Using Type and Constructor Classes to Interpret Object Structures

Peter Achten and Rinus Plasmeijer
Computing Science Institute, University of Nijmegen, 1 Toernooiveld, 6525ED, Nijmegen, The Netherlands
peter88@cs.kun.nl, rinus@cs.kun.nl

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 paper 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 assign a type to this 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 state based processes, and show how to build composable Graphical User Interfaces with local state.

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. A successful exponent of such a system is the object-oriented programming paradigm.

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 as objects, classes, inheritance, and so on, within a functional framework.

In this paper we concentrate on one specific aspect of the object-oriented paradigm, namely the construction of objects and compositions of objects. 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. The first kind of objects consist of a local state and one state transition function. The second kind of 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 would 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 in scope only 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 paper 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 assign a type to the interpretation function. We show that to type the interpretation function of objects of the first kind we need to resort to *type classes* as in Haskell (Hudak *et al.*, 1992), and in case of objects of the second kind *constructor classes* (Jones, 1993; 1995).

The functional language we use in this paper 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.

The layout of this paper is as follows. In Section 2 we introduce our framework of objects and also show why it is not possible to assign a type to the interpretation functions in a polymorphic Milner/Mycroft type system. Section 3 introduces type classes and constructor classes as incorporated in the type system of Clean. Given this type system we can assign types to the interpretation functions. This is shown in Section 4. The object structures represent an important class of programming problems and can be applied in a wide range of areas. In Section 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, and present conclusions in Section 7. Finally, we give some leads to current and future work in Section 8.

2 A framework of objects

Objects of the first kind in our framework 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* 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 an object of the first kind in our framework.

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

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 type able 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* φ. φ, 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 an argument that is neither an object nor a pair φ simply yields its argument and terminates.

Below we give the recursive equations that define φ. (In Clean variables are character sequences starting in lowercase. The expression ‘r.y’ denotes the selection of the value of a field named y of a record value r. An update of this field 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 φ’s first alternative matches object values, and its second alternative matches all other arguments.

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

Figure 1 The recursive equations of φ.

Because we want to have a strongly typed system, let’s see if we can derive the type of φ in a polymorphic Milner/Mycroft type system. We start with considering the first alternative. From the argument pattern we derive that $obj::(Object\ x)$. So we can assume $\varphi::(Object\ x)\rightarrow y$ 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 $\varphi::((Object\ x)\rightarrow y)\ M::x$ 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 φ is limited to object values. We can modify the scheme by tagging the alternatives that constitute an object structure, and obtain the following scheme. The type constructor *Object* is an algebraic data type. The alternatives of *Object* define that it is either an object (***Obj***) or something else (***Else***). (In Clean definitions of alternative constructors of algebraic data types are separated by ‘|’. By convention of presentation we print alternative constructors of algebraic types in **boldface**.)

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

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

Figure 1a Type tagged recursive equations of φ.

We can now derive that φ has type *(Object x)→(Object x)*, but unfortunately the type restricts all leaf states to be of the same type *x*. There is no escape from this problem. In Section 4 we show that to assign a type to φ, and use the original definition of *Object*, we need to resort to type classes.

Objects of the second kind in our framework 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)→(s,c)*. Let the type of such kind of object structures be the record type *(CObject s c)* below. Every value of type *(CObject s c)* is an object of the second kind in our framework.

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

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 object structures of the second kind.

Again, we define an interpretation function φ_c for object structures of the second kind. Analogous to φ, φ_c 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. φ_c, when applied to an argument that is not the pair of an object structure, leaves its argument unchanged and terminates. Below we give the recursive equations that define φ_c.

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

Figure 2 The recursive equations of φ_c.

As with φ we cannot assign a type to φ_c in a polymorphic Milner/Mycroft type system. In Section 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. First we will introduce type classes and constructor classes of the Clean type system in the next section.

3 Type and constructor classes in Clean

In this section we introduce the overloading mechanism of Clean 1.0 (Plasmeijer and van Eekelen, 1994) based on type and constructor classes. 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. Although type classes can be simulated in a polymorphic type system with records and higher-order functions such a simulation can be very inefficient. Including type classes in the type system of a language offers the compiler opportunities to generate good code in many cases (Plasmeijer and van Eekelen, 1994). 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 system of Clean 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 some examples of overloading in Clean.

In Clean 1.0 the basic way to define an overloaded function is by the overload declaration. Consider for instance the definition of an overloaded, infix, equality operator == that is applied to two arguments of the same type and yields True whenever the arguments are equal.

```
overload (==) infix 2 x :: x x -> Bool
```

An overload declaration specifies a type scheme. The type variable before '::' specifies in which positions the function type is 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 overload declaration:

```
overload g x :: (x,y) x -> (x,y)
```

taking the instance (x,y) does not result in the type ((x,y),y) (x,y) $\rightarrow$ ((x,y),y) but in the type ((v,w),y) (v,w) $\rightarrow$ (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.

One can also overload a group of functions. Such an overloaded group is a *type class*. For instance, suppose that we also define an overloaded, infix, ordering operator < along with ==.

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

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 $f::a \to b$, `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.

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

instance map []
map f [x:xs] = [f x:map f xs]
map _ []     = []

::  Tree x
=   (/\) infixl 0 (Tree x) (Tree x)
|   Leaf x

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

In Clean the type of a list of elements of type x is denoted by '[x]'. The empty list is denoted by '[]', and a list with head elements $a_1,\ldots a_n$ and tail list a is denoted by '[$a_1,\ldots a_n$: a]'. The example shows that binary alternative constructors may appear on infix positions. In the next section we will encounter more elaborate examples of the use of constructor classes. For a wide range of examples see Jones (1993, 1995).

4 The type of the interpretation functions

In this section we assign types to the interpretation functions φ and φ_c introduced in Section 2. First we define the classes *Objects* and *Objects*$_c$ with single members φ and φ_c respectively.

```
class Objects x
where
φ      :: x -> x

class Objects_c x
where
φ_c    :: (x,c) -> (x,c)
```

We first attend to the task of assigning a type to φ. From Figure 1 we can derive that the type instances for which φ should be overloaded are *(Object s)* and 'something else'. We define 'something else' to be everything that is not to be interpreted as an object. Every value x that is not to be interpreted as an object is denoted by the construction **`Local`** x for which we introduce the following type.

```
::  Local s = Local s
```

So φ is overloaded with *(Object s)* and *(Local s)*. Now consider the definition of φ. In case of *(Object s)*, the local state component `state` may contain further *Object* or *Local* structures. If we want to apply φ recursively to the `state` field, then we need to restrict the application to the type class *Objects* of φ. In case of *(Local s)*, φ simply is the identity function. A straightforward definition of φ then becomes:

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

instance Objects (Local s)
where
φ     :: (Local s) -> Local s
φ     x = x
```

Observe the similarity between this definition of the instances of φ with the recursive equations given in Figure 1.

Before we continue with objects of the second kind, we add a small refinement to object structures. Due to the restriction of the local state type constructor to be of class *Objects* the type system enforces the local state of an object structure to be another object structure. Consequently, in this framework an object is either a terminal value of type *Local*, or an object structure of type *Object* which local state is an object. Of course the local state should also be used to store local information of the object. We refine this by the following two combinator type constructors:

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

With *List* one 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 3. This sums up the framework of object structures of the first kind.

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

instances Objects
where
Local  s,
Pair   s t        | Objects s & Objects t,
List   s          | Objects s,
Object s          | Objects s
```

Figure 3 The framework of objects of the first kind.

We will now assign a type to φ_c, the interpretation function for objects of the second kind. As a first attempt we simply follow the scheme we used for φ, and define instances of φ_c for *(CObject s c)* and *(Local s)*. If we instantiate x with *(CObject s c)* then, as explained in Section 3, the substitution mechanism introduces a fresh variable name for c. So the type instances of φ_c we end up with are:

```
instance Objectsc (CObject s c)
where
φ     :: (CObject s d, c) -> (CObject s d, c)
…
instance Objectsc (Local   s)
where
φ     :: (Local   s,   c) -> (Local   s,   c)
…
```

These type instances violate the desired property of legal compositions of objects of the second kind 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 φ_c:

```
class Objectsc x
where
φc    :: (x c,c) -> (x c,c)
```

This scheme expresses that φ_c must be applied to a type constructor x that depends on the type variable 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 *Local* for x this results in the type *(Local c)*. This type is inappropriate because it means that the type of the local state of an object has always the type of the context. The definition of *Local* can be changed as follows to undo this (and also for the combinator constructor types *List* and *Pair*):

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

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

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

instance Objectsc  (Local s)
where
φc    :: (Local s c, c) -> (Local s c, c)
φc    x = x
```

Again, observe the similarity between this definition of the instances of φ_c with the recursive equations of Figure 2. Figure 4 gives the collection of the definitions of object structures of the second kind.

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

```
instances Objects_c
where
Local   s,
Pair    s t        | Objects_c s & Objects_c t,
List    s          | Objects_c s,
CObject s          | Objects_c s
```

Figure 4 The framework of objects of the second kind.

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 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 (Achten *et al*., 1993; Achten and Plasmeijer, 1993; 1995) and their composition in a functional framework (Achten and Plasmeijer, 1994). In this section we will consider simplified versions of these systems.

5.1 Building process structures

In this section we consider how to obtain a small framework of compositional, state based processes. A process is basically an object of the first kind in our framework: 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 φ. 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.

Because simplified processes are objects of the first kind, we can apply the scheme developed in Section 4 and obtain a framework of composable processes. A straightforward way to achieve this is to make use of the modular structure of Clean. Let the definitions in Figure 3 be defined in a module named `objects`, and the definitions in Figure 4 be defined in a module named `cobjects`. If we want processes to be synonym with objects of the first kind then we can simply define a new module, say `processes`, that imports the module `objects`, and add the synonym type definition :: Process s :== Object s. The processes structures that can be defined now are completely equivalent with object structures of the first kind.

```
definition module processes

import objects

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

In Achten and Plasmeijer (1994) an inter-process communication primitive called *data sharing* has been introduced. 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 local states s_i are of the form $s_i=(l_i,s)$. So the local states have a common substructure s and state transition functions of type $s_i \rightarrow s_i$. In our framework we can identify such processes as objects of the second kind. Such a set of processes form a *process group*. In our framework we can define process groups as follows:

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

The type *(Group 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 is a process that depends exclusively on the shared state.

(**2**) Substitution with the type *(CProcess local)* (given below) gives a process of type *(CObject local share)*. This is a process that has a local state and changes also the shared state.

```
::  CProcess local share :== CObject local share
```

(**3**) Substitution with the type *(SProcess local)* (given below) gives a process of type *(Object local)*. This is a process with a local state that simply ignores the shared state.

```
::  SProcess local share :== Object local
```

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 *Local*, *Pair*, and *List* are imported from `cobjects`.

```
definition module processes

from objects  import  Object, Objects
from cobjects import CObject,CObjects,Local,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 Objects
where
Group    s share      | Objects s,
SProcess l s          | Objects s
```

Figure 5 A framework of compositional processes including data sharing.

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 (Achten *et al*., 1993; Achten and Plasmeijer, 1993; 1995). In this system it seemed to be very difficult to define local state easily. In this section we show how we can use the object framework to solve this issue. First we give a simplified version of the sort of Graphical User Interface elements that we like to extend with local state.

Figure 6 gives a simplified version of the Event I/O system. In essence the 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**) a button element with a name and a state transition function that defines its meaning; (**b**) an edit text field with an initial string value; (**c**) a window that contains further Graphical User Interface elements. A program then is basically an object of the first kind: it consists of an initial value of the state and a set of Graphical User Interface elements, defined as a list. In this system we can dynamically add and remove GUI elements in a program. The associated semantics is a slightly adapted version of the interpretation function φ. We do not consider its definition.

```
::  GUI     s
=   Button  String (s->s)      (a)
|   Edit    String             (b)
|   Window  [GUI s]            (c)
::  Program s
=   {   pstate:: s,
        pGUI  :: [GUI s]
    }
```

Figure 6 A simple system of Graphical User Interface elements.

If we compose GUI objects in this system by allowing the state of a GUI object to contain other GUI objects, we need to enforce a nested GUI object to change the same state s. This composition requires object composition of the second kind with a context of type s. To obtain this we define the type *CGUI* which is basically a CObject in disguise:

```
::  CGUI s c
=   {   guistate:: s,
        guidef  :: GUI (s,c)
    }
```

Figure 7 gives the resulting definition of the system of compositional Graphical User Interfaces. To be able to define overloaded instances we also define the class *GUIs* which contains an adapted version of the interpretation function φ_c that we do not discuss. The type instances we substitute the state with is the familiar list. The final link we need to make between a program (which was basically an object of the first class) and GUI objects is by a small change in the definition of the type of a program. A program is a structured value of a state of type s and a GUI object that depends on this state. The function `doProgram` that is provided for programmers to start evaluation of such a program defines that the type instances of a program are indeed GUI objects. This function is basically the interpretation function φ_c.

```
::  Program s gui
```

```
=   {   pstate  :: s,
        pGUI    :: gui s
    }
::  CGUI    s c
=   {   guistate:: s,
        guidef  :: GUI (s,c)
    }
::  GUI     s
=   Button  String (s->s)
|   Edit    String
|   Window  [GUI s]

doProgram :: state (Program state gui) -> state | GUIs gui

class GUIs x
where
φc    :: (x c,c) -> (x c,c)

instances GUIs
where
Local s     | GUIs s,
Pair  s t   | GUIs s & GUIs t,
List  s     | GUIs s,
GUI,
CGUI s      | GUIs s
```

Figure 7 A compositional Graphical User Interface system.

6 Related work

As we stated in the introduction, a lot of research has been done in the area of object-oriented programming in functional languages, we mentioned the studies by Abadi (1994), Bruce (1994), and Pierce and Turner (1994). In this paper we have concentrated on the issue of constructing objects and compositions of objects from objects. In the other studies objects all exist on a global level, 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 quantification. Expressed in Clean (which also has Existential Types), their *Object* type becomes:

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

In Clean all type variables of a type constructor appear as arguments on the left-hand-side of a type definition. Prefixing a type variable with '*E.*' indicates existential quantification. With this type of *Object* we can now easily 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 specially defined type constructor *Void* must be substituted for all existentially quantified type arguments.

Defining objects in this way, and object composition as in our framework simplifies the task of assigning types to φ and φ_c. We have φ::*(Object Void)* $\rightarrow$

(Object Void) and φ_c::*(Object Void c, c)* $\rightarrow$ *(Object Void c, c)*. But this framework has as disadvantage that 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 because different objects have different access. One can use overloading to introduce new objects, but the essential flaw of this approach is that it is no longer possible to define an interpretation function that can be overloaded for all cases without recompilation.

7 Conclusions

In this paper we have presented a functional, type-safe framework of compositional state based objects. The concept of state transition systems is very 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 has been necessary only to assign types to 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.

8 Current and future work

Currently we are evaluating in what way the framework as defined in this paper can be incorporated in the Clean Event I/O system and enhance its design. We have paid attention in Section 5 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.

References

Abadi, M. 1994.
Baby Modula-3 and a theory of objects. In *Journal of Functional Programming* **4**(2), April 1994, Cambridge University Press, pp. 249-283.

Achten, P.M., van Groningen J.H.G., and Plasmeijer, M.J. 1993.
High Level Specification of I/O in Functional Languages. In Launchbury, J., Sansom, P. eds., *Proceedings Glasgow Workshop on Functional Programming*, Ayr, Scotland, 6-8 July 1992. Workshops in Computing, Springer-Verlag, Berlin, 1993, pp. 1-17.

Achten, P.M. and Plasmeijer, M.J. 1993.
The Beauty and the Beast. *Technical Report No.93-03*, March 1993. Research Institute for Declarative Systems, Department of Informatics, Faculty of Mathematics and Informatics, University of Nijmegen.

Achten, P.M. and Plasmeijer, M.J. 1994.
A Framework for Deterministically Interleaved Interactive Programs in the Functional Programming Language Clean. In Bakker, E. ed. *Proceedings Computing Science in the Netherlands, CSN'94*, Jaarbeurs Utrecht, The Netherlands, November 21-22, Stichting Mathematisch Centrum, Amsterdam, 1994, pp. 30-41.

Achten, P.M. and Plasmeijer, M.J. 1995.
The ins and outs of Clean I/O. In *Journal of Functional Programming* **5**(1) - January 1995, Cambridge University Press, pp. 81-110.

Bruce, K.B. 1994.
A paradigmatic object-oriented programming language: Design, static typing and semantics. In *Journal of Functional Programming* **4**(2), April 1994, Cambridge University Press, pp. 127-206.

Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, and Plasmeijer, M.J. 1987.
Clean: A Language for Functional Graph Rewriting. In Kahn. G. ed. *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.

Goldberg, A. 1992.
Object-Oriented Programming Languages. In Sebesta, R.W. *Concepts of Programming Languages* (2nd ed.). The Benjamin/Cummings Publishing Company, Inc.

Goldberg, A. and Robson, D. 1983.
Smalltalk-80—The Language and its Implementation. Addison-Wesley, Reading, MA.

Hudak, P., Peyton Jones, S., Wadler, Ph., Boutel, B., Fairbairn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, Th., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. 1992.
Report on the Programming Language Haskell. *ACM SigPlan Notices* **27**, (5), pp. 1-164.

Jones, M.P. 1993.
A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*. Copenhagen, Denmark, 9-11 June 1993. ACM Press, pp. 52-61.

Jones, M.P. 1995.
A system of constructor classes: overloading and implicit higher-order polymorphism. In *Journal of Functional Programming* **5**(1) - January 1995, Cambridge University Press, pp. 1-35.

Meyer, B. 1992.
Eiffel: the language. Prentice Hall International (UK) Ltd, 1992.

Milner, R.A. 1978.
Theory of Type Polymorphism in Programming. In *Journal of Computer Science and System Sciences*. Vol. 17, no. **3**, pp. 348-375.

Mycroft, A. 1984.
Polymorphic Type Schemes and Recursive Definitions. In *Proceedings of the sixth International Conference on Programming*. LNCS **167**, Springer-Verlag, pp. 217-228.

Noble, R. and Runciman, C. 1994.
Functional Languages and Graphical User Interfaces - a review and a case study. Department of Computer Science, University of York, England, February 3, 1994.

Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. 1991.
Concurrent Clean. In Aarts, E.H.L., Leeuwen, J. van, Rem, M., eds., *Proceedings of Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202-219.

Pierce, B.C., and Turner, D.N. 1994.
Simple type-theoretic foundations for object-oriented programming. In *Journal of Functional Programming* **4**(2), April 1994, Cambridge University Press, pp. 207-247.

Plasmeijer, M.J. and van Eekelen, M.C.J.D. 1993.
Functional Programming and Parallel Graph Rewriting. Addison-Wesley Publishing Company 1993.

Plasmeijer, M.J. and van Eekelen, M.C.J.D. 1994.
Clean 1.0 Reference Manual. *Technical Report*, in preparation. University of Nijmegen, The Netherlands.

Stroustrup, B. 1991.
The C++ Programming Language (2nd ed.). Addison-Wesley, Reading, MA, 1991.