

Haskell++: An Object-Oriented Extension of Haskell

John Hughes and Jan Sparud,
Department of Computer Science,
Chalmers University,
Göteborg, SWEDEN.
{rjmh,sparud}@cs.chalmers.se

April 27, 1995

1 Introduction

Lazy functional languages such as Haskell[Hud92] provide excellent support for writing re-useable code. Polymorphism, higher-order functions, and lazy evaluation are all key contributing features:

- instead of writing many *sort* functions at different types, we re-use one;
- instead of writing many functions which recurse over lists, we capture common recursion patterns as higher-order functions *map*, *foldr* and so on, and just re-use them;
- instead of writing many loops that iterate *n* times, we write loops producing infinite lists and re-use *take n* to count the iterations.

This re-useability is reflected, for example, in the very heavy use that Haskell programs make of functions from the standard prelude. Indeed, we have argued elsewhere[Hug89] that these features are largely responsible for the improved productivity that functional programming offers.

Haskell's overloading system [WB89] also contributes to code re-useability. For example, most numeric functions in Haskell programs can be re-used with *any* implementation of numbers. Although in this case overloading can be regarded as syntactic sugar for parameterising numeric functions on the implementations of the arithmetic operations, the sugar is important because it makes re-useable code easy to write. Most programmers would probably regard passing the operations explicitly as unacceptably clumsy, and therefore wouldn't do it. The result: less re-useable code.

However, there is a form of re-use which Haskell does not support. Suppose we define a type T which is an instance of class C , and now we want to define a new type T' which contains a T and some additional components. Suppose further we want to make T' an instance of the same class. It is quite likely that the new components will only be significant for *some* of the class operations, and we would therefore like to write an instance declaration for T' where we only define these operations explicitly, and re-use T 's definitions for the others. Sadly this is impossible: instance declarations in Haskell must define *all* of the methods in the class¹. T' cannot *inherit* method definitions from T .

This kind of re-use is of course supported by *object-oriented* languages. Although the object-oriented languages used in practice are imperative, there is no shortage of functional variants in the theoretical literature — see for example [jfp94]. But these variants require relatively powerful and complicated type systems. For example, Pierce and Turner [PT94] show how an object-oriented language can be simply translated into F_{\leq}^{ω} , that is λ -calculus with higher-order polymorphism and subtyping.

Such a translation could be used directly to implement a functional object-oriented language, given a compiler for the target language, but unfortunately good compilers for F_{\leq}^{ω} are in short supply. In this paper we show how to translate an object-oriented language into Haskell instead. Our translation is rather similar to Pierce and Turner's, but where they use subtyping to allow an inherited method to be applied at different types, we use Haskell's overloading with automatic generation of suitable instances. We have implemented a simple pre-processor which translates an object-oriented extension of Haskell, 'Haskell++', into vanilla Haskell which can then be compiled with any of the existing compilers. We believe that, like Haskell's class system, our syntactic sugar can significantly improve code reuseability in practice.

2 An Introduction to Haskell++

We begin by giving an informal presentation of Haskell++, and showing how some well-known object-oriented examples can be programmed.

2.1 Object Classes and Instances

Haskell++ extends Haskell by providing a new kind of *class*, whose methods may be inherited from one instance to another. These classes are defined by an **object class** declaration. For example, let us declare an object class of points, with methods for extracting the coordinates, moving the point, and showing the point.

¹Haskell's default methods are not useful here: we want to re-use T 's methods, not some general class-wide method.

```

object class Point where
  x   :: Int
  y   :: Int
  mv  :: Int → Int → self
  sh  :: String

```

All the methods of an object class must be applied to an object of an appropriate type, and the type of this object is always called *self*. Since this first parameter must always be present, it is not given explicitly in the object class definition. The type of the *mv* method declared above, for example, is actually

$$mv :: Point\ self \Rightarrow self \rightarrow Int \rightarrow Int \rightarrow self$$

Moreover, since the type of the object operated on is always called *self*, there is no need for a type variable in the **object class** line.

To get any further we need a type which can be made an instance of this class. Type definitions are as in Haskell, for example

```

data VanillaPoint = P Int Int

```

Now we can make *VanillaPoint* an instance of class *Point* by making an **object instance** declaration. Just as the first parameter of each method was implicit in the object class definition, so it is implicit in the object instance definition. But in order to refer to the components of the object, we give a pattern that it must match in the declaration header. Within the method definitions we can refer to the variables bound in this pattern.

```

object instance Point (P x0 y0 :: VanillaPoint) where
  x = x0
  y = y0
  mv x' y' = self (P x' y')
  sh = show (x0, y0)

```

The occurrence of *self* in the *mv* method requires explanation. Remember that these methods may be inherited by other instance declarations. That means that the *mv* method defined here may very well be applied to types other than *VanillaPoint*! In other words, even though we are defining the *VanillaPoint* instance here, the type we called *self* above need not be *VanillaPoint*. To define *mv* by

$$mv\ x'\ y' = P\ x'\ y'$$

would therefore be a type error: the result is of type *VanillaPoint* but should be of type *self*. In order to construct values of the right type, Haskell++ provides a *function*, also called *self*, which can be used only in method definitions and whose type in this case is

$$self :: VanillaPoint \rightarrow self$$

Every type which inherits from *VanillaPoint* will contain a *VanillaPoint* component. The *self* function builds a copy of the object that the method it is used in is applied to, in which the *VanillaPoint* component is replaced by its argument. It provides a mechanism for Haskell++ methods to to invoke other methods in the same object, and to construct modified versions of the object they are applied to.

2.2 Inheritance

Now suppose we want to define a type of points which also carry a colour. We define

```
data Colour = Red | Yellow | Blue deriving Text
data ColourPoint = CP Colour VanillaPoint
```

Let us make *ColourPoint* an instance of class *Point*. The only method affected by the colour is *sh*: we want to show the colour too. We would like just to inherit the other methods from *VanillaPoint*. To do so, we define

```
object instance Point (CP c p :: ColourPoint)
inheriting x, y, mv from p where
  sh = super_sh ++ ", " ++ show c
```

Now when *mv*, for example, is applied to a *ColourPoint* it will just move the *VanillaPoint* component, leaving the *Colour* unchanged. For example,

$$mv\ 1\ 2\ (CP\ Red\ (P\ 0\ 0)) = CP\ Red\ (P\ 1\ 2)$$

The *mv* method inherited from *VanillaPoint* rebuilds a *ColourPoint* by applying the *self* function, which in this case is *(CP Red)*.

Even when defining methods explicitly, we can refer to the inherited methods under names beginning with *super_*. For example, here we have defined the *sh* method for *ColourPoints* in terms of the inherited *sh* method for *VanillaPoints*:

$$sh\ (CP\ Red\ (P\ 1\ 2)) = "(1,2), Red"$$

2.3 Virtual Methods

Inheritance behaves rather subtly when one object method is defined in terms of another. As an example, suppose we define a new class *UpperPoint*, whose objects can be shown in upper case. We shall provide two alternative methods for doing so, so we declare

```
object class Point  $\Rightarrow$  UpperPoint where
  shU  :: String
  shU' :: String
```

Here we require that instances of *UpperPoint* are also instances of *Point*.

Now let us make *VanillaPoint* an instance of this class:

```
object instance UpperPoint (p :: VanillaPoint) where  
  shU = map toUpper (sh (self p))  
  shU' = map toUpper (sh p)
```

Of course, these methods are defined in terms of *sh*.

We can inherit these definitions for *ColourPoints*:

```
object instance UpperPoint (CP c p :: ColourPoint)  
  inheriting shU, shU' from p
```

But *ColourPoints* have a different *sh* method from *Points*! The question is, do *shU* and *shU'* ‘see’ the *ColourPoint* *sh* method, even though they are inherited from *VanillaPoint*? In object-oriented languages the answer is ‘yes’, and indeed, classes often contain so-called *virtual methods*, which are left undefined at the ‘vanilla’ instance, and are intended to be overridden at *every* inheriting type.

In Haskell++ the behaviour of *shU* and *shU'* is different. When *shU* is applied to a *ColourPoint*, it uses the *self* function to reconstruct a *ColourPoint* and applies *sh* to that. It therefore ‘sees’ the new *sh* method:

```
shU (CP Red (P 1 2)) = "(1,2), RED"
```

On the other hand, *shU'* applies *sh* directly to a *VanillaPoint*. It does not see the new method therefore:

```
shU' (CP Red (P 1 2)) = "(1,2)"
```

This difference in behaviour helps explain why we need *super_*. Looking back at the definition of *sh* for *ColourPoints*, the reader may have wondered why we wrote

```
sh = super_sh ++ ", " ++ show c
```

instead of

```
sh = sh p ++ ", " ++ show c
```

Why do we need special syntax to call an inherited method, instead of just applying the method to the component we inherit from?

In this case the two definitions are actually equivalent, because the *sh* method for *VanillaPoints* does not use any other object methods. But suppose *sh* were defined in terms of the *x* and *y* methods, instead of the *x₀* and *y₀* components. These methods might be overridden at other types — for example, we might define a type *XAxisPoint*, inheriting from *ColourPoint*, whose *y* method always returns 0. Does the *sh* method for *XAxisPoints*, inherited from *ColourPoints*, see the new *y* method or not? With the definition in terms of *super_sh*, the answer is ‘yes’: the *sh* method inherited from *VanillaPoints* sees the modified *y* method. But with the definition in terms of *sh p* the answer is ‘no’: here we simply apply *sh* to a *VanillaPoint*, and all other information is lost.

2.4 Multiple Inheritance

It is straightforward for a type in Haskell++ to be an instance of several object classes, and to inherit from more than one component. For example, let us define a class of coloured objects:

```
object class Coloured where  
  colour      :: Colour  
  setColour   :: Colour → self
```

The simplest instance is just the type *Colour*:

```
object instance Coloured (c :: Colour) where  
  colour = c  
  setColour c' = self c'
```

Now it is easy to make *ColourPoints* into *Coloured* objects, by inheriting from the *Colour* component:

```
object instance Coloured (CP c p :: ColourPoint)  
  inheriting colour, setColour from c
```

ColourPoints now inherit in two different ways.

This is actually only a limited form of multiple inheritance, because we inherit the operations of each class quite independently, in separate instance declarations. We can't, for example, define a *sh* method for *ColourPoints* which uses both the inherited *sh* method from *VanillaPoint* and the inherited *colour* method from *Colour*.

2.5 Dynamic Binding

A very useful aspect of object-oriented languages is so-called *dynamic binding* of methods, which allows the overloading implicit in a method application to be resolved at run-time. For example, one may make a list containing both *VanillaPoints* and *ColourPoints*, and indeed any other instance of the *Point* class, and then map the *sh* method over the list. Thanks to dynamic binding, the appropriate *sh* method is invoked for each element.

In Haskell, and also Haskell++, one cannot even build such a list because the elements have different types. Pierce and Turner overcome this problem by concealing the representation type of *Point* objects using an existential type, so that every kind of point actually has the same type. Läufer and Odersky have shown how existential types can be added smoothly to ML [Läu92, LO92] and Haskell [Läu94], and how the resulting extension supports dynamic binding. Existential types are not yet a part of standard Haskell, but Augustsson has implemented them in **hbc**.

With Läufer's extension, we can define a type

```
data DynamicPoint = Point p => DP p
```

The *DP* constructor can be applied to *any* type *p* in class *Point*, but *the type of the result does not depend on p!* So if *vp* is a *VanillaPoint*, and *cp* is a *ColourPoint*, then we can form both *DP vp* and *DP cp*, and in both cases the result is of type *DynamicPoint*. We can therefore place both values in the same list. Of course, when we use a *DynamicPoint*, all we know about the component is that it is in class *Point*, and so the only operations we may apply to it are the corresponding class operations. When we do so, the appropriate instance is selected dynamically, implementing dynamic binding.

This extension is quite independent of Haskell++, but if we are using the preprocessor with a compiler supporting existential types, then we can conveniently make *DynamicPoint* an instance of class *Point* using inheritance:

```
object instance Point (DP p :: DynamicPoint)
inheriting x, y, mv, sh from p
```

3 Translating Haskell++ to Haskell

The fundamental problem in translating Haskell++ to Haskell is the implementation of inheritance. Our approach is as follows: any object which inherits methods from, say, *VanillaPoint*, must first be decomposed into a component of type *VanillaPoint*, and a function which rebuilds the rest of the object from that component (that is, *self*). So method bodies in Haskell++ actually have *two* hidden parameters: the *self* function and the component inherited from. Object instance declarations are translated into Haskell instance declarations just by adding these two parameters to every method. The ‘object pattern’ given in an object instance declaration is of course used to match the object parameter.

For example, the object instance declaration for *VanillaPoints*

```
object instance Point (P x0 y0 :: VanillaPoint) where
  x = x0
  y = y0
  mv x' y' = self (P x' y')
  sh = show (x0, y0)
```

is translated into the Haskell instance declaration

```
instance Point VanillaPoint where
  xBody self (P x0 y0) = x0
  yBody self (P x0 y0) = y0
  mvBody self (P x0 y0) x' y' = self (P x' y')
  shBody self (P x0 y0) = show (x0, y0)
```

Notice that the operations in the generated class are actually ‘method bodies’, not the methods themselves.

Correspondingly, object class declarations are translated by renaming the methods and adding the two hidden parameters to the type of each method. For example,

```
object class Point where
  x    :: Int
  y    :: Int
  mv   :: Int → Int → self
  sh   :: String
```

is translated into

```
class Point obj where
  xBody :: Point self ⇒ (obj → self) → obj → Int
  yBody :: Point self ⇒ (obj → self) → obj → Int
  mvBody :: Point self ⇒ (obj → self) → obj → Int → Int → self
  shBody :: Point self ⇒ (obj → self) → obj → String
```

Notice that the type parameter of the class is *not* the type *self*, it is a new type variable *obj* representing the type of the ‘object pattern’ in an instance declaration. When we define a particular instance, it is of course this type that we fix — but every instance definition must still be polymorphic in the type *self*. A little care is needed here to translate type contexts correctly, both in the types of individual methods and in the class header, but we leave the details to the reader.

The object methods themselves are defined by using the trivial decomposition of an object into itself and the identity function:

```
x obj = xBody id obj
y obj = yBody id obj
mv obj = mvBody id obj
sh obj = shBody id obj
```

These definitions are generated when the object class declaration is processed.

Inheriting a method is equivalent to defining it to be equal to the corresponding *super_* method, so for example

```
object instance Point (CP c p :: ColourPoint)
  inheriting x, y, mv from p where
    sh = super_sh ++ ", " ++ show c
```

is equivalent to

```
object instance Point (CP c p :: ColourPoint)
  inheriting from p where
    x = super_x
    y = super_y
    mv = super_mv
    sh = super_sh ++ ", " ++ show c
```

So it only remains to explain the translation of *super_* methods. They can only be used within the scope of *self* and the object pattern. We can therefore translate an occurrence of *super_mv*, for example, into an application of *mvBody* to the object component we're inheriting from, and a suitably extended *self* function. In this example, *self* maps *ColourPoints* to the *self* type, and we must pass the *VanillaPoint mv* method a function from *VanillaPoints* to this type. We can construct it as $self \circ (\lambda p \rightarrow (CP\ c\ p))$, and in general the appropriate *self* function is constructed similarly by composing the outer *self* with a λ -expression constructed from the object pattern and the name of the component we are inheriting from. So the generated definition for the *ColourPoint mv* method is

$$mvBody\ self\ (CP\ c\ p) = mvBody\ (self \circ (\lambda p \rightarrow (CP\ c\ p)))\ p$$

and similarly for the other methods.

This implementation of inheritance is strongly reminiscent of the one Pierce and Turner use [PT94]. But whereas we inherit from *obj* to *self* by passing the inherited method bodies an *obj* and a function $obj \rightarrow self$, Pierce and Turner pass it three parameters with types

- *self*,
- $self \rightarrow obj$,
- $self \rightarrow obj \rightarrow self$

namely the original object, a function to extract the component we are inheriting from, and a function to replace this component. In effect, we pass the applications of these two functions to the original object, rather than passing the object and functions separately. Pierce and Turner's idea is clearly more general, but we are unable to adopt it because in our framework *no such functions need exist!*

The problem is caused by existential types. Recall the type of dynamic points

$$\mathbf{data}\ DynamicPoint = Point\ p \Rightarrow DP\ p$$

which inherits all its methods from *p*. Our implementation of inheritance defines *DynamicPoint* methods by pattern matching on $(DP\ p)$ and invoking the corresponding method on *p*. The result is always of a type which does not involve *p*, and so the definition is well-typed. But using Pierce and Turner's idea, we would need to construct a *function* with type $DynamicPoint \rightarrow p$, which cannot be done because the existentially quantified type *p* escapes from its scope.

4 A Larger Example

To test the object-oriented features gained by combining the Haskell class system and existential types [Läu94] we had already written a simple drawing program in an object-oriented style. We have since rewritten the program in Haskell++.

In the Haskell version we defined a class for graphical objects with methods to draw them, move them, get information about them etc.

```

class Object a where
    origin      :: a → Point
    newOrigin   :: a → Point → a
    move        :: a → Point → a
    draw        :: a → [DrawCommand]
    ...

```

To introduce a new graphical object type, we define a Haskell datatype for it and then make the type an instance of the graphical object class. One benefit of this approach is that when extending the program with a new object type, all changes are made in one place. This is really a consequence of the object oriented methodology used. On the other hand, if we want to extend the class with a new method, then every instance declaration must be modified — and they may be spread over several different files.

In our simple program we defined object types for points, lines, rectangles, and circles. Here are the instance declarations for lines and rectangles.

```

data Line = Line Point Point
instance Object Line where
    origin (Line p1 _) = p1
    newOrigin (Line _ p2) p = Line p p2
    move (Line p1 p2) d = Line (move p1 d) (move p2 d)
    draw l = [DrawLine l]
    ...

data Rect = Rect Line
instance Object Rect where
    origin (Rect l) = origin l
    newOrigin (Rect l) p = Rect (newOrigin l p)
    move (Rect l) d = Rect (move l d)
    draw l = [DrawRectangle l]
    ...

```

(Rectangles are represented by their diagonal).

From these declarations we can see a drawback of the approach: it is clumsy to re-use properties of one instance of the graphical object types in another. For example, rectangles and lines have a common property in that they are characterised by two corner points. If we define a *move* function for lines, we would like to be able to use that function also when moving rectangles. To do this we must explicitly define a function in the rectangle instance that uses the *move* function from the line instance.

In order to manipulate objects of different graphical types together, we also defined an existential type and made it an instance of the graphical object class

(see 2.5). The instance declaration is straightforward but boring to write; every class method is defined in terms of the corresponding method for the contained ‘object’.

```

data O = (Object obj) ⇒ O obj
instance Object O where
    origin (O obj)           = origin obj
    newOrigin (O obj) p     = O (newOrigin obj p)
    move (O obj) d         = O (move obj d)
    draw (O obj)           = draw obj
    ...

```

Values of the different graphical object types were then embedded into this existential type so that they, e.g., could be put into a list. For example,

```

p1 = Point 0 0
p2 = Point 100 200
p3 = Point 50 100

objects = [O p3, O (Line p1 p2), O (Rect (Line p2 p3))]

```

One can now map a function over all objects in the list. Of course, the only interesting functions to use here are the class methods, since other functions cannot do anything with the values in the existential type. For example,

```

newObjects = [move obj (Point 30 50) | obj ← objects]
drawing = concat (map draw newObjects)

```

Thanks to dynamic binding, the *move* and *draw* methods used depend on the actual type of the object contained in each existential value.

After rewriting the program using Haskell++, the definition of the graphical object class looks like:

```

object class Object where
    origin      :: Point
    newOrigin   :: Point → self
    move        :: Point → self
    draw        :: [DrawCommand]
    ...

```

We can now exploit inheritance to reuse method definitions in several instances. In our case the rectangle instance can inherit the *origin*, *newOrigin*, and the *move* functions from its line component (and lines can inherit from their first point).

```

data Line = Line Point Point
object instance Object (Line p1 p2 :: Line)
  inheriting origin, newOrigin from p1 where
    move d = self (Line (move p1 d) (move p2 d))
    draw   = [DrawLine (Line p1 p2)]
  ...
data Rect = Rect Line
object instance Object (Rect l :: Rect)
  inheriting origin, newOrigin, move from l where
    draw = [DrawRectangle l]
  ...

```

We derive a number of benefits from using Haskell++. When we define a new object type that is very similar to an existing object type, it is easy to inherit most methods and redefine the few ones that need to be changed, saving programming effort. Moreover, if the type of a class method is changed, then the method definitions have to be changed in every instance when using plain Haskell. But in Haskell++ no changes are necessary in those instances which inherited the method — only in instances where it is really redefined. Since less needs to be written, the risk of errors is also reduced.

We exploited Haskell++’s virtual methods to define a *create* operation, that uses mouse dragging to place a new shape on the canvas. The *create* method is defined in the *Line* instance and inherited by every descendant, but makes use of the *draw* method that is redefined in each instance.

The Haskell++ version uses an existential graphical object type in just the same way as the Haskell one, but the corresponding instance declaration becomes very simple: it just inherits everything!

```

data O = (Object obj) => O obj
object instance Object (O obj :: O)
  inheriting origin, newOrigin, move, draw, ... from obj

```

In this example in particular, it is somewhat inconvenient to have to list all the methods one wants to inherit in an object instance. It would be better if all methods that are not explicitly redefined in an object instance declaration could be inherited automatically.

The graphical objects module in our example program defines one object class with 13 methods. There are five instances, in which 23 methods are defined explicitly, 32 are inherited (and ten are virtual). The number of source lines decreased from 160 to 90 when moving from Haskell to Haskell++ — a reduction of over 40%.

Of course, these figures very much depend on the application. An important point is that the lines that we do not need to write anymore are very ‘mechanical’: they are boring to write and prone to errors, and labour intensive to change should that be necessary.

5 Conclusions

Haskell++ extends Haskell with object classes, whose instances may inherit methods from one another. It is a rather minimal extension of Haskell: there are no new ‘object types’ or ‘object expressions’, for example. The only new features are object class and instance declarations.

Consequently the Haskell++ programmer must define object types using the existing Haskell type definition mechanism. Some may regard this as ‘hair shirt’ object-oriented programming. We prefer to say the new features are well integrated with the old — any Haskell type may be declared to be an instance of an object class.

The translation of Haskell++ to Haskell is straightforward. But the translations are sufficiently clumsy that few programmers would write them by hand. Therefore we believe that our ‘syntactic sugar’ leads to a real improvement in code reusability in practice.

We have tested Haskell++ in a non-trivial example, a simple object-oriented drawing program. The results show a significant improvement in code reuse compared to using existential types alone.

Our translator keeps no context information and therefore needs all inherited methods to be named explicitly, which is a little unusual for an object-oriented language. This can become tedious, especially when a new method is added to an object class and all instance declarations have to be changed. An easy extension would automatically inherit all methods which are not explicitly defined.

Initial experience of Haskell++ is encouraging, but much more work is required to establish whether a combination of object-oriented and functional programming is truly valuable in practice.

A prototype translator is available from the authors.

References

- [Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.
- [Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.
- [jfp94] Special issue on type systems for object-oriented programming. *Journal of Functional Programming*, 4(2), April 1994.
- [Läu92] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, Department of Computer Science, New York University, New York City, USA, 1992.

- [Läu94] Konstantin Läufer. Combining Type Classes and Existential Types. In *Proc. Latin American Informatics Conference (PANEL)*, Mexico, September 1994. ITESM-SEM.
- [LO92] Konstantin Läufer and Martin Odersky. An Extension of ML with First-Class Abstract Types. In *Proc. Workshop on ML and its Applications*, San Francisco, June 1992. ACM SIGPLAN.
- [PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.
- [WB89] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

A The Syntax of Haskell++

The syntax of Haskell is extended as follows:

$$\begin{array}{ll}
 \text{topdecl} & \rightarrow \text{object class } [\text{objcontext} \Rightarrow] \text{tycls } [\mathbf{where} \{ \text{objcbody } [;] \}] \\
 & | \text{object instance } [\text{context} \Rightarrow] \text{tycls } (\text{pat} :: \text{inst}) \\
 & \quad [\mathbf{inheriting } \text{var}_1, \dots, \text{var}_m \mathbf{from } \text{var}] \\
 & \quad [\mathbf{where} \{ \text{method}_1; \dots; \text{method}_n [;] \}] \\
 \text{objcontext} & \rightarrow \text{tycls} \\
 & | (\text{tycls}_1, \dots, \text{tycls}_n) \\
 \text{objcbody} & \rightarrow \text{objsign}_1; \dots; \text{objsign}_n \\
 \text{objsign} & \rightarrow \text{var} :: [\text{context} \Rightarrow] \text{type} \\
 \text{method} & \rightarrow \text{var } \text{apat}_1 \dots \text{apat}_n = \text{exp } [\mathbf{where} \{ \text{decls } [;] \}]
 \end{array}$$

Non-determinals which are left undefined here are defined in the Haskell report [Hud92].