e., the full method suite of `n`), but whose instance variables have been replaced by `oself`. Thus when the evaluation of method `n` results in an attempt to access `self.k`, an error will result, as `oself` does not have field of type `k`.

Exercise 3.2 *Write out the translation of classes `C` and `SC` in detail, evaluate `new SC` to get an object, and then send that object the message `m` in order to show that a type error results.*

At this point, you are probably objecting that this is a cheat, as no object-oriented language would actually allow you to write a method like `m` above. Luckily, that is correct, for if it was legal, virtually all existing statically type-safe languages would allow type errors. Object-oriented languages do just not allow you to take apart `self` and then build a new object with a different collection of instance variables.

In fact we can only generate type errors with our encoding if we allow the methods to refer to the record of instance variables as a whole. If we only are allowed to access individual instance variables and methods of self, no errors result.

As a result, we could restrict the typed lambda calculus used here to disallow such access and then try to prove the correctness of our translation. However, it is quite difficult to restrict manipulations of `self` in the translation without also restricting other useful constructs in our underlying lambda calculus. Thus, rather than restricting the underlying calculus, we choose instead to modify our encoding of classes so that it will be type safe no matter what manipulations of `self` are made in the method bodies.

4 Rethinking classes to obtain type-safe inheritance

Let us rethink the definition of class from first principles. The problem is that we type-checked the methods presuming that the type of `self` was the same as the (visible) type of the objects generated by the class. Yet, these methods may be inherited in subclasses which generate objects having more instance variables and methods (let alone the fact that existing methods may be overridden). It's no wonder that we got into trouble!

4.1 Defining classes

If we want the methods in classes to be able to be inherited in subclasses without encountering new errors, we should type check them under the assumption that they may be included in any subclass of the class being defined. Of course, at the time we are defining a class, we have no idea what subclasses may later be defined. Though we showed earlier that if class `C` has type `Class(IV,M)` and `C'` is a subclass with type `Class(IV',M')`, then $\mathtt{IV'}^{\mathtt{ref}} <: \mathtt{IV}^{\mathtt{ref}}$ and $\mathtt{M'} <: \mathtt{M}$.

We can use this to define the encoding of classes in such a way that inherited methods may be used in a type-safe way in subclasses. As before, let `inst` of type `IV` be the record of initial values of instance variables for a class, and let `meth` of type `M` be the record of method bodies (which may include the keyword `self`). We modify the definition of class to explicitly consider the possibility of subclasses by abstracting over the types of the records of instance variables and methods.

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

As noted earlier, the types of the records of instance variables and method suites of subclasses will be subtypes of the superclass, so we need only abstract over those types. (While we have not formally defined this type of bounded quantification, we hope that its use will be clear here.) The type of a class is also modified:

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

The record of initial values of instance variables still has type `IV`, but the parameter `self` to the methods suite now has type `VOT(IV',M')`.

The creation of objects from classes is just a simple modification of our previous encoding. Apply the class to the type $\mathtt{IV}^{\mathtt{ref}}$ of its record of instance variables and `M`, the type of its record of methods, and then proceed exactly as before. Let `C` be a class of type `ClassType(IV,M)`. Then

```
new C = close[IVref,M]⟨instref,meth'⟩
```

where

```
C[IVref][M] = ⟨inst, λ(self:VOT(IVref,M)).meth⟩  and
meth'(inst':IVref) = meth(⟨inst',meth'⟩)
```

as before.

Going back to our `CellClass` example, we have `CellClass` = `class(inst,meth)` which is

```
Λ(M' <: CellM). Λ(IV' <: CellIVref).
     ⟨{x = 0},
      λ(self:VOT(IV',M'))
           {get = λ(_:void). self.x,
            set = λ(nuVal:int). self.x := nuVal,
            bump = λ(_:void). self.set(self.get()+1}⟩
```

As noted earlier, the type of `self` is now `VOT(I',M')`. Thus in order to ensure that the methods suite is well-typed, we must type check the methods under the assumption that `self` has type `VOT(I',M')`, where all we know is that `IV'` <: $\texttt{IV}^{\texttt{ref}}$ and `M'` <: `M`.

This may seem restrictive, but in fact it provides us with exactly the information necessary to type-check accesses to `self`. Because `IV'` <: $\texttt{IV}^{\texttt{ref}}$, `self.x` has type `Ref` τ iff `x` has type τ in `IV`.

Similarly, `self.m(arg)` is guaranteed to be safe only if `m` is in every extension `M'` of `M`, and in each has a type $\sigma_{M'} \to \tau_{M'}$, where the type of `arg` is some subtype of all of these $\sigma_{M'}$. This is equivalent to having `m` in `M` with type $\sigma \to \tau$ and `arg` having type σ.

Exercise 4.1 *Prove this last remark characterizing when* `self.m(arg)` *is type safe.*

Thus it appears we haven't lost any power in type-checking by parameterizing over subtypes of the type of the instance variables and methods. However, this change does eliminate our problem code from the last section which broke subtyping.

Here is the example again:

```
class C {
   j:int;
   m():void {⟨oself, snd(self)⟩.n()}
   n():int {val self.j}
}
```

where `oself` has type `{j:int}`.

Because we may only assume that the type of `self` is `VOT(IV',M')`, the example no longer type-checks. In particular, `⟨oself,snd(self)⟩` has type `IV * (IV' → M')` rather than `IV * (IV → M)`. Because we only know `IV'` <: `IV`, we cannot apply the second component of the pair to the first component. As a result, we cannot send a message to this pair. The class `C` will not type-check, and hence is illegal.

4.2 Defining subclasses: Try 2

So far, so good. Let's go back and define subclasses and make sure that inherited methods really will type check.

Suppose $\texttt{c} = \texttt{class(inst,meth)}$ has type $\texttt{ClassType(IV,M)}$, and let $\texttt{c}_{sub} = \texttt{class inherit c}$ $(\texttt{inst}_{sub},\texttt{meth}_{sub})$. As before, we assume that if $\texttt{inst}_{sub}$ redefines the initial value of an instance variable in `inst`, then they have the same type, and if a method in `meth` is redefined in $\texttt{meth}_{sub}$, then the redefined method must have a type which is a subtype of the original. Define $\texttt{inst}_{all} = \texttt{inst} \oplus \texttt{inst}_{sub}$ with type $\texttt{IV}_{sub}$ and $\texttt{meth}_{all} = \texttt{meth} \oplus \texttt{meth}_{sub}$ with type $\texttt{M}_{all}$. By the assumptions on overriding initial values of instance variables and methods, $\texttt{IV}_{all}{}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M}_{all} <: \texttt{M}$.

Define:

$$\texttt{class inherit c } (\texttt{inst}_{sub},\texttt{meth}_{sub}) = \texttt{class}(\texttt{inst}_{all},\texttt{meth}_{all})$$
$$= \Lambda(\texttt{M'} <: \texttt{M}_{all}).\ \Lambda(\texttt{IV'} <: \texttt{IV}_{all}{}^{\texttt{ref}}).\ \langle \texttt{inst}_{all},\ \lambda(\texttt{self:VOT(IV',M')}).\texttt{meth}_{all}\rangle$$

What about those inherited methods? Are they going to be type-correct? Recall that the methods in `meth` were type checked under the assumption that `self` had type `VOT(IV',M')` for $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$.

In the subclass the inherited methods are used where `self` has type `VOT(IV',M')` for $\texttt{IV'} <: \texttt{IV}_{all}{}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}_{all}$. However, as we noted earlier, $\texttt{IV}_{all}{}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M}_{all} <: \texttt{M}$. By transitivity, $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$, and we know that the inherited methods type check under that assumption.

In summary, in the subclass, the assumptions on `IV'` and `M'` are stronger than in the superclass, so anything that type-checks under the original, weaker assumptions, is guaranteed to type check under the stricter assumptions in the subclass. As a result, we have now succeeded in modeling objects, classes, and subclasses in such a way that inherited methods are guaranteed to remain type-correct if they were type-correct in the superclass.

4.3 A complication with `self`

Perhaps save this discussion until after discuss rigidities in typing systems.

There is one complication with `self` that we have not yet dealt with. Suppose we have a method which returns `self`. In type-checking methods of a class with type `ClassType(IV,M)`, we assumed that the type of `self` was `VOT(IV',M')` where $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$. If the method returns `self`, then it returns an element of type `VOT(IV',M')`.

However, it makes no sense to write `VOT(IV',M')`, as the return type of the method, as this will mean nothing to the user. Even when we create a new object and replace `IV'` by $\texttt{IV}^{\texttt{ref}}$ and `M'` by `M`, we cannot use $\texttt{IV}^{\texttt{ref}}$ outside of the object. This is forbidden by our typing rules for existentials because $\texttt{IV}^{\texttt{ref}}$ is hidden in the existential.

We solve this problem by applying `close` to `self` (or any other object of type $\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$ before returning it or using it as a parameter. Recall that $\texttt{close[IV}^{\texttt{ref}}\texttt{,M](self)}$ has type `OT(M)`, which may occur as a type.

One last complication is that the type of `self` in a method suite is taken as `VOT(IV',M')` for $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$. In order for a method returning `self` to type-check properly, we must replace `self` in the method body by `close[IV',M'](self)`, which has type `OT(M')`.

Even this is problematic as the type of a class is written as `ClassType(IV,M)`, and does not include mention of `M'`. Luckily, this can be taken care of by using the fact that if $\texttt{M'} <: \texttt{M}$, then $\texttt{OT(M')} <: \texttt{OT(M)}$. (This follows easily from the definition of `OT(M)` as an existential and the subtyping rule for existentials found in the homework.) Thus we can treat `close[IV',M'](self)` as though it has type `OT(M)`. Hence the return type of a method returning `self` in a class with type `ClassType(IV,M)` can be given as `OT(M)`. Similarly, `self` can be used as an actual parameter whenever the formal parameter has type `OT(M)`.

This solves our problem, though at the cost of losing type information. Let `C` be a class with type `ClassType(IV,M)` which has a method `m` which returns `self`. As noted above the return type of `m` will be `OT(M)`, the type of objects generated by `C`. Now suppose that `C'` is a subclass of `C` with type $\texttt{ClassType(IV}_{all}\texttt{,M}_{all}\texttt{)}$ where $\texttt{IV}_{all}{}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M}_{all} <: \texttt{M}$. Assume that `C` does not override method `m`. Then the type checker will assume the return type of the inherited method `m` will be `OT(M)`, even though if it were overridden with exactly the same body, it would type check as having return type $\texttt{OT(M}_{all}\texttt{)}$.

This loss of information is similar to that found in Java, where `clone` is declared in `Object` with return type `Object`. Now in all subclasses, `clone` also must return `Object`. In our encoding, we can actually do a bit better. If the user is willing to override `clone` in a class, we can change its return type to be the object type corresponding to the class. However, inherited methods never have their types automatically modified.