6
Object Structures

Objects in the object-oriented programming paradigm consist of a state and a number of methods. Objects interact by calling methods. The method of an object A, when applied, updates the state of A and may reply to its caller. Although the interaction between two objects is local, it has a global effect: all objects that had A in scope before the interaction, now should have A with a changed state in scope. In this chapter we present a small framework in a pure, strongly typed, functional language in which we can define objects of arbitrary state and objects that consist of objects of arbitrary state. The objects in these structures can interact locally. Because we work in a functional framework, we need to define this local interaction between any two objects in an arbitrary object structure on a global level. This meaning is defined by an interpretation function on the global object structure level. We show that it is not possible to define a well-typed interpretation function in a standard polymorphic Milner/Mycroft type system, but instead need to strengthen the type system with type classes and constructor classes. The importance of this class of structures is illustrated by two applications of the framework: we obtain a model of composable, communicating, state based processes, and show how to build composable graphical user interfaces with local state.

6.1 Introduction

The effort of constructing software in a given programming language or programming system can be eased greatly if the system has tools to construct software components in isolation and compose these components afterwards to obtain new software components. Reasoning about the behaviour of software components constructed in this way is enhanced if it can be defined in such a system to what extent the internal state of a component is protected from external access. The object-oriented programming paradigm is a successful exponent of this approach.

There are many object-oriented languages, such as Smalltalk (Goldberg and Robson, 1983; Goldberg, 1992), Eiffel (Meyer, 1992), and C++ (Stroustrup, 1991). The object-oriented paradigm has been studied in functional languages as well. Some recent studies that also contain many pointers to other research in this area are Abadi (1994), Bruce (1994), and Pierce and Turner (1994). The main emphasis in these studies has been to unify the whole of the object-oriented paradigm, including concepts such as objects, classes, inheritance, and so on, within a functional framework.

In this chapter, we concentrate on the construction of objects and compositions of objects in the style of the object-oriented paradigm. We present a small, strongly typed framework in a pure functional language in which we can define objects in isolation and compose objects from objects. In our framework we study two kinds of objects: *context-independent* and *context-dependent* objects. Context-independent objects consist of a local state and one state transition function. Context-dependent objects have a local state as well, but their state transition function can also change a context of fixed type. In both cases this state transition function defines the behaviour of that object. Composition in this framework is obtained by allowing objects to contain other objects in their local state. The resulting object structures can have arbitrary type, and can be arbitrarily deeply nested. The behaviour of an object structure is defined with a special interpretation function.

Objects in our object structures interact by applying the method of an arbitrary object that is in their scope. The method of an object A, when applied, updates the state of A and may respond to its caller with an answer. The update of the state models a truly destructive update: to all objects that had A in their scope before the interaction now have A in their scope with a changed state. If we proceed in a similar way to define local interaction between any two objects in an object structure in a functional style, we will not obtain the desired meaning. Due to the functional semantics, the object interaction results in a new instance of A that is only in the scope of the calling object. Consequently, if we want to model this in a pure functional framework, then a local interaction (changing the state of some object) has a global effect (all objects now refer to the new object with the changed state). Basically we need to define an interpretation function on the global level as a function from object structure to object structure.

The type system of the functional language in this account must be sufficiently flexible not to restrict the types of the local state of objects, but also be sufficiently powerful to result in a strongly typed system. In this chapter we show that the standard polymorphic Milner/Mycroft type system (Milner, 1978; Mycroft, 1984), that forms the basis of the type systems of many functional programming languages, is not powerful enough to define a well-typed interpretation function. We show that to define the interpretation function of context-independent objects we need to resort to *type classes* as in Haskell (Hudak *et al*., 1992), and in case of context-dependent objects to *constructor classes* (Jones, 1993; 1995).

The functional language we use in this chapter is Clean (Brus *et al.*, 1987; Nöcker *et al.*, 1991; Plasmeijer and van Eekelen, 1993). The type system of Clean is based on a polymorphic Milner/Mycroft type system. Clean version 1.0 (Plasmeijer and van Eekelen, 1994) has been extended with overloading based on type classes and constructor classes. One other extension of Clean that we use in this chapter are records.

The layout of this chapter is as follows. In Section 6.2 we introduce our framework of objects and also show why it is not possible to define the interpretation functions in a polymorphic Milner/Mycroft type system. Section 6.3 introduces type classes and constructor classes as incorporated in the type system of Clean. Given this type system we reconstruct the interpretation functions. This is shown in Section 6.4. The object structures represent an important class of programming problems and can be applied in a wide range of areas. In Section 6.5 we give two applications of this framework. We develop a small framework of compositional state based processes, and a framework of compositional graphical user interface programs. We discuss related work in Section 6.6, and present conclusions in Section 6.7. Finally, we give some leads to current and future work in Section 6.8.

6.2 A framework of objects

6.2.1 Context-independent objects

In our framework *context-independent* objects are structured pairs of a state value and a function that, given this state value, computes a next state value of the same type. The polymorphic record type:

```
::  Object s
=   {   state :: s,
        change:: s -> s
    }
```

defines the type of an object with a local state of type s and a state transition function of type $s \rightarrow s$. Every value of type *Object s* is a context-independent object. (In Clean, type constructors and type variables are denoted by character sequences starting with a capital and lowercase respectively. Type definitions start with the '::' symbol. Record types are between '{' and '}' and consist of at least one field definition. A field definition consists of a field name, followed by a type definition. Field definitions are separated by ','. The type of a function with arguments of type $a_1,\ldots a_n$ and result type a is denoted by '$a_1 \ldots a_n$->a'.)

Composition in our framework is obtained by the observation that, due to polymorphism, the local state of an object can contain any typeable expression. In particular it can be a value that contains, or is, an object value. In this way we can compose arbitrarily deeply nested structures of objects of arbitrary local state type. Because values of type *Object* can contain nested occurrences of objects we call such values *object structures*.

The meaning associated with an object value is that whenever the object 'needs to act', its action depends on its local state and is defined by its state transition function. The operational behaviour of an object structure is defined by an *interpretation function* `interpret`. `interpret`, when applied to an object, applies the state transition function of the object to the local state of the object, and then continues recursively with all object elements in the new local state value. When applied to something else `interpret` simply yields its argument and terminates.

Figure 6.1 gives the recursive equations that define `interpret`. (In Clean variables are character sequences starting in lowercase. Inside a record pattern, the expression 'y=x' binds the variable x to the value of the field named y. An update of this field in a record r with a new value E is denoted by '{r & y=E}'. The expression 'x=:E' binds the variable x to the value denoted by

E.) Function alternatives are checked in textual order, and matched for patterns. So `interpret`'s first alternative matches object values, and its second alternative matches all other arguments.

```
interpret obj=:{state=s,change=f} = {obj & state=s1}
where
    s1 = interpret (f s)
interpret x = x
```

Figure 6.1 The recursive equations of `interpret`.

Because we work in a strongly typed polymorphic Milner/Mycroft system, let's see if we can derive the type of `interpret`. Consider the first alternative. From the argument pattern we derive that obj::*(Object x)*. So we can assume `interpret`::*(Object x)*$\rightarrow$*y* for *y* to be fixed during type derivation. Because obj::*(Object x)*, we have that s::x, and f::$x\rightarrow x$. Let $M\equiv(f\ s)$. Then M::x. The application `interpret`::*((Object x)*$\rightarrow$*y) M*::*x* has a type conflict, because there is no unifier for the type terms *(Object x)* and *x*.

The type conflict is due to the fact that the argument type of `interpret` is limited to object values. We can modify the scheme by tagging the alternatives that constitute an object structure. The type constructor *Object* is an algebraic data type. Its alternatives define that it is either an object (***Obj***) or something else (***Else***). (In Clean alternative constructors of algebraic data types are separated by '|'. By convention of presentation we print alternative constructors in **boldface**.)

```
::  Object s
=   Obj  (Object s) ((Object s)->(Object s))
|   Else s

interpret (Obj  s f) = Obj  (interpret (f s)) f
interpret (Else x  ) = Else x
```

We can now derive that `interpret` has type *(Object x)*$\rightarrow$*(Object x)*, but unfortunately this type restricts all leaf states to be of the *same* type *x*. There is no escape from this problem in Milner/Mycroft. In Section 6.4 we show that to assign a type to `interpret`, and use the original definition of *Object*, we need to resort to type classes.

6.2.2 Context-dependent objects

In our framework *context-dependent* objects have a local state of polymorphic type *s* and a state transition function that not only changes the local state value but also a global context value of polymorphic type *c* that is fixed for all objects. So the type of this transition function is $(s,c)\rightarrow(s,c)$. Let the type of such kind of object structures be the record type:

```
::  CObject s c
=   {   state :: s,
        change:: (s,c) -> (s,c)
    }
```

Every value of type *(CObject s c)* is a context-dependent object. If the local state of an object of type *(CObject s c)* contains an object structure of type *(CObject t c)* then this is a legal composition of context-dependent object structures.

The interpretation function for context-dependent object structures is `cinterpret`. Analogous to `interpret`, `cinterpret` when applied to the pair of an object structure of type *(CObject s c)* and a context value of type *c*, applies the state transition of the object to the pair of its local state and context value, and then continues recursively with all object structures in the pair formed by the new local state and context value. `cinterpret`, when applied to something else, leaves its argument unchanged and terminates. Figure 6.2 gives the recursive equations that define `cinterpret`.

```
cinterpret (obj=:{state=s,change=f},c) = ({obj & state=s1},c1)
where
    (s1,c1)  = cinterpret (f (s,c))
cinterpret x = x
```

Figure 6.2 The recursive equations of `cinterpret`.

As with `interpret` we cannot assign a type to `cinterpret` in a polymorphic Milner/Mycroft type system. In Section 6.4 we show that it is also not possible to assign a type in a system extended with type classes. The additional typing power we need is obtained by constructor classes. But first we introduce type classes and constructor classes of the Clean type system in the next section.

6.3 Type and constructor classes in Clean

In this section we introduce the type and constructor classes of Clean 1.0 (Plasmeijer and van Eekelen, 1995). Type classes have been introduced in Haskell (Hudak *et al*., 1992). Constructor classes introduced in Gofer (Jones, 1993; 1995), are a generalisation of type classes. Clean's type classes and constructor classes do not differ essentially from those of Haskell and Gofer. In contrast with Gofer, overloaded type and function definitions can be defined, applied, exported and imported in the module systems of Clean and Haskell in the same way as other language constructs can. This allows a programmer in the course of software development to add new instances to existing overloaded schemes. Below we give some examples of overloading in Clean.

In Clean 1.0 *type classes* identify collections of overloaded functions. Consider for instance the class *Eq* consisting of two overloaded, infix, operators == and <.

```
class Eq x
where (==) infix 2 :: x x -> Bool
      (<)  infix 2 :: x x -> Bool
```

A class declaration specifies a (number of) type scheme(s). The type variable after the class name specifies in which positions the function types are overloaded. Defining an instance of an overloaded function is done by uniform substitution of this type variable with a *flat* type definition. A type definition is flat if it is of the form T $a_1 \ldots a_n$ with T a type constructor of arity n, a_i a type variable, and $a_i \neq a_j$ for $i \neq j$. Synonym types are not legal type instances. The type variables in the substitution type are taken disjoint from the type variables in the type scheme. So given the *singleton* class declaration *class g x :: (x,y) x→(x,y)* taking the instance *(x,y)* does not result in the type *((x,y),y) (x,y) → ((x,y),y)* but in the type *((v,w),y) (v,w) → (v,w)*. This is important when we try to solve the local state problem in the next section. Observe that the result type of taking an instance is always a legal type.

Besides flat type instances, one can also define a *generic instance*. A generic instance has the same type as the type scheme of the overloaded function definition. So the concrete generic function has a polymorphic type. For instance, for the class Eq generic instances can be defined as:

```
instance Eq
where (==) infix 2 :: x x -> Bool
      (==) _ _ = False
      (<)  infix 2 :: x x -> Bool
      (<)  _ _ = False
```

Function definitions that use overloaded functions become overloaded themselves only if the Clean system is not able to infer a restricted context of application of such a function. Consider the following two functions:

```
eqnultuple (a,b)       = a==0 && b==0
eqtuple    (a,b) (c,d) = a==c && b==d
```

Although `eqnultuple` uses the overloaded == it can be inferred that an Integer instance is used due to the constant 0. So we obtain the derived type (Int, Int) $\rightarrow$ Bool. However, this cannot be inferred in case of `eqtuple`. In its type definition one needs to add the type classes of the overloaded functions, hence the type is (x,y) (x,y) $\rightarrow$ Bool | Eq x & Eq y.

Using type classes one specifies polymorphic type schemes of functions in which the type variables can be instantiated with different types. *Constructor classes* take a further step in this direction. The type scheme in a constructor class system allows type variables on type constructor positions. This permits a programmer to define function and type schemes that offers a restricted form of polymorphism in their type constructors.

Defining an instance of a function overloaded in a constructor position is also done by uniform substitution of this type variable with a *partial* flat type. A type definition is partial flat if it is a flat type of the form T $a_1 \ldots a_k$ with T a type constructor of arity n and $0 \leq k \leq n$. After substitution the resulting type must be a legal polymorphic type definition. Note that in contrast with type classes this is not always the case.

An illustration of constructor classes is the ubiquitous `map` example below. Given a function $f::a \rightarrow b$, `map` applies *f* to all elements of some object of type *c of a* in order to obtain an object of type *c of b*. The example defines overloaded instances of *f* for lists and trees.

```
class map c :: (a->b) (c a) -> c b

::  List x = Cons x (List x)
           | Nil
::  Tree x = Node (Tree x) (Tree x)
           | Leaf x

instance map List
map f (Cons x xs) = Cons (f x) (map f xs)
map _ Nil         = Nil

instance map Tree
map f (Node l r) = Node (map f l) (map f r)
map f (Leaf x)   = Leaf (f x)
```

In the next section there are more examples of the use of constructor classes. For a wide range of examples see Jones (1993, 1995).

6.4 The type of the interpretation functions

In this section we reconstruct the interpretation functions `interpret` and `cinterpret` introduced in Section 6.2. We start with `interpret`. First we introduce its class *Obj* .

```
class Obj x
where
interpret :: x -> x
```

From Figure 6.1 we can derive that the type instances for which `interpret` should be overloaded are *(Object s)* and 'something else'. So we need to define one instance of `interpret` for *(Object s)*, and one instance for 'something else'. In case of an *(Object s)* argument, `interpret` should be applied recursively to the `state` component of type *s* of its argument. Therefore the *(Object s)* instance needs to be restricted to the type class of `interpret`, Obj. When `interpret` is applied to 'something else', it is simply the identity function. This is the generic instance of Obj. The straightforward definition of `interpret` then becomes:

```
instance Obj (Object s)              | Obj s
where
interpret :: (Object s) -> Object s | Obj s
interpret obj=:{state=s,change=f} = {obj & state=s1}
where
    s1 = interpret (f s)

instance Obj
where
interpret :: x -> x
interpret x = x
```

Observe the similarity between this definition of the instances of `interpret` with the recursive equations given in Figure 6.1.

Before we continue with context-dependent objects, we add a small refinement to object structures. Due to the restriction of the local state type constructor to be of class *Obj* the type system enforces the local state of an object structure to be another object structure. Consequently, an object is either an *Object* value which local state is an object, or 'something else'. Of course the local state should also be used to store additional local information of the object. For this purpose we introduce two type constructor combinators:

```
::  List  s    = List  [s]
::  Pair  s t  = Pair  s t
```

List combines an arbitrary number of objects of the same type and *Pair* combines two objects of different type. Finally, we collect these type instances and definitions into one definition module, given in Figure 6.3.

```
definition module objects

::  Pair   s t = Pair  s t
::  List   s   = List  [s]
::  Object s   = {   state :: s,
                     change:: s -> s }

instance Obj (Pair   s t)        | Obj s & Obj t
```

```
instance Obj (List   s)          | Obj s
instance Obj (Object s)          | Obj s
instance Obj
```

Figure 6.3 The framework of context-independent objects.

We will now reconstruct `cinterpret`, the interpretation function for context-dependent objects. As a first attempt we simply follow the scheme we used for `interpret`: first we define its type class, and define one instance of `cinterpret` for *(CObject s c)* and one generic instance for 'something else' values. If we instantiate x with *(CObject s c)* then, as explained in Section 6.3, the substitution mechanism introduces a fresh variable name for *c*. So the type instances of `cinterpret` we end up with are:

```
class CObj x
where
cinterpret :: (x,c) -> (x,c)

instance CObj (CObject s c)
where
cinterpret :: (CObject s d, c) -> (CObject s d, c)
…
instance CObj
where
cinterpret :: (x,c) -> (x,c)
…
```

These type instances violate the desired property of legal compositions of context-dependent objects that the context of the composite structure is of the same type as the context we started with. In the type class system we cannot enforce type equality between the c type variable of the scheme and the c type variable of a type instance. However, this can be done if we use constructor classes and the following alternative type scheme of `cinterpret`:

```
class CObj x
where
cinterpret :: (x c,c) -> (x c,c)
```

This scheme expresses that `cinterpret` must be applied to a type constructor x that depends on the type variable c. The corresponding new definition of *CObject* is *CObject s c = {state::s c, change::(s c,c)→(s c,c)}*. If we substitute the partial type *(CObject s)* for x in the type scheme then we obtain the required type *(CObject s c)*. Although it is legal to substitute the type constructor *List* for x this results in the type *(List c)*. This type is inappropriate because it means that the type of the list elements of an object equal the type of the context. The definition of *List* (and analogously *Pair*) can be changed as follows to undo this:

```
::  List  s   c = List  [s c]
::  Pair  s t c = Pair  (s c) (t c)
```

So now we can instantiate x with *(CObject s)*, *(List s)*, and *(Pair s t)*. We also add the constructor class restrictions and obtain the following instance definitions of `cinterpret`.

```
instance CObj (CObject s)                           | CObj s
where
cinterpret :: (CObject s c, c) -> (CObject s c, c) | CObj s
cinterpret (obj=:{state=s,change=f},c) = ({obj & state=s1},c1)
```

```
where
    (s1,c1) = cinterpret (f (s,c))

instance CObj
where
cinterpret :: (x c, c) -> (x c, c)
cinterpret x = x
```

Again, observe the similarity between this definition of the instances of `cinterpret` with the recursive equations of Figure 6.2. Figure 6.4 gives the collection of the definitions of context-dependent object structures.

```
definition module cobjects

::  Pair    s t c = Pair  (s c) (t c)
::  List    s   c = List  [s c]
::  CObject s   c = {   state :: s c,
                        change:: (s c,c) -> (s c,c) }

instance CObj (Pair    s t)        | CObj s & CObj t
instance CObj (List    s)          | CObj s
instance CObj (CObject s)          | CObj s
instance CObj
```

Figure 6.4 The framework of context-dependent objects.

6.5 Applications of the framework

In this section we discuss two applications of the object framework. In the first example we show how to construct a framework of communicating, compositional processes, and in the second example we show how to construct graphical user interface programs. The examples are inspired by earlier work on the development of a concept of interactive process (Chapter 2, and Achten *et al*., 1993; Achten and Plasmeijer, 1993) and their composition in a functional framework (Chapter 3). In this section we will consider simplified versions of these systems.

6.5.1 Building process structures communicating by data sharing

In this section we consider how to obtain a small framework of compositional, state based processes. A process is basically a context-independent object: it is a structured pair of a local state and a state transition function. The semantics of a process is the subsequent application of the interpretation function `interpret`. We do not consider termination (processes evaluate infinitely) nor are we specific about the state transition function. Instead we concentrate on the *composition* of processes and inter-process communication based on *shared global state*.

Defining simplified processes as context-independent objects, we can apply the scheme developed in Section 6.4 and obtain a framework of composable processes. We define a new module `processes` that imports the module `objects` (Figure 6.3), and add the synonym type definition *Process s :== Object s*. The process structures that can be defined now are completely equivalent with context-independent object structures.

```
definition module processes

import objects

::  Process s :== Object s
```

Processes need to communicate. One inter-process communication primitive is *data sharing* (Chapter 3). Communication by data sharing is based on the fact that processes are state transition systems. In essence, an arbitrary number of processes P_i can communicate by data sharing if their states s_i are of the form $s_i=(l_i ,s)$. So the states have a common substructure s and state transition functions of type $s_i \rightarrow s_i$. Such a set of processes form a *process group*. In our framework we can identify such processes as context-dependent objects. The type:

```
::  Group s share
=   {   share     :: share,
        processes :: [s share]
    }
```

defines a process group as a structured pair of a shared value of type *share* and a list of *s* structures that depend on *share*. For *s* we intend to substitute process types. It appears that there are three sorts of process types that make sense to be substituted for *s*:

(**1**) Substitution with the type *Process* gives a process of type *(Object share)*. This process depends exclusively on the shared state.

(**2**) Substitution with the type *(CProcess l)* (with CProcess local share :== CObject local share) gives a process of type *(CObject l share)*. This process has a local and shared state.

(**3**) Substitution with the type *(SProcess l)* (with SProcess local share :== Object local) gives a process of type *(Object l)*. This process has a local state but ignores the shared state.

The fourth logical process that has neither a local nor a shared state is not considered to be a process because it has no state at all.

If we add the collection of new process definitions to the module `processes` defined above we obtain the following framework of composable processes. Observe that because synonym types are not legal type instances, the type *SProcess* has been defined as an algebraic type. The type instance *Object* is imported from `objects`. The other type instances and *Pair* and *List* are imported from `cobjects`.

```
definition module processes

from objects  import  Object, Obj, interpret
from cobjects import CObject,CObj,cinterpret,Pair,List

::  Process    s   :== Object  s
::  CProcess l s   :== CObject l s
::  SProcess l s   =   SObject (Object l)
::  Group  s share =   {   share     :: share,
                           processes :: [s share]   }

instance Obj (Group s share)      | Obj s
```

```
instance Obj (SProcess l s)        | Obj s
```

Figure 6.5 A framework of compositional processes including data sharing.

6.5.2 Building graphical user interfaces

One way to look at the construction of functional programs with graphical user interfaces is to regard them as state transition systems. This point of view has been explored extensively in the Clean event I/O project (Chapter 2, and Achten *et al*., 1993; Achten and Plasmeijer, 1993). In this system it seemed to be difficult to define local state. In this section we show how we can apply the object framework to solve this issue.

Figure 6.6 gives a simplified version of the event I/O system. In this version we ignore the issue of how a program provides visual feedback to the user. The simplified system consists of a tagged set of state transition functions that change the same state of polymorphic type *s*. The tagged functions are values of type *GUI s*. A value of this type defines that a graphical user interface element is either a button element with a name and a state transition function that defines its meaning, or a text field with a function that defines its content, or a window that contains further graphical user interface elements. Composition of GUI objects occurs by allowing the state of a GUI object to contain other GUI objects. So we must enforce composite GUI objects to change the same state s. So GUI composition is context-dependent object composition. For this purpose we introduce the disguised CObject type *CGUI s c*.

A graphical user interface program is a graphical user interface structure which is some type constructor *gui* depending on some state *s*. A program is evaluated by the function `doProgram`. This function basically repeatedly applies the interpretation function `cinterpret` to its argument. Again, we do not consider termination. The class restriction of `doProgram` defines the type instances that we allow, and therefore the graphical user interface object structures. These are the type instances *Pair*, *List*, and the generic instance of the constructor class CObj (imported from `cobjects`), and the type instances *GUI* and *CGUI s*.

```
definition module guis

from cobjects import CObj, Pair, List

::  CGUI s c
=   {   guistate:: s c,
        guidef  :: GUI (s c,c)
    }
::  GUI  s
=   Button String (s->s)
|   Text   (s->String)
|   Window [GUI s]

instance CObj GUI
instance CObj (CGUI s) | CObj s

doProgram :: (gui state) -> gui state | CObj gui
```

Figure 6.6 A compositional graphical user interface system.

Figure 6.7 presents an example of a graphical user interface program that is a compositional increment-decrement counter with reset button. The function `counter` is a composite control, consisting of a text field displaying the current integer local state, and two buttons, labelled '+' and '-' that increment and decrement the counter value respectively when selected. The local state of the counter is the current integer value. It should be observed that the local state can be of type Integer because of the generic instance of the constructor class CObj. The function `reset` has in its local state the counter, and adds a button, labelled 'reset'. When selected, the reset button sets the local counter state back to zero.

```
module counter

import guis, StdString, StdInt

Start   = doProgram reset

counter = { guistate = 0,
            guidef   = Window [ Text   display,
                                Button "+" (set 1),
                                Button "-" (set (-1)] }
where
    set dx  (count,c) = (count+dx,c)
    display (count,c) = toString count

reset   = { guistate = counter,
            guidef   = Button "reset" reset }
where
    reset (counter,c)= ({counter & guistate=0},c)
```

Figure 6.7 A compositional counter with reset button.

6.6 Related work

As we stated in the introduction, a lot of research has been conducted on object-oriented programming in functional languages, we mentioned the studies by Abadi (1994), Bruce (1994), and Pierce and Turner (1994). In this chapter we have concentrated on the issue of constructing objects and compositions of objects from objects. In the other studies objects exist on a global level only, and they are not concerned with object compositions.

Our approach of defining objects has been inspired by the work of Pierce and Turner (1994). In their scheme, objects are given essentially by the same record structure as our *Object* type, but the state is hidden using existential types. Expressed in Clean, their *Object* type is:

```
::  Object E.s
=   {   state :: s,
        change:: s -> s
    }
```

(In Clean all type variables of a type constructor must be introduced on its left-hand-side definition. Prefixing a type variable with '*E.*' indicates existential quantification.) With this type of *Object* one can define recursive data structures that contain objects each having a different type of the state field. Take for example *[{state=3, change=I}, {state=True, change=I}]* (with *I x =*

x) which is a list of objects and has type *[Object Void]*. (In a type instance the special type constructor *Void* must be substituted for existentially quantified type arguments.)

Defining objects in this way, and object composition as in our framework simplifies the task of typing `interpret` and `cinterpret`. We have `interpret`::*(Object Void)* $\rightarrow$ *(Object Void)* and `cinterpret`::*(Object Void c, c)* $\rightarrow$ *(Object Void c, c)*. But in this framework parent objects no longer have access to the state of their child objects. To solve this problem nested objects can allow access only via the change function. Consequently, objects are no longer of similar structure. Overloading can help to introduce new objects, but the problem remains that it is not possible to define an interpretation function that can be overloaded for all cases without recompilation.

6.7 Conclusions

In this chapter we have presented a functional, type-safe framework of compositional state based objects. The concept of state transition systems is general and can be applied in many ways. The framework demonstrates that state transition systems can be composed flexibly and uniformly. It should be observed that the composite structures that are defined in the framework do not require the additional overloading typing power of type classes and constructor classes. The additional typing power is necessary only to construct the interpretation functions. These interpretation functions define the operational semantics of composite object structures. In a well designed system they are not in scope of the programmer. We think that this account gives an interesting and useful application of the use of type and constructor classes in functional programming.

6.8 Current and future work

Currently we are evaluating in what way the framework as defined in this chapter can be incorporated in the Clean event I/O system and enhance its design. We have paid attention in Section 6.5 to how we can build process structures in this framework. In the same way we can incorporate the framework to build graphical user interface structures. The resulting system will remedy the main flaws of the simple, basic event I/O system which also have been observed by Noble and Runciman (1994).

Acknowledgements

The authors would like to thank Sjaak Smetsers, Eric Nöcker, and Marko van Eekelen for the many discussions on the use of type and constructor classes. Sjaak Smetsers has extended the type system of Clean with type and constructor classes.