5
Data Abstraction

Data abstraction mechanisms in programming languages are important tools in the process of producing reliable software. One well-known data abstraction mechanism is the abstract data type. Abstract data types either provide an abstraction of one implementation per program (single implementation), or allow the abstraction of arbitrary many implementations per program (multiple implementation). Two well-known approaches to multiple implementation abstract data types are data encapsulation and procedural encapsulation. Data encapsulation relies on a type system with existential types. Procedural encapsulation can be applied in a polymorphic Milner/Mycroft type system with algebraic types. In this chapter we investigate the incorporation of these multiple implementation abstract data types in the lazy, pure functional programming language Clean (version 1.0). The type system of Clean is based on a polymorphic Milner/Mycroft type system with algebraic types and existential types. One special feature of its type system is that it has a special linear type like type system called uniqueness types. The encorporation of the two encapsulation methods is straightforward if uniqueness types are not taken into account. However, the uniqueness type system causes complications with respect to procedural encapsulation. We present a semantically equivalent procedural encapsulation method that alleviates these complications. Finally, we compare the runtime performance of the two multiple implementation abstract data types with the standard single implementation abstract data type mechanism of Clean.

5.1 Introduction

Data abstraction is a useful and important technique in the development of reliable software (Fairley, 1985; Radin, 1992). The term data abstraction covers the area of software engineering methods that in one way or another hide implementation details of data structures. One well-known and widely used data abstraction technique is the abstract data type mechanism, that is incorporated in many programming languages.

The approaches to abstract data types in programming languages can be divided into two categories. In the first category, the *single implementation*, we find those abstraction mechanisms that simply encapsulate *one* specific implementation of the intended abstract data type. In the second category, the *multiple implementation*, we find mechanisms that offer a generic method of defining different implementations of one abstract type. For instance, introduce in either category an abstract data type for complex numbers implemented as pairs of real numbers (x,y), such that $(x,y) = x + iy$. In the single implementation category this will then be the only possible implementation of complex numbers in the same program. In the multiple implementation category we can introduce another implementation of complex numbers based on, say polar notation, in the same program. In this way programs can take advantage of different internal implementations in terms of run-time efficiency or space efficiency. A disadvantage of this approach is that coercions between different implementations are sometimes necessary. Single implementation abstract data types do not have this problem.

In this chapter we address multiple implementation data abstraction in a functional language. The way we regard abstract data types is rather conventional (following Mitchell and Plotkin (1985; 1988) and Pierce and Turner (1994)). Abstract data types are represented by *abstract object signatures*, which are structures of the following format: they consist of a value S, called the *abstract object state* (which is of some concrete type σ), and a fixed number of operations $f_1 \ldots f_N$ on values of that type, the *abstract object operations*. The type of the abstract object operations is the type Ops x = $(\underline{f}_1,\ldots,\underline{f}_N)$. In this chapter, the notation $\underline{f}$ is a shorthand notation for the type of f. The type of an abstract object signature is the type Sig x = (x, Ops x). Given such a structure $(S, (f_1,\ldots,f_N))$ of type Sig σ we want to obtain an equivalent structure, the *abstract structure*, that hides the actual implementation S. The type of the abstract structure, the *abstract data type*, hides the type σ of S. The only access to the implementation should be by the operations. Because the encapsulation method must allow multiple implementations, the abstract data type should be such that any two structures $(S, (f_1,\ldots,f_N))$ and $(T, (g_1,\ldots,g_N))$ with S of type σ and T of type τ, are of equal abstract data type if the type of each operation f_i modulo σ equals the type of g_i modulo τ.

Being able to represent abstract structures that have abstract data types as values in a functional language has as advantage that they are first-class citizens: they can be used as function arguments, and also be created by functions.

Solutions to multiple implementation methods can be divided into *data encapsulation* and *procedural encapsulation* approaches (Reynolds, 1978).

With data encapsulation both the implementation and the operations are given explicitly in the abstract structure, but the type and value of the implementation are hidden. Solutions in this direction have been taken by Mitchell and Plotkin (1985; 1988) who propose to use existential types to model abstract data types, and similarly Pierce and Turner (1994) who use existential types to model object-oriented data abstraction. So the way to proceed to obtain an abstract structure given a concrete structure $(S, (f_1,\ldots,f_N))$, is by giving it the abstract data type $T = \exists x.(x, \text{Ops } x)$. The rules for existential types hide all external access to the implementation.

For instance, define the type $T = \exists x.(x, x \rightarrow x \rightarrow x)$, and the function *new::Int→T* with *new i = (i, (+))*. Correctly typed expressions are: *new (3+5)* and $\lambda(i, f) = (f\ i\ i, f)$. Incorrectly typed expressions are: *(i+5, f) where (i, f) = new 3* and *swap* (i_1,f_1) *(i2,f2)* $= ((i_1,f_2), (i_2,f_1))$.

With procedural encapsulation the implementation of an abstract structure is encapsulated within the local data structures of the operations on the abstract value. To discuss procedural encapsulation, we require that the abstract object operations are *well-formed*. An abstract object operation f_i is well-formed if $\underline{f}_i$ has one of the following forms:

(1) $\underline{f}_i = s \rightarrow T_{i1} \ldots \rightarrow T_{in_i} \rightarrow s$
(2) $\underline{f}_i = s \rightarrow T_{i1} \ldots \rightarrow T_{in_i} \rightarrow T_{i(1+n_i)}$
(3) $\underline{f}_i = s \rightarrow T_{i1} \ldots \rightarrow T_{in_i} \rightarrow (s, T_{i(1+n_i)})$

The type variable s does not occur in any of the types T_{ij}, and for any i, $n_i \geq 0$. An abstract object operation is a function that accesses the abstract object state. The three forms correspond with the three kinds of possible access: an abstract object operation either updates the object state (**1**), or retrieves information from the object state without changing it (**2**), or updates the object state and retrieves information (**3**).

Let $(S, (f_1,\ldots,f_N))$ be an abstract object signature of type Sig σ with well-formed abstract object operations for which we derive an abstract type using procedural encapsulation. First we introduce function types $\underline{f}'_i$ such that $\underline{f}_i = \sigma \rightarrow \underline{f}'_i$. The abstract type is defined by the recursive type $T = (\underline{f}'_1,\ldots,\underline{f}'_N)\ [\sigma:=T]$ (the type obtained by substitution of all occurrences of σ by T in each $\underline{f}'_i$). As we have defined the abstract object state of an abstract structure to be some value S, we can encapsulate the abstract object state by curried application of the operations with S.

Figure 5.1 sketches how procedural encapsulation works. Consider the abstract object signature Sig x = (x, Ops x) with Ops x = (x→Int→x, x→Int→(Bool,x)). The encapsulated abstract type of this signature is T = (Int→T, Int→(Bool,T)). The function `intro`, given a signature, yields an abstract structure of type T as follows:

```
::  T      = (      Int -> T,       Int -> (Bool,T))
::  Ops x  = (x -> Int -> x,  x -> Int -> (Bool,x))

intro :: (x, Ops x) -> T
intro (x,(f1,f2))
=   (def1 x, def2 x)
where
    def1  x i     =    (def1 x`, def2 x`)
    where x`      = f1 x i

    def2  x i     = (b,(def1 x`, def2 x`))
    where (b,x`)  = f2 x i
```

Figure 5.1 Straightforward procedural encapsulation.

The type systems of most functional programming languages, such as Miranda and Haskell, are based on polymorphic Milner/Mycroft type systems, and do not have existential types. The abstract data type mechanisms supported in these languages are usually single implementation mechanisms. This implies that in these languages one needs to use the procedural encapsulation method sketched above to introduce a multiple implementation mechanism. It is unclear however what the effect will be on efficiency if this path is taken.

The lazy, pure functional programming language Clean (Brus *et al.,* 1987; Nöcker *et al.,* 1991; Plasmeijer and van Eekelen, 1993) has a polymorphic Milner-Mycroft type system, and supports single implementation abstract data types. Recently, its type system has been extended with existential types (going from version 0.8 to 1.0). The reason for this decision was caused by a practical need to have multiple implementation in the implementation and modelling of interactive processes (see Chapter 3)). This made the question on the consequences of efficiency more urgent, as a non-efficient implementation would be likely to deteriorate the performance of interactive programs. The type system of Clean extended with existential types enables us to investigate the performance of both encapsulation approaches in a state-of-the-art compiler.

Based on the conventional type system the type system of Clean has a linear kind of type system called uniqueness types (Smetsers *et al.,* 1993; Barendsen and Smetsers, 1993*a-b*, 1995, Barendsen, 1995). In Clean, conventional types of arbitrary data structures, functions, and recently also existential types, can be attributed with uniqueness information. The type checking algorithm will verify that functions that require a uniquely attributed argument have private access to these values. Uniqueness types have important applications in the area of incorporating I/O in pure functional languages (Chapter 2, and Achten *et al*., 1993), but also for optimizations of memory usage (Smetsers *et al.,* 1993).

The data encapsulation method, using existential types, complies well with respect to uniqueness information, and gives the results we would expect. Briefly, it allows concrete implementations of abstract structures to be single threaded. However, the procedural encapsulation method in the form as sketched above does not allow this. In this chapter we present a semantically equivalent procedural encapsulation method that behaves equally well with respect to uniqueness information as the data encapsulation method.

This chapter is structured as follows. In Section 5.2 we give a brief account of those aspects of uniqueness types that are relevant to this chapter. In Section 5.3 we explain the effect of uniqueness types on the encapsulation methods as sketched above, and show that the data encapsulation method is appropriate and the procedural encapsulation method is inappropriate with respect to uniqueness information. We continue in Section 5.4 by presenting a semantically equivalent procedural encapsulation method that has the same uniqueness properties as the data encapsulation method. We compare and analyze the runtime performances of abstract objects as obtained by the two methods in Section 5.5. In order to compare their 'absolute' efficiency we also compare them with the single implementation abstract data type mechanism of Clean. Related work is presented in Section 5.6 and Section 5.7 concludes.

5.2 Uniqueness types

In this section we give a brief introduction to the uniqueness type system of Clean. Clean is a term graph rewriting language (Barendregt *et al.*, 1987). While writing programs in Clean one does not need to be aware of the fact that these programs are actually computation graphs, that data types correspond with graph constructors, and that functions are graph transformations, also called rewrite rules. In the graph rewriting world we have the notion of reference count of a value which gives the number of references to the top node of that value. In Clean, functions can require values that have, at the moment of rewriting, reference count one. This means effectively that a function can request private access to arbitrary data structures. This request is indicated by annotating the corresponding type with the uniqueness attribute, the symbol '*'.

Uniqueness and the techniques to derive or enforce uniqueness are fundamental tools to solutions of important problems in functional programming languages. Using uniqueness types, it is possible to write programs that use memory efficiently because uniquely attributed data structures can be updated destructively rather than by copying (in Smetsers *et al.* (1993) an example is given of a functional quicksort algorithm that in-situ sorts a list of elements). Uniqueness types are crucial when operations are incorporated like file I/O and graphical user I/O (see Chapter 2, and Achten *et al*., 1993).

Uniqueness types are discussed extensively elsewhere: the first paper on uniqueness is Smetsers *et al.* (1993); thorough discussions on the framework are Barendsen and Smetsers (1993*a*,*b*) and Barendsen (1995). Type inference in Clean is discussed in Barendsen and Smetsers (1995*a*), and a presentation using a natural deduction style can be found in Barendsen and Smetsers (1995*b*). Relevant to this chapter is how conventional types are attributed in a consistent way with uniqueness information. We will illustrate this by an example from Barendsen and Smetsers (1993*b*). For any conventional type α, $*\alpha$ is the type α with the uniqueness attribute $*$. We first consider the recursive, polymorphic, algebraic type *List*, defined as List x = **Cons** x (List x) | **Nil**. Assignment of attributes must be done *uniformly* over type constructors (List), alternative constructors (**Cons** and **Nil**), and type variables (x). There are four possible uniform uniqueness attribute assignments for List, which are shown in Figure 5.2.

Cons = (x, List(x)) → List(x) (**1**)
Cons = (x,*List(x)) → *List(x) (**2**)
Cons = (*x, List(*x)) → List(*x) (**3**)
Cons = (*x,*List(*x)) → *List(*x) (**4**)

Figure 5.2 Uniqueness attribution in data types.

(**1**) Defines ordinary lists, (**2**) defines *spine-unique* lists, (**3**) defines *spine-shared* lists containing unique elements, and (**4**) defines totally unique lists. It should be noted that variant (**3**) cannot guarantee the uniqueness property to hold for its elements because the spine of the list can be shared, leading to multiple access by pattern-matching. The rule that rejects variation (**3**) is called the *uniqueness propagation rule*: if one of the argument types of a constructor has the uniqueness attribute, then the constructor also obtains the uniqueness attribute. This is taken into account in the type of **Cons**. The only legal types for **Cons** are the versions (**1**), (**2**), and (**4**).

For the function type constructor → there is no such propagation. So for the type F a b :== a→b, then either a or b or both can be attributed with * without F becoming unique as well. In a sense, one can say that → 'stops' uniqueness propagation. When using currying, the following holds. Consider f:: $*\sigma_1\ \sigma_2 \to \tau$, and an expression M:: $*\sigma_1$. Now the anonymous function (f M) obtains the uniqueness attribute and therefore has type $\sigma_2 * \!\!\to \tau$. (This is explained in more detail in Barendsen and Smetsers (1995*a*-*b*).)

5.3 The effect of uniqueness types

In this section we show the effect of the uniqueness type system on the encapsulation methods described in the introduction. First we consider data encapsulation method in Section 5.3.1, and then procedural encapsulation method in Section 5.3.2.

5.3.1 Uniqueness and data encapsulation

Before we discuss the effect of the uniqueness type system on the data encapsulation method we introduce how existential types have been incorporated in Clean. In the introduction we presented the type T = $\exists$x.(x, Ops x). However, in Clean all type variables of type constructors must be introduced on the left-hand side of the specification of the type constructor. Consequently, existentially quantified type variables also have to be quantified on the left-hand side, which deviates from regular notation. So the type T given above defined in Clean is: T E.x = (x, Ops x). Another consequence of this approach is that we need to give an instance to an existentially quantified type constructor. For this purpose the special type constructor *Void* is introduced. So, if we want to have a list of T, then this is defined as [T Void].

The purpose of introducing type variables only on the left-hand side of type constructors is that it enables us to determine all uniqueness information of the type variables used in the type constructor by inspecting the left-hand side only. For instance, a list of T of unique elements is defined as [T *Void].

As usual, multiple occurrences of type variables on the right-hand side of a type enforce type equality. So if we have the type definition T E.x = **T** x x, then (**T** 5 3) is a legal instance, while (**T** 5 3.0) is rejected because 5 and 3.0 are of different basic type. This can be applied as well for recursive types. Consider for instance the following definition of a list of existentially quantified elements: List E.x = **Cons** x (List Void) | **Nil**. Then (**Cons** 5 (**Cons** 3.0 **Nil**)) is of type List Void, because of the recursive occurrence of List. This is different from the more regular type List = ∃x.(List\` x) with List\` x = **Cons** x (List\` x) | **Nil** in which (**Cons** 5 (**Cons** 3.0 **Nil**)) is an illegal expression.

Existential types limit the access and use of existentially quantified values to existential scopes because in general it is not known what the actual instance type is: it is not legal to define a function that yields an existentially quantified value. For instance, the function hd that takes the head of a List Void argument, defined as hd (**Cons** x _) = x, is rejected by the type system because x is existentially quantified.

We can now consider how the uniqueness type system has its effect on the data encapsulation method. The Clean solution to the data encapsulation method in the extended type system with existential types is T E.x = (x, Ops x). If the abstract object state is attributed to be unique, then by uniqueness propagation the outermost constructor becomes unique as well. However, although the types of the operations depend on x, due to the function type constructor → uniqueness propagation stops. So the functions themselves do not become unique. So the attribution we obtain when x is unique is: *T E.*x = *(*x, Ops *x). This fully attributed type can also be abbreviated as *T E.*x = (x, Ops x). It should be observed that although x has the uniqueness attribute, this does not propagate to *Ops *x.

5.3.2 Uniqueness and procedural encapsulation

In Figure 5.1 we gave a straightforward solution to do procedural encapsulation. We defined the function `intro`, which generates an abstract structure of type T given an abstract object signature of type (x, Ops x). When applied to a unique abstract object state, `intro` has argument type (*x, Ops *x). It is easy to see that the introduction function can not yield an abstract structure that has a single threaded abstract object state because each of the operations *shares* the abstract object state explicitly:

$$\text{intro}\ (x, (f_1, f_2)) = (\text{def}_1\ x, \text{def}_2\ x).$$

So, because of this definition, x can not be unique. This is in conflict with the argument type which demands a uniquely attributed x. In the next section we present an alternative solution to the procedural encapsulation method that will allow us to create abstract structures with single threaded implementations.

5.4 Reconciling procedural encapsulation with uniqueness

In this section we present an alternative solution to the procedural encapsulation method in Section 5.4.1. We then show in Section 5.4.2 that the new approach allows the encapsulation of single threaded abstract object states.

5.4.1 The new procedural encapsulation method

To illustrate how the method works we use an adapted version of an example by Pierce and Turner (1994). In this example an object Point is introduced to create 'movable, one-dimensional point objects'. There are three operations on points, namely GetX, SetX, and RangeX. GetX::Point→Int, returns the integer value of a point. SetX::Point Int→Point, sets the current value of a point to a new integer value. Given a range of two values, RangeX::Point (Int, Int)→(Bool, Point), sets the point to the closest point in range if the point was out of range, otherwise the point is not changed. The Boolean result indicates whether the point was out of range.

The only language constructions we use in this method are higher-order functions, currying, and algebraic types. In the presentation below we use record types instead of algebraic types for convenience. However, in Clean record types are considered to be algebraic types with exactly one alternative constructor. So the presentation holds also for algebraic types.

We intend to construct an abstract data type P for some object. We will do this by some transformation steps. In the course of the transformation we introduce new type constructors. Their names are formed by extending the name P with a SMALL CAPS extension. The abstract object signature of this object is a parameterized record type PSIG x = {state::x, ops::POPS x}, with POPS x = $\{r_1::\underline{f_1},\ldots,r_N::\underline{f_N}\}$, with r_i the name of f_i starting in lowercase. (In Clean, record types are given between '{' and '}', having at least one field. A field is given by a name, followed by a type definition.) The abstract object operations $f_1,\ldots,f_N$ must be well-formed. The abstract object signature of our point example type Point is:

```
::  PointSIG x =   {  state :: x,
                      ops   :: PointOPS x
                   }
::  PointOPS x =   {  getX  :: x->Int,
                      setX  :: x  Int->x,
                      rangeX:: x (Int,Int)->(x,Bool)
                   }
```

So <u>GetX</u> = x→Int, <u>SetX</u> = x Int→x, and <u>RangeX</u> = x (Int,Int)→(x,Bool). Note that all these operations are well-formed, and that of each form one operation is present.

The basic idea of the encapsulation method is that the abstract object operations can also be represented by one single function. Any set of functions $f_1\ldots f_N$ with $\underline{f_i}=A_i\to B_i$, which is a structure of type $(A_1\to B_1)\times\ldots\times(A_N\to B_N)$ can be embedded (≤) in one function with tagged arguments $(A_1 \oplus\ldots\oplus A_N)$ and tagged results $(B_1 \oplus\ldots\oplus B_N)$, of type $(A_1 \oplus\ldots\oplus A_N) \to (B_1 \oplus\ldots\oplus B_N)$. So, we have:

$$(A_1\to B_1)\times\ldots\times(A_N\to B_N) \leq (A_1 \oplus\ldots\oplus A_N) \to (B_1 \oplus\ldots\oplus B_N)$$

For this function, the arguments and results of the abstract object operations are reordered. Of each operation, the argument types T_{ij} are tagged, and the result types are tagged. The new function has two arguments: the first argument is a concrete implementation and the second argument is a tagged argument type structure. Given the tag, the function knows which abstract operation from the concrete implementation to apply to the object state of the concrete implementation. The result of this application is provided with the proper tag. The essential step is that the new abstract object is the *curried application of this function* with the new abstract object state. So an abstract object is an anonymous function which type does not depend on the type variable x. The remaining part of this section presents the encapsulation method which consists of three steps.

Step 1 Reordering types and tagging

The abstract type P is a function type, defined as P :== PIN → POUT. The algebraic types PIN and POUT are tagged collections of the input arguments and output results of the abstract object operations respectively. For each abstract object operation f_i with $\underline{f_i}$ = x $T_{i1} \ldots T_{in_i} \rightarrow$ T, PIN has an alternative constructor tag that indicates the name of f_i and contains the arguments of type $T_{i1} \ldots T_{in_i}$. Similarly, for each f_i, POUT has an alternative constructor tag that indicates the name of f_i, and contains the result of type T[x:=P]. Figure 5.3 gives the results for our abstract type Point.

```
::  Point           :== PointIN -> PointOUT
::  PointIN          =   GetXIN
                     |   SetXIN     Int
                     |   RangeXIN   (Int,Int)
::  PointOUT         =   GetXOUT    Int
                     |   SetXOUT    Point
                     |   RangeXOUT  (Point,Bool)
```

Figure 5.3 The point abstract object type.

Furthermore, we define for each abstract object operation f_i two additional functions Tagf_i and Untagf_i. Let $\underline{f_i}$ = x $T_{i1} \ldots T_{in_i} \rightarrow$T. The function Tagf_i creates the algebraic type alternative of PIN that corresponds with f_i. So we have $\underline{\text{Tagf}_i} = T_{i1} \ldots T_{in_i} \rightarrow$PIN. The function Untagf_i extracts the operation result that is the argument of the POUT constructor that corresponds with f_i. So $\underline{\text{Untagf}_i}$ = POUT→T[x:=P]. Figure 5.4 gives the tagging and untagging functions of Point.

```
TagGetX      :: PointIN
TagGetX      = GetXIN

TagSetX      :: Int -> PointIN
TagSetX      n = SetXIN n

TagRangeX    :: (Int,Int) -> PointIN
TagRangeX    n = ScaleXIN n

UntagGetX    :: PointOUT -> Int
UntagGetX    (GetXOUT n) = n

UntagSetX    :: PointOUT -> Point
UntagSetX    (SetXOUT n) = n

UntagRangeX :: PointOUT -> (Point,Bool)
UntagRangeX (RangeXOUT n) = n
```

Figure 5.4 The tagging and untagging functions of the point example.

Step 2 The object definition function

The *object definition function* PDEF is the function that relates concrete implementations of type PSIG x with tagged inputs PIN and tagged outputs POUT. The type of the object definition function is (PSIG x)→PIN→POUT (which is equivalent to the type (PSIG x)→P). For each abstract object operation the object definition function has a corresponding function alternative. Figure 5.5 gives the PointDEF function. (In Clean, record patterns are given between '{', and '}'. Variable names can be bound to the record value (like r=:{…}), or by giving the field name (like {state,…}). Given a record reference r, its field x can be selected by 'r.x'. Given a record reference r, its field x can be updated resulting in the new record {r & x=…}.)

```
PointDEF :: (PointSIG x) PointIN -> PointOUT
PointDEF x=:{state,ops} GetXIN
=   GetXOUT (ops.getX state)
PointDEF x=:{state,ops} (SetXIN n)
=   SetXOUT (PointDEF {x & state=ops.setX state n})
PointDEF x=:{state,ops} (RangeXIN n)
=   RangeXOUT (PointDEF {x & state=state1}, b)
where
    (state1,b) = ops.rangeX state n
```

Figure 5.5 The object definition function of the point example.

Step 3 Abstract object introduction and abstract operations

In the previous step we have defined the object definition function. From this function we derive abstract objects. They are created by parametrization of the object definition function with a concrete implementation. This is done by an *abstract object introduction function*. Figure 5.6 gives the definition of the abstract object introduction function for points.

```
PointINTRO :: (PointSIG x) -> Point
PointINTRO impl = PointDEF impl
```

Figure 5.6 The abstract object introduction function of the point example.

Finally we introduce for each abstract object operation f_i a corresponding *abstract operation* F_i with $\underline{F}_i = \underline{f}_i[x:=P]$. Each abstract operation F_i basically has the form $F_i\ p\ t_1 \ldots t_{n_i}$ = Untagf_i (p (Tagf_i $t_1 \ldots t_{n_i}$)). Figure 5.7 gives the abstract operations on points.

```
GetX :: Point -> Int
GetX point = UntagGetX (point TagGetX)

SetX :: Point Int -> Point
SetX point n = UntagSetX (point (TagSetX n))

RangeX :: Point Real -> (Point, Bool)
RangeX point r = UntagRangeX (point (TagRangeX r))
```

Figure 5.7 The abstract operations on points.

The abstract object introduction function and the abstract operations are functional sugar. They can be defined entirely in terms of the object definition function. But this function presents some problems. The first problem is that the relation between arguments and result is not clear from the type of the object definition function. For instance, it cannot be determined from the type of PointDEF that **GetXIN** will always result in (**GetXOUT** n), for some integer n. Another problem is that applications of PointDEF always need to untag the result of PointDEF, which results in a clumsy programming style. Finally, the implementation of the object definition function cannot be hidden by some an abstract data type mechanism, because one loses the information that abstract objects are functions. These problems are solved by the abstract operations and the abstract object introduction function.

5.4.2 Procedural encapsulation hides unique data

We will now consider the effect of adding uniqueness information to abstract objects. The new procedural encapsulation method is applied to well-formed signatures of type PSIG x. This signature can be attributed with uniqueness information in three ways:

(**1**) The object state x is unique

(**2**) Some operation arguments are unique

(**3**) Some operation results are unique

Any combination of these assignments is permitted. The effect of a combination of these assignments on the encapsulation method is the accumulation of the effects of (**1**), (**2**), and (**3**) individually. So we can study the effects of each assignment individually.

(**1**) The well-attributed signature of an abstract object with a unique object state is *PSIG *x which results in the fully attributed type *PSIG *x = *{state::*x, ops::POPS *x}. The abstract type P is also unique because it is the parametrization of a function of type PDEF with the signature value of unique type *PSIG *x. Note that POPS is not unique, because it contains function types only.

(**2**) The signature type PSIG x is not changed if some of the arguments of the operations are unique because uniqueness propagation stops at the function type constructor. Of the elementary types derived in the first step of the encapsulation method, only the PIN type becomes unique because of the uniqueness propagation rule. Therefore the type of the abstract object function PDEF is (PSIG x) *PIN→POUT. So the abstract object P has type *PIN→POUT, but is not unique itself (again, the uniqueness propagation rule blocks at the function type constructor).

(**3**) The signature type PSIG x is not changed if some of the results of its operations are unique because uniqueness propagation stops at the function type constructor. Of the elementary types derived in the first step of the encapsulation method, now the POUT type becomes unique because it depends on the POUT type. The type POUT is unique because it depends on the result types of the operations. So the type of the abstract object function PDEF is (PSIG x) PIN→*POUT, and the abstract object P has type PIN→*POUT, but again, is not unique itself.

From these cases we can conclude that assigning uniqueness attributes to an abstract object signature yields the types we expect: the abstract data type P is unique if and only if the concrete implementation is unique. There are no restrictions on the uniqueness of the types of the operations. So the new procedural encapsulation method has the same uniqueness properties as the data encapsulation method.

5.5 Efficiency

In this section we compare the runtime efficiency of the programs that result from the two abstraction methods. The abstract objects we consider in the tests are complex numbers, given by the abstract type C. Its signature is given in Figure 5.8.

```
::  CSIG     x = { state :: x,
                   ops   :: COPS x           }
::  COPS     x = { get   :: x -> (!Real,!Real),
                   sq    :: x -> x           }
```

Figure 5.8 The abstract object signature of complex numbers.

The abstract operation get retrieves the two Real components from a complex number as in the interpretation in Section 5.1. The operation sq simply takes the square of itself. Finally, the introduction function z_0 yields the zero complex number.

As a standard measure, Figure 5.9*a* gives an implementation of complex numbers using the standard Clean abstract data type mechanism to compare the runtime performances. It uses a strict tuple implementation, and the abstract operations Get and Sq are also strict (annotated in the type with the '!' symbol) in their arguments. In general, adding strictness information gives a considerable increase in runtime performance (Nöcker and Smetsers, 1990, 1993; Smetsers *et al*., 1991; Nöcker, 1993; 1994).

Figure 5.9*b-c* gives the resulting program using procedural and data encapsulation respectively. For presentational reasons we have introduced the (un)tagging functions and the object definition function as local function definitions of the abstract operations and abstract object introduction function respectively. There are two further changes to the program codes:

Firstly, the types CSIG and COPS have been replaced by algebraic types rather than record types. First tests showed that algebraic types were more efficient than record types. The first method using record types ran for 0.36 seconds, while the second method with records ran for 0.25 seconds. So with algebraic types tests ran 116% and 120% faster.

Secondly, we have annotated types and functions with strictness information. If we would not have done this, the comparison would very obviously give bad results for the encapsulation methods compared with the very efficient version of Figure 5.9*a*. This is because the encapsulation methods both generate lazy contexts where the strict tuple version generates strict contexts. To neutralize this effect, we have annotated the types and functions of the encapsultation methods in the corresponding places. So the final measurements will be affected only by the true overhead costs of both methods.

(a) Implementation of complex numbers based on strict tuples:

```
::  C :== (!Real,!Real)

z0     :: Complex
z0     = (0.0,0.0)

Get    :: !C -> (!Real,!Real)
Get    c = c

Sq     :: !C -> C
Sq     (c1,c2) = (c1*c1+c2*c1,c1*c2+c2*c2)
```

(b) Complex numbers with procedural encapsulation:

```
::  CSIG     x =    CSIG x (COPS x)
::  COPS     x =    COPS (x -> (!Real,!Real)) (x -> x)
::  C          :== CIN -> COUT
::  CIN        =    GetIN                      | SqIN
::  COUT       =    GetOUT !(!Real,!Real)      | SqOUT !C

z0 :: C
z0 = CDEF (CSIG (0.0,0.0) (COPS get sq))
where
    get :: !(!Real,!Real) -> (!Real,!Real)
    get c = c

    sq :: !(!Real,!Real) -> (!Real,!Real)
    sq (c1,c2) = (c1*c1+c2*c1,c1*c2+c2*c2)

    CDEF :: !(CSIG c) !CIN -> COUT
    CDEF (CSIG c ops=:(COPS get sq)) GetIN
    =  GetOUT (get c)
    CDEF (CSIG c ops=:(COPS get sq)) SqIN
    =  SqOUT  (CDEF (CSIG (sq c) ops))

Get :: !C -> (!Real,!Real)
Get c = Untagget (c Tagget)
where
    Tagget   :: CIN
    Tagget   =  GetIN
    Untagget :: !COUT -> (!Real,!Real)
    Untagget (GetOUT r) = r

Sq :: !C -> C
Sq c = Untagsq (c Tagsq)
where
    Tagsq   :: CIN
    Tagsq   =  SqIN
    Untagsq :: !COUT -> C
    Untagsq (SqOUT c) = c
```

(c) Complex numbers with data encapsulation:

```
::  CSIG      E.x =   CSIG x (COPS x)
::  COPS        x =   COPS (x -> (!Real,!Real)) (x -> x)
::  C             :== CSIG Void

z0 :: C
z0 = CSIG (0.0,0.0) (COPS get sq))
where
    get :: !(!Real,!Real) -> (!Real,!Real)
    get c = c

    sq :: !(!Real,!Real) -> (!Real,!Real)
    sq (c1,c2) c = (c1*c1+c2*c1,c1*c2+c2*c2)

Get :: !C -> (!Real,!Real)
Get (CSIG c ops=:(COPS get sq)) = get c

Sq :: !C -> C
Sq (CSIG c ops=:(COPS get sq)) = CSIG (sq c) ops
```

Figure 5.9 The three implementations of complex numbers.

Figure 5.10 gives the test program which calculates z_{10000} with $z_0=0$, and $z_{i+1}=z_i^2$. We take zero values because we are only interested in the overhead costs.

```
Start = iterate 10000 z0
where
    iterate :: !Int !C -> (!Real,!Real)
    iterate i c
    |   i == 0    = Get c
    |   otherwise = iterate (i-1) (Sq c)
```

Figure 5.10 The test program.

The programs have been compiled using the Clean 1.0.2 distribution version on a Macintosh IIfx. For all applications stack checks and index checks were generated. Each generated application has been supplied with sufficient heap space to prevent interfering garbage collections, and sufficient stack space to prevent stack overflow. Timings were obtained by setting the application option ‘Show Execution Time’, and taking the average of ten runs.

	strict tuples	procedural encapsulation	data encapsulation
heap	710 Kb	1820 Kb	1185 Kb
stack	75 b	160 Kb	160 Kb
time	0.15 s.	0.30 s.	0.21 s.

Figure 5.11 The test programs and evaluation times.

In terms of execution time, the abstract objects generated by the procedural encapsulation method take 200% of the standard implementation, and objects generated by the data encapsulation method takes 140%. Both methods produce slower code, but this should be expected because the standard implementation associates abstract operations with the abstract state at compile time, so it has no runtime overhead costs. The encapsulation methods generate data structures in which the abstract operations are included, so there is heap consumption and heap traversal necessary to locate the operations.

The latter reason also explains the differences in heap use. In the first method, objects consume 256% of the standard implemented objects, and of the second method objects use 167%. The most dramatic difference between the encapsulation methods and the standard implementation is their stack consumption. The cause of this is largely determined by the fact that in both cases the data structures put the computation of the new abstract state in a lazy context, rather than a strict context. Figure 5.12 gives the remedy to these cases.

(a) The new local definition of CDEF of Figure 5.9*b*.

```
CDEF :: !(CSIG c) !CIN -> COUT
CDEF (CSIG c ops=:(COPS get sq)) GetIN
=   let! c1
    in   GetOUT c1
where    c1 = get c
CDEF (CSIG c ops=:(COPS get sq)) SqIN
=   let! c1
    in   SqOUT  (CDEF (CSIG c1 ops))
where    c1 = sq c
```

(b) The new definition of Sq of Figure 5.9*c*.

```
Sq :: !C -> C
Sq (CSIG c ops=:(COPS get sq))
=   let! c1
    in   CSIG c1 ops
where    c1 = sq c
```

Figure 5.12 The two changes to the encapsulation methods.

Figure 5.13 gives the new performances of the annotated programs.

	procedural encapsulation	data encapsulation
heap	1580 Kb	950 Kb
stack	150 b	150 b
time	0.28 s.	0.20 s.

Figure 5.13 The new performance figures after adding eager evaluation annotations.

The heap usage of the two programs has modestly decreased by 30% to 226% and 134% respectively, as well as the execution times to 187% and 133%, when compared with the standard implementation. However, the stack consumption has decreased dramatically!

5.6 Related work

This work was motivated from a practical need to create data structures that contain objects of the same signature, but with arbitrary (and irrelevant) internal representations in the settings of a functional programming language with a polymorphic Milner/Mycroft type system. We have shown one way to solve this problem by extending the type system with existential types. This direction has been taken by Mitchell and Plotkin (1985; 1988) who assign existential types to abstract data types. Other work in this direction is done by Pierce and Turner (1994) who explore the use of existential types to formalize object-oriented concepts. Their approach has been inspiring to both encapsulation methods presented in this chapter. From a type theoretical angle, Coquand and Paulin-Mohring (1988) have shown that existential types can be represented by inductively defined types. The relationship between their approach and the approach presented in this chapter requires further investigation. It should be mentioned however that our method is neither intended nor sufficiently general to be used as an 'implementation' of existential types in a polymorphic Milner/Mycroft type system with algebraic types. As an example, the following type does not fit into our procedural encapsulation method. This example is due to Sjaak Smetsers, a similar definition was given by Läufer and Odersky (1994).

```
::  F E.γ α β
=   Func (α -> β)
|   Comp (F Void α γ) (F Void γ β)
```

This type expresses that 'a function' from $\alpha \rightarrow \beta$ is either a 'real function' from $\alpha \rightarrow \beta$, or a composition of 'a function' from $\alpha \rightarrow \gamma$ and 'a function' from $\gamma \rightarrow \beta$.

5.7 Conclusions

In this chapter two multiple implementation data abstraction methods have been presented. The first method uses higher-order functions and currying to hide implementation details of data structures and can be applied in any functional language with a polymorphic Milner-Mycroft type system with algebraic data types. The second method requires an extended type system with existential types. The abstract objects that are created can have different internal implementations. In both methods abstract objects are first-class citizens. The abstraction method derives properly typed abstract objects if uniqueness types are involved. The emphasis of the first method is to demonstrate that in a functional language with a 'conventional' polymorphic Milner/Mycroft type system with algebraic types it is in principle possible to implement a flexible data abstraction. It gives a good account on the expressive power of functional programming languages. The emphasis of the second method is on readability, and it gives a more straightforward approach to data abstraction based on extending the type system.

Although the performance figures of abstract objects generated by the first method are roughly twice as high as the standard Clean abstract types, they can be a good alternative in a language with a type system without existential types. For efficiency reasons we included existential types in the type system of Clean. The test cases show that existential types indeed improve with respect to execution time and heap space. The stack consumption is equally high. Other reasons to incorporate existential types are that they have more expressive power, as the example in Section 5.6 shows, and they are easier to use. Both methods are less efficient than the standard Clean single implementation abstract type approach, but they are also more expressive. Still, for efficiency reasons one should combine both approaches in the language. We have seen that records are implemented less efficient than algebraic types. Further investigations are needed to see if the abstraction methods can be improved by program transformation techniques such as deforestation (Wadler, 1988), and partial evaluation of term graph rewriting programs (Koopman, 1992).

Acknowledgements

The author would like to thank Erik Barendsen, Sjaak Smetsers, and Marco Kesseler for discussions of the chapter. Erik Barendsen was especially helpful in providing background information on some type theoretical aspects. Simon Peyton Jones suggested to use existential types to solve the data hiding problem.