A New Look at Pattern Matching in Abstract Data Types

Palao Gostanza, Pedro
Ricardo Peña
Manuel Núñez

Departamento de Informática y Automática
Universidad Complutense de Madrid, Spain

e-mail: {ecceso,ricardo,manuelnu}@dia.ucm.es

Abstract

In this paper we present a construction smoothly integrating pattern matching with abstract data types. We review some previous proposals [19, 23, 20, 6, 1] and their drawbacks, and show how our proposal can solve them. In particular we pay attention to equational reasoning about programs containing this new facility. We also give its formal syntax and semantics, as well as some guidelines in order to compile the construction efficiently.

1   Introduction

Most modern functional programming languages allow the programmer to define his/her own types by using the *algebraic type* facility. An algebraic type is a set of terms freely generated by a collection of special operations called *constructors*. For instance, the complex numbers type can be defined as follows:

1   **data** Complex $=$ *Cart* Real Real

where $Cart :: Real \rightarrow Real \rightarrow Complex$ is a tupling constructor taking two real numbers as parameters and producing a complex number.

Constructors play also a second role: they may appear in *patterns* in the left hand side of equations. Its role here is to access the value components. In this case, the constructor in fact acts as a *destructor*. In the complex numbers example we can define an *add* operation as follows:

2   *add* $(Cart\ r_1\ i_1)\ (Cart\ r_2\ i_2) =$
3        $Cart\ (r_1 + r_2)\ (i_1 + i_2)$

When more than one constructor exists, functions can be defined by using several equations having different left hand sides. Usually, these equations are tried sequentially, and the first equation whose left hand side matches the patterns is the one applied. In this case, constructors are acting not only as destructors but also as *checking functions*. For instance, we can define the add operation for natural numbers as:

4   **data** Nat $=$ *Zero* $\mid$ *Succ* Nat

5   *add* $(Succ\ n_1)\ n_2 = Succ\ (add\ n_1\ n_2)$
6   *add*   *Zero* $n_2 = n_2$

The combination of algebraic types and pattern matching has resulted in an easy and expressive way of defining types as well as of defining operations manipulating them. But it also has drawbacks. The first of them is the positional notation of the constructor components. The introduction of records and the extension of patterns in order to use them almost solves this problem.

The worst one is the difficulty of using pattern matching in abstract data types. The complex numbers example illustrates well this problem. With the chosen representation, we could write the *add* function in a concise way, but this representation is not appropriate to define a product operation. For that purpose, it would have been better to define complex numbers by using polar representation:

7   **data** Complex $=$ *Pole* Real Real
8   *product* $(Pole\ m_1\ a_1)\ (Pole\ m_2\ a_2) =$
9        $Pole\ (m_1 * m_2)\ (a_1 + a_2)$

The programmer has to make a (rather difficult) choice between these two representations, and cannot have the advantages of both at the same time.

In functional languages —as well as in most modern programming languages— the decision can be delayed by using an abstract data type hiding the representation. We could define a set of abstract operations for complex numbers as follows:

10   **abstype** Complex
11   *cart*, *pole* :: Real $\rightarrow$ Real $\rightarrow$ Complex
12   *real*, *imaginary* :: Complex $\rightarrow$ Real
13   *magnitude*, *angle* :: Complex $\rightarrow$ Real

With these operations we can elegantly and symmetrically define both *add* and *product* as follows:

14   *add* $z_1\ z_2 = cart\ (real\ z_1\ +\ real\ z_2)$
15                   $(imaginary\ z_1\ +\ imaginary\ z_2)$
16   *product* $z_1\ z_2 = pole\ (magnitude\ z_1\ *\ magnitude\ z_2)$
17                   $(angle\ z_1\ +\ angle\ z_2)$

Now the definition is more wordy because we have lost pattern matching. Let us note that the destruction implicitly performed in the left hand side, now must be explicitly performed in the right hand side, by applying the adequate functions. This transformation is even worse when applied

to functions defined by using several rules, because it is necessary to join them in a unique rule, and then distinguish between the different cases by means of checking functions.

In general, whenever a type is transformed into an abstract type, the implementation is hidden and so the possibility of using patterns is lost. In the complex numbers example, the constructors which would be useful for an ADT user are also adequate for a possible implementation. But generally this is not the case. For instance, sets are usually implemented by means of AVL trees, priority queues are implemented by using binomial trees, and so on. Useful patterns for priority queues could be one for checking if the queue is empty, and one for extracting the first element.

This problem has appeared before in programming languages. When a program is complex enough, it is necessary to split it into several modules, some of them implementing ADT's, and hiding the representations. Pattern matching can only be used inside the modules implementing abstract data types. Thus, a pleasant mechanism such as pattern matching has a limited usability.

In this paper we propose a new mechanism to make pattern matching cohabit with data abstraction. It is named *active destructors*. Its main characteristic is that pattern matching and algebraic types are separated. So, active destructors are not directly related to ADT's implementations.

The rest of the paper is organized as follows. In Section 2 we review earlier proposals in this area: *laws* [21, 19, 20], *views* [23, 6] and *abstract value constructors* [1]. The last one is part of the New Jersey's SML implementation [2]. The other ones were respectively parts of Miranda [21] and Haskell [9]. Because of equational reasoning problems, they were not included in the new versions of these languages [10]. In Section 3 active destructors are introduced by means of examples. In Section 4 we give a sketch of our syntax and a denotational semantics for it. In Section 5 we show how to perform equational reasoning by using the new mechanism. This has been the main drawback of many of the previous proposals. In Section 6 we sketch some possible ways of compiling our construction, while in Section 7 we present our conclusions. A thorough description of our proposal, including all technical details, is contained in [14].

2 Previous Work

In this section we review several previous proposals trying to introduce the idea of abstraction into algebraic types.

Miranda laws [19] were the first proposal. The aim there was the definition of non free types. The equations relating the data constructors —called *laws*— are interpreted as rewriting rules trying to reach a normal form. These non free data constructors can be used both for pattern matching and for value construction. For example, the following algebraic type with laws implements complex numbers in polar form

18 **data** Complex = *Pole* Real Real
19 *Pole* 0.0 $a \Rightarrow Pole$ 0.0 0.0
20 *Pole* m $a \Rightarrow Pole$ $(-m)$ $(a + \pi)$, $m < 0.0$
21     $\Rightarrow Pole$ m $(a + 2.0 * \pi)$, $m > 0.0 \land a < 0.0$
22     $\Rightarrow Pole$ m $(a - 2.0 * \pi)$, $m > 0.0 \land a \geq 2.0 * \pi$

In this example, the added laws keep values in canonical form.

Laws are compiled separating the underlying free type from the laws themselves. These define a *normalizing func-*

tion for each of the constructors. In the previous example, we would have the type

23 **data** Complex = *Pole* Real Real

and the function

24 *pole* 0.0 a = *pole* 0.0 0.0
25 *pole* m a = *pole* $(-m)$ $(a + \pi)$, $m < 0.0$
26   = *pole* m $(a + 2.0 * \pi)$, $m > 0.0 \land a < 0.0$
27   = *pole* m $(a - 2.0 * \pi)$, $m > 0.0 \land a \geq 2.0 * \pi$
28 *pole* m a = *Pole* m a

When a constructor of a lawful type is used in the right hand side of an equation, it is replaced by the corresponding function; when it is used in a pattern, the constructor remains unaffected.

From the pattern matching point of view, laws are not as convenient as it would be desired, because normalization takes place before matching. For instance, if one wants to implement complex numbers in such a way that both a pattern for the cartesian form and another one for the polar form could be used, then we need to define an algebraic type with two constructors and a law relating them; for example

29 **data** Complex = *Cart* Real Real
30 | *Pole* Real Real
31 *Pole* m $a \Rightarrow Cart$ $(m * cos$ $a)$ $(m * sin$ $a)$

We have chosen ($Cart$ x y) as the canonical representation of a complex number. Should we define the product as follows

32 *prod* ($Pole$ m_1 a_1) ($Pole$ m_2 a_2) =
33   *Pole* $(m_1 * m_2)$ $(a_1 + a_2)$

and should we try to evaluate the expression ($prod$ z_1 z_2), we would obtain a matching error, because z_1 and z_2 would be first normalized to the cartesian representation.

Besides these problems, laws were excluded from Miranda due to well known problems in equational reasoning. Thompson solves some of these problems in [20] by distinguishing, for reasoning purposes, between the two uses of a data constructor: as a pattern and as a normalizing function. But this solution leads to the unpleasant fact that for reasoning about a program, it is necessary to perform the same transformation that the compiler would do (as described above). In a later work [6], this fact is made explicit in the program text: the constructor and the normalizing function receive different names. Even though constructors can only be used in patterns, some abstraction is lost, because the user can still access the values implementation.

The main drawback of all the work related to laws is that the choice of possible implementations for an ADT is severely limited, because a unique canonical representative for each abstract value must exist. For instance, the only possible implementation of sets is isomorphic to ordered lists.

Wadler's *views* [23] are the most versatile proposal trying to solve the main problem of this paper. In contrast to the traditional solution of hiding the implementation, Wadler proposes to achieve abstraction, i.e. independence of the chosen representation, by showing as many implementations as possible. Each implementation is a *view* of the type. There is a distinguished one corresponding to the actual representation of the type. For each other view, two functions, called

in and **out**, must be defined in order to transform values to and from the distinguished view. These functions are inverse of each other. Because all implementations are visible, pattern matching over each view can be performed. For example, the complex numbers would have the following two implementations:

```
34  data Complex  =  Cart Real Real
35  view Complex  =  Pole Real Real
36     in (Cart r i)  =  Pole (sqrt (r² + i²)) (atan2 r i)
37     out (Pole m a)  =  Cart (m * cos a) (m * sin a),
38        if (m == 0.0 ∧ a == 0.0) ∨
39           (0.0 < m ∧ 0.0 ≤ r < 2.0 * π)
```

In this case, the distinguished implementation is the cartesian one; in the case of the polar implementation, functions must be defined for doing transformations from and to the cartesian one. The **out** function transforms a complex number in polar representation to one in cartesian representation, while the **in** function does the opposite. As these functions are inverse of each other, they must be injective, hence the guard appearing in lines 38 and 39.

But this guard leads to non trivial problems. For instance, which is the value of an expression like $(Pole\ 0.0\ 1.0)$? In [23] it is not specified the value of such expressions, but they are not forbidden. In general, the injectivity constraint for these functions has been considered very restrictive (see [6]), but it is necessary in order to make equational reasoning feasible. Without this constraint, implicit equations may appear. For instance, if we removed the guard in the previous example, we would introduce the equation:

$$\forall a \forall b\ Pole\ 0.0\ a = Pole\ 0.0\ b$$

These problems have been reviewed in [6], where they are solved by using the same idea as Thompson used in [20] for laws. They argue that the problems come again because of the dual role played by views constructors: as patterns and as value constructor functions. They propose to remove the second role. As a consequence of this, only the **in** function is required and it is not forced to be injective. The authors regard a view as a bundle of case recognition and component selection functions.

Unfortunately, they insist in transforming from the representation to the view *before* the matching is performed. The reason for that seems to be exclusively a compilation issue. As a consequence, in the definition of a function, only constructors belonging to the same view can be used. This fact imposes severe limitations in accessing an ADT. Also, for a view to be useful when defining functions, it must be "total" in the sense of providing enough observations about the abstract value. For this reason, all the views examples shown in the mentioned papers could also work as alternative implementations of the ADT. Double queues are a typical example where these limitations appear clearly, because there must exist a way of accessing the first element as well as accessing the last element of the queue. In this case, two constructors can be defined: $(First\ e\ q)$ accessing the first element and $(Last\ e\ q)$ accessing the last one. But each of them must be defined in different views because *First* and *Last* are not free between themselves. So, a function like

```
40  bothEnds (First True q)  =  some computations
41  bothEnds (Last False q)  =  some other computations
42  bothEnds q  =  default computations
```

accessing both extremes could not be defined. Of course, a constructor like $(FirstLast\ fe\ le\ q)$ accessing simultaneously both extremes could be defined. But this pattern would not match queues with one element, and a new pattern $(SingletonQueue\ e)$ should be used in these cases. Using these constructors, the *bothEnds* function could be written, but it would be too verbose.

We think that these limitations are a consequence of trying to make *evolve* the constructor concept instead of starting the problem from scratch. In short, the way to get abstraction is not to show all possible implementations.

The last proposal we review is *abstract value constructors* (AVC) [1]. An AVC is a special function such that the right hand side of its definition is an expression formed exclusively by constructors and variables. This restriction makes possible to use AVCs both as normal functions and as patterns. There are some special AVCs —called projections— which may have variables in the right hand side not appearing in the left hand side, but they can be used only in patterns, and they can be considered as functions giving partial observations of the ADT. AVCs do not suffer from equational reasoning problems but their expressive power is very limited due to their unability to perform computations. Although we conceived our proposal before getting knowledge of AVCs, our active destructors can be seen as projection functions where computations can be performed.

3 Active Patterns

Our solution to achieve a smooth integration between ADT's and pattern matching comes from considering the problem from the very beginning. What is the role of pattern matching in the ADT philosophy? In our opinion, patterns must provide a comfortable way of *observing* abstract properties, because construction of values is a different matter. ADT's usually split the operations of a type into several disjoint groups: value constructors in one hand, and checking, accessing or modification functions —in short, observer functions— in the other hand. In general, constructors and observers need not to be dual to each other. A typical example of this are FIFO queues: while the construction operation inserts an element at the end, the observation one accesses the element at the front.

An *active pattern* will play both roles of checking and of observer function. Additionally, an active pattern will be able to do as many computations as necessary for playing these roles. The execution of an active pattern starts by checking the value in order to determine whether it satisfies the pattern applicability condition; if this is the case, a code computing the observations made by the pattern is executed. These computations are done at matching time.

3.1 First Example: Complex Numbers

We will start the description of our proposal with the complex numbers example. The relevant observations of this type are: *real, imaginary, magnitude* and *angle* (see code lines 12 and 13). Each of these operations may be used in pattern matching, and we want their syntactic use to coincide with that of constructors. An operation implemented in such a way will be called *active destructor* (*AD*) and its name will be capitalized. For example, the definition of the *isReal* function, which indicates whether a complex number

is a real one, would be:

43   $isReal\ (Imaginary\ i)\ =\ (i\ ==\ 0.0)$

The *Imaginary* AD, with one argument, defines a pattern which matches a complex number z. If the argument of *Imaginary* is a variable i, it will be bound to the imaginary part of z; if the argument is a pattern p then the matching process will go on, trying to match the imaginary part of z with p. So, the *isReal* function could also be defined as follows:

44   $isReal\ (Imaginary\ 0.0)\ =\ $ True
45   $isReal\ _\ \qquad\qquad\quad =\ $ False

In the previous definition, we only had to extract the imaginary part of a complex number, but when dealing with operations like *add*, we have to perform two observations on the same complex number: its real and its imaginary parts. We use the @ symbol[1] to denote the conjunction of patterns. A value matches a pattern of the form $p_1@p_2$ if and only if it matches p_1 and p_2. Moreover, the produced bindings are the union of those coming from the matching of p_1 together with those coming from the matching of p_2. Then, the *add* function can be written as:

46   $add\ (Real\ r_1)@(Imaginary\ i_1)$
47       $(Real\ r_2)@(Imaginary\ i_2)\ =$
48       $cartesian\ (r_1 + r_2)\ (i_1 + i_2)$

where *cartesian* is a value constructor function of the Complex ADT. Patterns using ADs and the @ symbol are called *active patterns*.

If we implement complex numbers by using a cartesian representation, the definition of the *Imaginary* AD will be:

49   $Imaginary\ i$ **match** $Cart\ r\ i$

This definition is syntactically similar to that of a function: the name of the AD, its arguments, the keyword **match** (instead of the = symbol), and its definition. However, everything behaves in the opposite way. The right hand side does not define the value of an AD. Instead, it is a pattern defining which values match the AD. In this example, any complex number will match. This pattern is called a *condition pattern*. Let us note that the argument of *Imaginary* is an *output* one, which means that it is the result of a computation instead of an argument for a computation. So, in an AD, data flow from the right hand side to the left one: the datum matching the AD is the input value, and it may match the condition pattern; this matching extracts parts, which are used to compute the values of the AD arguments. Following with our example, the previous definition can be written using an anonymous variable, because the real part of the number (i.e. r) is not needed:

50   $Imaginary\ i$ **match** $Cart\ \_\ i$

These ideas are presented more clearly in the definition of the *Magnitude* AD, which performs a computation:

51   $Magnitude\ (sqrt\ (r^2 + i^2))$ **match** $Cart\ r\ i$

---

[1]This symbol is used in some funtional languages to denote *as* patterns. We generalize its semantics, because a variable is a pattern. Let us note that our extension adds expressive power only to ADs. When it is applied to patterns built by usual algebraic constructors, the pattern will be unfolded until either one of the arguments is a variable or a pattern which always fails.

Let us note that the argument of *Magnitude* is an expression built from variables appearing on the right hand side of the definition. In general, the computations performed by an AD may be as complex as needed but, in order to simplify the notation, we allow the use of a **where** clause where the auxiliary definitions can be written:

52   $Magnitude\ m$ **match** $Cart\ r\ i$
53   **where** $m\ =\ sqrt\ (r^2 + i^2)$

If we implement complex numbers in polar form, these ADs can be reimplemented, so that functions like *add* remain correct. That is what we would expect from a system providing data abstraction. Moreover, it is not necessary to normalize the internal representation because ADs can normalize the returned value:

54   $Magnitude\ (abs\ m)$ **match** $Pole\ m\ \_$
55   $Angle\ (norm\ m\ a)$ **match** $Pole\ m\ a$
56   **where** $norm\ m\ a$
57        $=\ 0.0,\quad m == 0.0$
58        $=\ norm\ (-m)\ (a + \pi),\quad m < 0.0$
59        $=\ norm\ m\ (a + 2.0 * \pi),\quad m > 0.0 \wedge a < 0.0$
60        $=\ norm\ m\ (a - 2.0 * \pi),\quad m > 0.0 \wedge a \geq 2.0 * \pi$
61        $=\ a$

3.2   Second Example: FIFO Queues

Now we present a more complex example, which illustrates the usual fact that constructor and observer functions are not dual to each other: FIFO queues. Its signature is:

62 **abstype** Queue α
63   *empty* :: Queue α
64   *enqueue* :: Queue $\alpha \rightarrow \alpha \rightarrow$ Queue α
65   *isEmpty* :: Queue $\alpha \rightarrow Bool$
66   *first* :: Queue $\alpha \rightarrow \alpha$
67   *dequeue* :: Queue $\alpha \rightarrow$ Queue α

Clearly, *empty* and *enqueue* are constructor functions, so they must be implemented as normal functions. The *isEmpty* function is a checking operation, so it may be a good candidate for an AD, which we will call *Empty*. We use the classic amortized constant time representation of a queue by means of two lists:

68 **data** $Queue\ \alpha\ =\ Queue\ [\alpha]\ [\alpha]$
69 $empty\ =\ Queue\ [\ ]\ [\ ]$
70 $enqueue\ (Queue\ [\ ]\ [\ ])\ x\ =\ Queue\ [x]\ [\ ]$
71 $enqueue\ (Queue\ fs\ \ ls)\ x\ =\ Queue\ fs\ (x : ls)$
72 $Empty$ **match** $Queue\ [\ ]\ [\ ]$

The implementations of *first* and *dequeue* are more subtle. On the one hand, both of them have a checking component, because they are not defined for empty queues. But from the ADT's point of view, the *first* function is an accessing one, while *dequeue* is a modification function. According to the *dequeue* role we consider more important —modification or checking—, it may be implemented either as a function or as an active destructor. Because we want to show as many ADs as possible, we will choose the second alternative:

73 $First\ x$ **match** $Queue\ (x : \_)\ \_$
74 $Dequeue\ (Queue\ fs\ ls)$ **match**

```
75        Queue (_ : (fs@(_ : _)))  ls
76  Dequeue (Queue (reverse ls) [ ])    match
77        Queue [_] ls
```

Let us remark some important points in this example. First, the definition of an AD C may need several equations, as it happens to functions. A value matches C, if it matches one of the condition patterns appearing in the right hand sides. Second, it is important to emphasize that condition patterns used in the right hand sides are not completely independent of each other. For instance, it would be nonsense that a particular queue matched, at the same time, the condition pattern of *Empty* and the one of *Dequeue*.

A second way of implementing these ADT operations is by recognizing the strong relation between them: sometimes it is useful to access the first element and to remove it at the same time. The following code defines a *Dequeue'* AD, with two arguments, implementing this idea:

```
78  Dequeue' x q   match   Queue (x : (fs@(_ : _)))  ls
79       where q = Queue fs ls
80  Dequeue' x q   match   Queue [x] ls
81       where q = Queue (reverse ls) [ ]
```

But we could also implement *Dequeue'* from *First* and *Dequeue* ADs

```
82  Dequeue' x q   match   (First x)@(Dequeue q))
```

The decision to implement an abstract data type operation either by using a function or by an AD is not *irrevocable*. For instance, if the Queue type had been defined only by using functions, we still could define ADs on top of them:

```
83  Empty    match   q,  if isEmpty q
84  First (first q)   match   q,  if not (isEmpty q)
85  Dequeue (dequeue q)   match   q,  if not (isEmpty q)
```

This example also shows the possibility of adding guards in the definition of an AD. A value v matches an AD C having a guard, if v matches the condition pattern of a rule of C and also its associated guard is satisfied. As it happens to function definitions, a guard can use both the variables bound by the condition pattern and the auxiliary variables defined in the additional **where** clause.

As another feature, we can also define functions on top of ADs:

```
86  isEmpty Empty  =  True
87  isEmpty _      =  False
88  first (First x)  =  x
89  dequeue (Dequeue q)  =  q
```

These examples show that ADs can be defined and used anywhere in the program text. It is not required to associate an AD to the definition of a certain abstract data type, unless it uses the representation. This freedom allows the programmer to create auxiliary ADs for solving a problem in the same way as he/she would create any needed auxiliary functions.

To show the elegance of the code obtained by programming using ADs, we present a function which computes the breadth-first traversal of a binary tree:

```
90  data Tree α = EmptyTree
91                  | Node (Tree α) α (Tree α)
```

```
92  breadthSearch :: Tree α → [α]
93  breadthSearch t = loop (enqueue empty t)
94     where
95        loop (First EmptyTree)@(Dequeue q) =
96           loop q
97        loop (First (Node t₁ a t₂))@(Dequeue q) =
98           a : loop (enqueue (enqueue q t₁) t₂)
99        loop Empty = [ ]
```

Let us note that inside the *First* AD an algebraic pattern over binary trees appears. That means that ADs can be used in the same way like algebraic constructors to create patterns without any kind of restrictions. In Section 6 we will show how these patterns can be efficiently compiled.

3.3   Types

If we wish to integrate active destructors in a typed functional language, it is important to know the type assigned to them. If we consider *Imaginary* and *Magnitude* as conventional constructors, their type will be $Real \to Complex$. On the one hand, this is a reasonable type, because if *Imaginary* is "applied" to a pattern of type Real, we obtain a pattern of type Complex. On the other hand, this is an erroneous type because *Imaginary* is not a function, so it cannot be used in the right hand side of an expression to build a complex number. There exists another problem with typing: How can a user infer the number of arguments of *Imaginary* from its type? In the case of usual constructors such as *Cart* or *Pole*, there is no problem because their arities are equal to the number of arguments to which they must be applied in order to obtain a non functional type. But this rule does not work for ADs because we could define ADs matching functional values. For example, an AD *Not* defined as follows

```
100 Not   match   f,  if ¬(f True) ∧ (f False)
```

matches all the functional values of type $Bool \to Bool$ behaving as the negation function. Because *Not* has no arguments, its type is the same as that of the values which matches, that is, $Not :: Bool \to Bool$. So, even though the types of *Imaginary* and *Not* have the same shape, the former has an argument while the latter has no arguments.

Therefore, we think that a new notation for the type of a destructor is needed. This notation must distinguish between destructors and functions and must make explicit that a destructor must be totally "applied" to patterns in order to built a new pattern, that is, it must contain explicit information about the arity of the pattern. The proposed notation encloses the type between $\langle \rangle$, and replaces arrows by commas. Then, the type of *Imaginary* and *Magnitude* would be $\langle Real, Complex \rangle$, while the type of *Not* is $\langle Bool \to Bool \rangle$. Note that our proposed notation clearly indicates that *Imaginary* has one argument, while *Not* has no arguments and matches functional values. Finally, the signature of the Queue abstract data type by using functions and active destructors would be:

```
101 abstype Queue α
102    empty :: Queue α
103    enqueue :: Queue α → α → Queue α
104    First :: ⟨α, Queue α⟩
105    Dequeue :: ⟨Queue α, Queue α⟩
```

and the type of the *Dequeue'* AD would be

```
106 Dequeue' :: ⟨α, Queue α, Queue α⟩
```

Once presented how to assign type to ADs, it seems natural to write higher order functions manipulating ADs. Let us see an example. ADs without arguments are equivalent to predicates with one argument. For instance, in the Queue example we showed how the *Empty* AD and the *isEmpty* function could be defined on top of each other. This kind of transformations can be generalized by using the following pair of functions:

107 $pred2Dest :: (\alpha \to Bool) \to \langle \alpha \rangle$
108 $pred2Dest\ p\ =\ C$
109   **where** C **match** x, **if** $p\ x$
110 $dest2Pred :: \langle \alpha \rangle \to (\alpha \to Bool)$
111 $dest2Pred\ C\ =\ \lambda x\ \to$ **case** x **of**
112                                      $C\ \to\ True$
113                                   $\_\ \to\ False$

These transformation schemes can be applied to the *Not* AD to obtain a predicate *isNot*

114 $isNot\ =\ dest2Pred\ Not$

3.4   Other Examples

Sets constitute an important example of ADT. We have not used them in order to introduce ADs because its behavior is too *symmetric*: constructor and access operations are dual to each other. Using ADs, the specification of these operations is:

115 **abstype** Set α
116     $emptySet :: $ Set α
117     $addSet :: \alpha\ \to\ Set\ \alpha\ \to\ Set\ \alpha$
118     $EmptySet :: \langle$Set$\ \alpha \rangle$
119     $ChooseSet :: \langle \alpha,$ Set $\alpha,$ Set $\alpha \rangle$

The *emptySet* and *addSet* functions respectively create an empty set and add an element to a set, while *EmptySet* is an AD matching only the empty set. The *ChooseSet* AD matches sets having at least one element, and it returns any of the elements in the set together with the previous set with this element removed. It is important to note that *ChooseSet* is non deterministic, and so it cannot be equationally specified. Any valid implementation of this operation will be deterministic but indeterminate. Care must be taken with functions using this operation. For example, the *map* function over sets causes no problem:

120 $mapSet\ f\ EmptySet\ =\ emptySet$
121 $mapSet\ f\ (ChooseSet\ e\ s)\ =$
122     $addSet\ (f\ e)\ (mapSet\ f\ s)$

We can also define a *fold* function over sets

123 $foldSet\ (\oplus)\ i\ EmptySet\ =\ i$
124 $foldSet\ (\oplus)\ i\ (ChooseSet\ e\ s)\ =$
125     $e\ \oplus\ (foldSet\ (\oplus)\ i\ s)$

But in this case, the binary operation $(\oplus)$ must verify

$$x \oplus (y \oplus z) = y \oplus (x \oplus z)$$

and i must be a right neutral element of this operation. In fact, *mapSet* can be defined from *foldSet*

126 $mapSet\ f\ =\ foldSet\ (addSet \circ f)\ emptySet$

because $addSet \circ f$ and $emptySet$ satisfy the previous conditions.

As it happens with views, we could define some apparently strange constructions using ADs. For example, the **as** patterns can be defined as:

127 $(@) :: \langle \alpha, \alpha, \alpha \rangle$
128 $(x\ @\ x)$ **match** x

3.5   Other facilities

Modern functional programming languages usually have special patterns to deal with predefined types. The patterns $(n + k)$, which allow to define recursive functions over the natural numbers, are the most classical example. Its definition, using the AD notation, is (please ignore the type declaration until the end of the paragraph):

129 $(+) :: \langle$Int, Int$\downarrow$, Int$\rangle$
130 $(n\ +\ k)$ **match** m,   **if** $m \geq k$
131   **where** $n\ =\ m - k$

A natural number m matches $(n+k)$ if $m \geq k$; in this case, n is bound to $m-k$. Obviously, the previous definition is incorrect, because k behaves as an input parameter. But this is precisely the main feature of this kind of patterns: they are a whole family of patterns, parameterized by an argument. The extension of ADs in order to cope with this kind of definitions is easy: A compiler can detect whether an argument is an input or an output one just by looking whether it has an associated definition in the condition pattern or in the **where** declaration. In this case, the variable n has a definition in the right hand side, while k has not. Thus, there are two input arguments (the implicit one matching m, and k) and an output argument: n. Now, we have to enhance the type notation to cope with this extension. In order to do this, we will add $\downarrow$ after the type of the input arguments. The type in code line 129 reflects this extension.

The possibility of defining this kind of patterns is important. When the predefined $(n + k)$ patterns are combined with overloading mechanisms many problems appear (they are described in detail in [5]). A class system, where ADs could be members, solves these problems:

132 **class** Num α **where**
133     $(+),\ (-),\ (*),\ \ldots :: \alpha \to \alpha \to \alpha$
134     $\cdots$
135     $(+),\ (-),\ (*),\ \ldots :: \langle \alpha, \alpha \downarrow, \alpha \rangle$
136     $\cdots$
137     $((m - k)\ +\ k)$ **match** m,   **if** $m \geq k$
138     $\cdots$

Let us note that the overloading of arithmetic operators both as functions and as ADs is easily solvable, because they are always used in different syntactic contexts.

4   Syntax and Semantics

In this section we formalize the abstract syntax and the semantics of programs using active destructors.

In order to achieve the first objective, we must define an abstract syntax for a small functional language together with some contextual restrictions, while for achieving the second one we must give a denotational semantics assigning

formal meaning to the terms generated by the abstract syntax. Even though this semantics will define our language as a lazy one, it is very easy to adapt it for an eager language. In fact, none of the needed modifications affects active destructors.
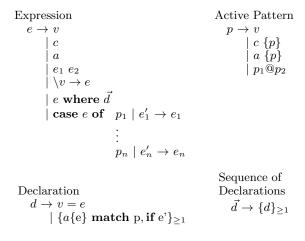
Expression
$$e \rightarrow v$$
$$| \; c$$
$$| \; a$$
$$| \; e_1 \; e_2$$
$$| \; \backslash v \rightarrow e$$
$$| \; e \; \textbf{where} \; \vec{d}$$
$$| \; \textbf{case} \; e \; \textbf{of} \quad p_1 \; | \; e'_1 \rightarrow e_1$$
$$\vdots$$
$$p_n \; | \; e'_n \rightarrow e_n$$

Active Pattern
$$p \rightarrow v$$
$$| \; c \; \{p\}$$
$$| \; a \; \{p\}$$
$$| \; p_1 @ p_2$$

Declaration
$$d \rightarrow v = e$$
$$| \; \{a\{e\} \; \textbf{match} \; p, \textbf{if} \; e'\}_{\geq 1}$$

Sequence of Declarations
$$\vec{d} \rightarrow \{d\}_{\geq 1}$$

Figure 1. Abstract Syntax

In Figure 1 we give the abstract syntax of a small functional language including active destructors and conditional expressions for pattern matching. We have not included the declaration of abstract and algebraic types, because they are not necessary for defining the semantics of active destructors.

We have four different syntactic domains:expressions, patterns, declarations, and sequences of declarations. We also will use objects belonging to three syntactic domains having an infinite number of symbols. In Figure 2 we show these domains, and the variables ranging over them.

Domain	Ranges over	Contents
V	$v, v', v_0, v'_0, \ldots$	Variables
C	$c, c', c_0, c'_0, \ldots$	Algebraic Constructors
A	$a, a', a_0, a'_0, \ldots$	Active Destructors
E	$e, e', e_0, e'_0, \ldots$	Expressions
P	$p, p', p_0, p'_0, \ldots$	Active Patterns
D	$d, d', d_0, d'_0, \ldots$	Declaration
$\vec{D}$	$\vec{d}, \vec{d'}, \vec{d_0}, \vec{d'_0}, \ldots$	Sequence of Declarations

Figure 2. Syntactic Domains

We assume that the domains associated with algebraic constructors and with active destructors are the disjoint infinite union of subdomains

$$C = \cup_{i \geq 0} C_0 \qquad A = \cup_{i \geq 0} A_0.$$

where the index of each subdomain denotes the arity of the elements belonging to it. The C_0 subdomain contains the boolean values: *False* and *True*. We also need the predefined constructors $Fail \in C_0$ and $OK_i \in C_i$ (for all $i \geq 0$) indicating the success or failure in matching an active destructor.

We will also need the following contextual restrictions:

- Patterns may not contain two ocurrences of the same variable (i.e. they must be linear).

- If $c \in C_i$, then c must be applied to i patterns in order to get another pattern. The same restriction applies to active destructors $a \in A_i$.

- A variable (or an active destructor) symbol may not appear twice in the left hand side of a sequence of declarations.

- For any **match** sequence of declarations belonging to the same active destructor definition, the name of this destructor and its arity must be kept.

- The definition of an active destructor $a \in A_i$ must be done by using i output arguments.

We will work with a unique semantic domain, named *Val*, which is recursively defined as follows:

$$Val = Func \oplus Cnst_0 \oplus Cnst_1 \oplus \cdots$$
$$Func = [Val \rightarrow Val]_\perp$$
$$Cnst_0 = (C_0)_\perp$$
$$Cnst_1 = (C_1 \times Val)_\perp$$
$$Cnst_2 = (C_2 \times Val \times Val)_\perp$$
$$\cdots$$

The coalesced disjoint sum of domains, where the domains bottoms are coalesced into only one is denoted by $\oplus$; the disjoint product of domains, where a new bottom is added, is denoted by $\times$; given a domain D, its lifted domain, where we add a new bottom, is denoted by $D_\perp$. The $Cnst_i$ subdomains are lifted in order to distinguish between a totally undefined value and the value of a constructor applied to undefined values. This fact implies that the semantics of our language will be lazy.

We also need the semantic domain for declaration environments, which allows us to assign a meaning to each variable and to each active destructor

$$Env = V + A \rightarrow_{\text{fin}} Val$$

This definition shows that active destructors have a meaning, while algebraic constructors have (almost) direct representatives in the *Val* domain. The last semantic domain we need is the subdomain of the boolean values

$$Bool = \{True, False, \perp\} \subset Cnst_0.$$

The expression x **in** *Val* will denote the inclusion in *Val* of an element x, belonging to any of the components of the coalesced disjoint sum (for example $x \in Func$). For each syntactic domain Γ shown in Figure 1, we define a semantic function transforming elements of Γ into elements of one of the semantic domains.

$$\mathcal{V} \colon E \rightarrow Env \rightarrow Val$$
$$\mathcal{D} \colon D \rightarrow Env \rightarrow Env$$
$$\vec{\mathcal{D}} \colon \vec{D} \rightarrow Env \rightarrow Env$$
$$\mathcal{AP} \colon P \rightarrow Env \rightarrow Val \rightarrow Bool$$
$$\mathcal{AE} \colon P \rightarrow Env \rightarrow Val \rightarrow Env$$

The $\mathcal{V}$ function computes the value of an expression (its definition is given in Figure 5). The $\mathcal{D}$ function produces an environment, having a unique binding, from the definition of an element, while the $\vec{\mathcal{D}}$ function is the generalization of $\mathcal{D}$

$$\text{apply} :: Val \to Val \to Val$$
$$\text{apply } x_1 \ x_2 = \begin{cases} f \ x_2, & x_1 = f \text{ in } Func \text{ in } Val \\ \bot, & \text{otherwise} \end{cases}$$

$$\text{is}_c^n :: Val \to Bool$$
$$\text{is}_c^n \ x = \begin{cases} True, & x = (c, y_1, \ldots, y_n) \text{ in } Cnst_n \text{ in } Val \\ False, & x = (c', y_1, \ldots, y_m) \text{ in } Cnst_n \text{ in } Val \\ & \wedge \ (c' \neq c \vee m \neq n) \\ \bot, & \text{otherwise} \end{cases}$$

$$\text{sel}_i^n :: Val \to Val$$
$$\text{sel}_i^n \ x = \begin{cases} x_i, & x = (c, x_1, \ldots, x_n) \text{ in } Cnst_n \text{ in } Val \\ \bot, & \text{otherwise} \end{cases}$$

$$\pm :: Env \to Env \to Env$$
$$(\rho_1 \pm \rho_2)(v) = \begin{cases} \rho_2(v), & v \text{ belongs to the } \rho_2 \text{ domain} \\ \rho_1(v), & \text{otherwise} \end{cases}$$

$$+ :: Env \to Env \to Env$$
$$\rho_1 + \rho_2 = \begin{cases} \rho_1 \pm \rho_2, & \rho_1 \text{ and } \rho_2 \text{ are disjoint domains} \\ \bot, & \text{otherwise} \end{cases}$$

Figure 3. Auxiliary semantic functions

$$\mathcal{AP}[\![v]\!]\rho x = True$$
$$\mathcal{AP}[\![c \ p_1 \cdots p_n]\!]\rho x = \text{is}_c^n(x) \wedge \mathcal{AP}[\![p_1]\!]\rho x_1 \wedge \cdots \wedge \mathcal{AP}[\![p_n]\!]\rho x_n$$
$$\text{where } x_i = \text{sel}_i^n(x)$$
$$\mathcal{AP}[\![a \ p_1 \cdots p_n]\!]\rho x = \mathcal{AP}[\![OK_n \ p_1 \cdots p_n]\!]\rho(\text{apply } \rho(a) \ x)$$
$$\mathcal{AP}[\![p_1@p_2]\!]\rho x = \mathcal{AP}[\![p_1]\!]\rho x \wedge \mathcal{AP}[\![p_2]\!]\rho x$$

$$\mathcal{AE}[\![v]\!]\rho x = [x/v]$$
$$\mathcal{AE}[\![c \ p_1 \cdots p_n]\!]\rho x = \mathcal{AE}[\![p_1]\!]\rho x_1 + \cdots + \mathcal{AE}[\![p_n]\!]\rho x_n$$
$$\text{where } x_i = \text{sel}_i^n(x)$$
$$\mathcal{AE}[\![a \ p_1 \cdots p_n]\!]\rho x = \mathcal{AE}[\![OK_n \ p_1 \cdots p_n]\!]\rho(\text{apply } \rho(a) \ x)$$
$$\mathcal{AE}[\![p_1@p_2]\!]\rho x = \mathcal{AE}[\![p_1]\!]\rho x + \mathcal{AE}[\![p_2]\!]\rho x$$

Figure 4. Definitions of $\mathcal{AP}$ and $\mathcal{AE}$

to a sequence of simultaneous declarations (their definitions are given in Figure 6). $\mathcal{AP}$ checks whether a value matches an active pattern, while $\mathcal{AE}$ computes the environment produced as a consequence of the matching (these functions are defined in Figure 4). Note that these two functions need an environment where the ADs appearing in the pattern are defined. These functions use some auxiliary functions which are defined in Figure 3.

5 Equational Reasoning

As it has been previously commented, early proposals trying to integrate pattern matching and abstract data types lacked good equational reasoning properties. The reason for that was that they tried, in some sense, to do equational reasoning at the ADT implementation level, while users of an ADT should only assume its specification. The improved proposals [6] do not suffer from this problem. Nevertheless, no general way to reason about the external behaviour of an ADT is given in those proposals.

Algebraic specifications of abstract data types have been around for many years, and there exists a consensus that they are an appropriate mean to inform a potential user about the external behavior of a data type without revealing its implementation details. For instance, if we wish to specify the behaviour of a FIFO queue, then the following equations will do the job:

> **abstype** Queue α
> $dequeue \ (enqueue \ empty \ x) \ == \ empty$
> $isEmpty \ q \ == \ False \Rightarrow$
> $\quad dequeue \ (enqueue \ q \ x) \ == \ enqueue \ (dequeue \ q) \ x$
> $first \ (enqueue \ empty \ x) \ == \ x$
> $isEmpty \ q \ == \ False \Rightarrow$
> $\quad first \ (enqueue \ q \ x) \ == \ first \ q$
> $isEmpty \ empty \ == \ True$
> $isEmpty \ (enqueue \ q \ x) \ == \ False$

Just to facilitate further reasoning, we have adopted a syntax as close as possible to that of functional languages. However, the symbol "$==$" has here a slightly different mean-

ing: $t_1 == t_2$ establishes a congruence between the pairs of terms which can be obtained by instantiating t_1 and t_2 in all possible ways. In particular, we have adopted a variant of algebraic specifications in which operations can be partial, equations can be conditional, and the symbol "$==$" is interpreted as *existential equality*. An equation $s == s' \Rightarrow t == t'$ specifies that if (an instance of) s is well defined and (the corresponding instance of) s' is congruent to s, then (the corresponding instances of) t and t' are well defined and they are congruent. A consequence of the theory underlying this specification style is that terms not explicitly mentioned in the conclusion of an equation are by default undefined. For instance, in the example above, (*first empty*) and (*dequeue empty*) are undefined. See [3, 15] for details.

The algebraic specification of a data type has two main uses: (1) it allows to reason about the correctness of functions external to the data type; (2) it serves as a requirement that any valid implementation of the data type must satisfy. The first use is rather familiar to functional programmers, as it simply consists of equational reasoning. An equation of the data type may be used as a rewriting rule, in either direction, as long as terms are well defined and the premises of the equation are satisfied. We remark that the equations of an algebraic specification are axioms needed in order to prove any interesting theorem about the ADT. In [6] these properties are discovered in an ad-hoc way for a particular implementation of the set ADT.

In the rest of this section we adapt the algebraic specification notation to cope with ADs. When executing active patterns of the form $A \ x_1 \cdots x_n$ appearing in the definition of functions external to the ADT, first, the value v we are trying to match is checked to determine whether the pattern is applicable or not; if this is the case, a computation takes place over v, and variables $x_1, \ldots, x_n$ are bound to some values. At the abstract level, this can be expressed by using a notation similar to that used in code lines 83–85:

$$A \ (o_A^1 \ v) \ \ldots \ (o_A^n \ v) \ \textbf{match} \ v \ \Leftrightarrow \ G_A \ v$$

where G_A is the abstract guard associated to the active destructor A, and the o_A^i are the observer functions applied to the matched value v when the guard evaluates to *True*. Both G_A and o_A^i are ADT operations that must be algebraically specified in the usual way. In the Queue ADT above we

$$\mathcal{V}[\![v]\!]\rho = \rho(v)$$
$$\mathcal{V}[\![a]\!]\rho = \rho(a)$$
$$\mathcal{V}[\![c]\!]\rho = c \text{ in } \mathit{Cnst}_0 \text{ in } \mathit{Val}, \quad \text{if } (c \in C_0)$$
$$\mathcal{V}[\![c]\!]\rho = f \text{ in } \mathit{Func} \text{ in } \mathit{Val}, \quad \text{if } (c \in C_n, n \geq 1)$$
$$\text{where } f(x_1, \ldots, x_n) = (c, x_1, \ldots, x_n) \in \mathit{Cnst}_n \in \mathit{Val}$$
$$\mathcal{V}[\![e_1 \; e_2]\!]\rho = \mathrm{apply} \; (\mathcal{V}[\![e_1]\!]\rho)(\mathcal{V}[\![e_2]\!]\rho)$$
$$\mathcal{V}[\![\backslash v \to e]\!]\rho = \lambda x.\mathcal{V}[\![e]\!](\rho \pm [x/v])$$
$$\mathcal{V}[\![e \; \textbf{where} \; \vec{D}]\!]\rho = \mathcal{V}[\![e]\!](\rho \pm \vec{\mathcal{D}}[\![\vec{D}]\!]\rho)$$

$$\mathcal{V} \left[\!\!\left[\begin{array}{l} \textbf{case } e \textbf{ of} \\ \quad p_1 \mid e_1' \to e_1 \\ \quad \cdots \\ \quad p_n \mid e_n' \to e_n \end{array}\right]\!\!\right] \rho = x \qquad \text{where} \begin{cases} x' = \mathcal{V}[\![e]\!]\rho \\ x = \begin{cases} \mathcal{V}[\![e_1]\!]\rho_1, & \mathcal{AP}[\![p_1]\!]\rho x' \wedge \mathcal{V}[\![e_1']\!]\rho_1 \\ \cdots, \\ \mathcal{V}[\![e_n]\!]\rho_n, & \mathcal{AP}[\![p_n]\!]\rho x' \wedge \mathcal{V}[\![e_n']\!]\rho_n \end{cases} \\ \rho_i = \rho \pm \mathcal{AE}[\![p_i]\!]\rho x' \end{cases}$$

Figure 5. Definition of $\mathcal{V}$

$$\mathcal{D}[\![v = e]\!]\rho = [\mathcal{V}[\![e]\!]\rho/v]$$

$$\mathcal{D}\left[\!\!\left[\begin{array}{l} a \; e_1^1 \; \cdots \; e_1^n \textbf{ match } p_1, \textbf{ if } e_1' \\ \qquad\qquad \cdots \\ a \; e_m^1 \; \cdots \; e_m^n \textbf{ match } p_m, \textbf{ if } e_m' \end{array}\right]\!\!\right]\rho = [f/a], \qquad \text{where } f = \mathcal{V}\left[\!\!\left[\begin{array}{l} \backslash v.\textbf{case } v \textbf{ of} \\ \quad p_1 \mid e_1' \;\; \to \;\; (OK_n \; e_1^1 \; \cdots \; e_1^n) \\ \qquad\qquad \vdots \\ \quad p_m \mid e_m' \to \; (OK_n \; e_m^1 \; \cdots \; e_m^n) \\ \quad \_\qquad\;\; \to \qquad\quad Fail \end{array}\right]\!\!\right]\rho$$

$$\vec{\mathcal{D}}[\![(d_1, \ldots, d_n)]\!]\rho = \rho \pm \rho' \text{ where } \begin{cases} \rho' = \rho_1 + \cdots + \rho_n \\ \rho_i = \mathcal{D}[\![d_i]\!](\rho \pm \rho') \end{cases}$$

Figure 6. Definitions of $\mathcal{D}$ and $\vec{\mathcal{D}}$

would write:

$$Empty \textbf{ match } q \Leftrightarrow isEmpty \; q$$
$$First \; (first \; q) \textbf{ match } q \Leftrightarrow \neg(isEmpty \; q)$$
$$Dequeue \; (dequeue \; q) \textbf{ match } q \Leftrightarrow \neg(isEmpty \; q)$$

where the *isEmpty*, *first* and *dequeue* operations have been specified above.

Let us assume that we have defined the *inv* operation reversing a queue, by using the active patterns *Empty, First* and *Dequeue* in the following way:

```
139  inv :: Queue α → Queue α
140  inv Empty = empty
141  inv (First x)@(Dequeue q) = enqueue (inv q) x
```

Now, we wish to prove the following inductive theorem:

$$first \; (inv \; (enqueue \; q \; x)) == x$$

The technique consists of performing matching at the abstract level, expanding the definition of *inv*, and then applying equational reasoning —by using the specification equations and perhaps some auxiliary theorems— in order to simplify the resulting expressions. First, we prove the base case, where we assume $q = empty$:

$$first \; (inv \; (enqueue \; empty \; x))$$
$$== \quad — \text{ by unfolding } inv \text{ with}$$
$$\qquad isEmpty \; (enqueue \; empty \; x) == False$$
$$first \; (enqueue \; (inv \; (dequeue \; q')) \; (first \; q'))$$
$$\qquad \textbf{where } q' == enqueue \; empty \; x$$
$$== \quad — \text{ by applying the specification equations}$$
$$first \; (enqueue \; (inv \; empty) \; x)$$
$$== \quad — \text{ by unfolding } inv \text{ with}$$
$$\qquad isEmpty \; empty \; == True$$
$$first \; (enqueue \; empty \; x)$$
$$== \quad — \text{ by applying the specification equations}$$
$$x$$

Now we proceed with the induction step, where we assume $q \neq empty$:

$$first \; (inv \; (enqueue \; q \; x))$$
$$== \quad — \text{ by unfolding } inv, \text{ and by using}$$
$$\qquad isEmpty \; (enqueue \; q \; x) == False$$
$$first \; (enqueue \; (inv \; q') \; x')$$

being $q' = dequeue \; (enqueue \; q \; x)$ and $x' = first \; (enqueue \; q \; x)$. Now, $q \neq empty$ implies that $q' == dequeue \; (enqueue \; q \; x)$ is not empty (this is an easy equational theorem). Then, by applying the following (also easy to prove) inductive theorem,

$$(isEmpty \; q') == False \Rightarrow (isEmpty \; (inv \; q')) == False$$

we have

$$
\begin{array}{ll}
& \textit{first } (\textit{enqueue } (\textit{inv } q') \; x') \\
== & \text{— by equational reasoning} \\
& \textit{first } (\textit{inv } q') \\
== & \text{— by unfolding } q' \\
& \textit{first } (\textit{inv } (\textit{dequeue } (\textit{enqueue } q \; x))) \\
== & \text{— by equational reasoning,} \\
& \qquad \text{knowing that } q \neq \textit{empty} \\
& \textit{first } (\textit{inv } (\textit{enqueue } (\textit{dequeue } q) \; x)) \\
== & \text{— by induction hypothesis knowing that} \\
& \qquad (\textit{dequeue } q) \text{ is smaller than } q \\
& x
\end{array}
$$

and the theorem holds.

6 Compilation

An important feature of pattern matching is the quality of the generated code. There have been several algorithms for pattern matching compilation, featuring different characteristics: with backtracking [4, 22], without backtracking [7], and lazy ones, either with or without backtracking [11, 16, 12]. So, it is important to investigate whether it is possible the efficient compilation of pattern matching when active destructors are used. In [14] there is a detailed study of ADs compilation by using two different techniques: in-line expansion of the condition pattern and by means of a projection function. Algorithms with and without backtracking were presented for both techniques. The first one produces a code similar to the one that would be generated by a programmer if he/she had access to the implementation. Nevertheless, this technique cannot be applied if ADs are used as parameters of a function (Haskell classes, ML structures or higher order functions). For this reason, we introduced a compilation technique using projection functions.

Due to the complexities involved in these techniques, we are going to describe only the in-line compilation technique by means of an example. The basic technique using projection functions is also described in [1] for AVCs together with some optimizations. In [14] new optimizations are described in order to avoid the repeated evaluation of the same projection function. These optimizations are very important in our proposal: while the projection function of an AVC only performs trivial computations, the projection function of an AD can perform time consuming ones.

6.1 In-Line Expansion

As usual, patterns are compiled by transforming them into a group of nested **case** expressions over simple patterns, i.e. constructors applied to variables. This process is performed by a function called $\mathcal{C}$ receiving as arguments the three important components of a function defined by pattern matching: a vector of variables, a matrix of patterns m and a vector of expressions e. For example, the code generated for the *loop* function (defined in code lines 95 through 99) is:

$$loop = \backslash \alpha_1 \rightarrow$$
$$\mathcal{C}\big((\alpha_1), \begin{pmatrix} (\textit{First EmptyTree})@(\textit{Dequeue } q) \\ (\textit{First } (\textit{Node } t_1 \; a \; t_2))@(\textit{Dequeue } q) \\ \textit{Empty} \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix}\big)$$

where e_1, e_2, e_3 are the right hand expressions of the rules defining *loop*. The $\mathcal{C}$ function is recursively defined by cases according to the shape of the patterns matrix. In the following, we concentrate on the compilation of patterns defined using ADs.

Before compiling a pattern, the rules defining ADs must be split into two parts: the condition pattern, and the rest of the code computing the output arguments (the partial projection functions). In the queue example we obtain the following condition patterns:

$$
\begin{array}{ll}
Empty^1 & = \; Queue \; [\,] \; [\,] \\
First^1 & = \; Queue \; (\_ : \_) \; \_ \\
Dequeue^1 & = \; Queue \; (\_ : \_ : \_) \; \_ \\
Dequeue^2 & = \; Queue \; [\_] \; \_
\end{array}
$$

and the following partial projection functions:

$$
\begin{array}{l}
empty^1 \; (Queue \; [\,] \; [\,]) = () \\
first^1 \; (Queue \; (x : \_) \; \_) = x \\
dequeue^1 \; (Queue \; (\_ : (fs@(\_ : \_))) \; ls) = (Queue \; fs \; ls) \\
dequeue^2 \; (Queue \; [\_] \; ls) = (Queue \; (reverse \; ls) \; [\,])
\end{array}
$$

Let us note that (1) all variables appearing in the condition patterns are anonymous and that (2) every partial projection function will be called only when the corresponding argument has matched the pattern, so they can be compiled without needing to perform the matching or to fail.

Once ADs have been translated, we can perform one more transformation step of the $\mathcal{C}$ function. We decompose the matrix m into two parts: the sequence of ADs appearing in each pattern m_A and the sequence of patterns appearing in their output arguments m_P:

$$
m_A = \begin{pmatrix} \langle First, Dequeue \rangle \\ \langle First, Dequeue \rangle \\ \langle Empty \rangle \end{pmatrix}
$$

$$
m_P = \begin{pmatrix} \langle (EmptyTree), (q) \rangle \\ \langle (Node \; t_1 \; a \; t_2), (q) \rangle \\ \langle () \rangle \end{pmatrix}
$$

The next step consists of replacing in m_A each sequence of ADs by the conjunction of their condition patterns. While *Empty* and *First* have only a defining rule, *Dequeue* has two. This fact gives rise to the duplication of the first two rows of m_A, m_P and e. Each pattern of m_P is associated to the corresponding partial projection function. After these manipulations, and after the simplification of the m_A patterns conjunctions, the matrices are

$$
m_A = \begin{pmatrix} Queue \; (\_ : (\_ : \_)) \; \_ \\ Queue \; [\_] \; \_ \\ Queue \; (\_ : (\_ : \_)) \; \_ \\ Queue \; [\_] \; \_ \\ Queue \; [\,] \; [\,] \end{pmatrix}
$$

$$
m_P = \begin{pmatrix} \langle \big(first^1, (EmptyTree)\big), \big(dequeue^1, (q)\big) \rangle \\ \langle \big(first^1, (EmptyTree)\big), \big(dequeue^2, (q)\big) \rangle \\ \langle \big(first^1, (Node \; t_1 \; a \; t_2)\big), \big(dequeue^1, (q)\big) \rangle \\ \langle \big(first^1, (Node \; t_1 \; a \; t_2)\big), \big(dequeue^2, (q)\big) \rangle \\ \langle \rangle \end{pmatrix}
$$

The computation carries on by using a new function $\mathcal{C}'$ having as arguments $(\alpha_1), m_A, m_P, (), (), e$. The $\mathcal{C}'$ function is expanded taking into account the shape of m_A (in our example, only constructors appear). Because the same pattern

$$e_1' = \mathcal{C}'\big((),(),(\langle\rangle),(\alpha_1),(),(e_3)\big)$$

$$e_2' = \mathcal{C}'\Big((),(),\left(\begin{array}{c} \big\langle\big(\mathit{first}^1,(\mathit{EmptyTree})\big),\big(\mathit{dequeue}^2,(q)\big)\big\rangle \\ \big\langle\big(\mathit{first}^1,(\mathit{Node}\ t_1\ a\ t_2)\big),\big(\mathit{dequeue}^2,(q)\big)\big\rangle \end{array}\right),(\alpha_1),(),\begin{pmatrix} e_1 \\ e_2 \end{pmatrix}\Big)$$

$$e_3' = \mathcal{C}'\Big((),(),\left(\begin{array}{c} \big\langle\big(\mathit{first}^1,(\mathit{EmptyTree})\big),\big(\mathit{dequeue}^1,(q)\big)\big\rangle \\ \big\langle\big(\mathit{first}^1,(\mathit{Node}\ t_1\ a\ t_2)\big),\big(\mathit{dequeue}^1,(q)\big)\big\rangle \end{array}\right),(\alpha_1),(),\begin{pmatrix} e_1 \\ e_2 \end{pmatrix}\Big)$$

Figure 7. Leaves of $\mathcal{C}'$

appears several times, the resulting expansion is a very small **case**'s *tree*:

```
142  case α₁ of
143     (Queue α₂ α₃) →
144         case α₂ of
145             [ ] →  case α₃ of
146                     [ ] → e₁′
147                     _  → MATCHING ERROR
148             (α₄:α₅) → case α₅ of
149                     [ ] → e₂′
150                     (_:_) → e₃′
```

where the α_i are fresh variables. This tree has four *leaves*. One of them is a failing branch, but it never will be taken because of the queue invariant. The other three leaves have calls to the $\mathcal{C}'$ function as it is shown in Figure 7. The e_1' expansion is trivial, and it produces e_3. But in order to expand e_2' or e_3', it is necessary to perform matching over the ouput values of the ADs, that is, over the result of the evaluation of the corresponding partial projection function. After performing these evaluations, the recursive expansion goes on by using the $\mathcal{C}$ function. Now we show the result of the e_2' expansion (the e_3' expansion is very similar):

```
151  let α₆ = first¹ α₁
152      α₇ = dequeue² α₁
```
$$153\ \mathbf{in}\ \mathcal{C}((\alpha_6,\alpha_7),\begin{pmatrix} \mathit{EmptyTree} & q \\ (\mathit{Node}\ t_1\ a\ t_2) & q \end{pmatrix},\begin{pmatrix} e_1 \\ e_2 \end{pmatrix})$$

The rest of the expansion only concerns constructors, and it is solved by means of usual techniques. In Figure 8, we present the resulting final code.

7 Conclusions

We have presented *active patterns*, a construction smoothly integrating pattern matching with abstract data types. We have illustrated its use and its semantics by means of some simple examples, and we have also given a sketch of a possible syntax and we have defined a denotational semantics for this syntax. We have shown how to reason about the programs containing this facility, and we have given some guidelines about their efficient compilation.

The previous proposals reviewed in this paper have also worked on these points, but comparatively, we think that active destructors have more expressive power and less problems than these proposals. For example, abstract value constructors [1] are the particular case of ADs in which patterns simply select some representation components and do not perform any computation. With respect to laws [19], the main difference is that ADs allow the programmer to choose any representation for the ADT. This choice is severely limited for laws. Also, ADs can do computations during the matching process. With respect to the latest version of views [6], ADs do not restrict the type of patterns that can be used while defining a function by pattern matching. Our patterns are independent of each other and they are not *bundled* in any way. In general, it is easy to see that all the deterministic patterns which can be expressed by using complex pattern languages as those defined in [8] or in [17], can be defined in terms of ADs.

We have also shown how to integrate the algebraic specification techniques developed for abstract data types during the 70's and the 80's into the equational reasoning process about functional programs using active patterns. In a previous paper [13] we showed how to formally verify that a functional implementation of an ADT satisfies an algebraic specification. We believe that the attempts to integrate the results of the functional and the algebraic specifications communities can be very profitable for both.

Finally, we have introduced two new features: the capability of defining non algebraic patterns as the $(n+k)$ patterns and a type system for ADs. This type system allows to treat patterns as *first class citizens*. As it is claimed in [18] for a concurrent functional language, a special type is the first step in order to get abstraction in a certain part of a language.

Acknowledgments

References

[1] William E. Aitken and John H. Reppy. Abstract data constructors. In *Proceedings ACM SIGPLAN Workshop on ML and its Aplications*, pages 1–11, 1992.

[2] A. W. Appel and D. B. MacQueen. A standard ML compiler. In *Functional Programming Languages and Computer Architecture'87, LNCS 274*, pages 301–324, 1987.

[3] Egidio Astesiano and Maura Cerioli. On the existence of initial models for partial (higher order) conditional specifications. In *TAPSOFT'89, LNCS 351*, pages 74–88, 1989.

[4] Lennard Augutsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture'85, LNCS 201*, pages 368–381, 1985.

[5] Emery Berger. FP + OOP = Haskell, 1991.

[6] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171 – 190, 1993.

$$
\begin{aligned}
loop &= \backslash\alpha_1 \rightarrow \\
&\quad \textbf{case } \alpha_1 \textbf{ of} \\
&\qquad Queue\ \alpha_2\ \alpha_3 \rightarrow \\
&\qquad\quad \textbf{case } \alpha_2 \textbf{ of} \\
&\qquad\qquad [\,] \rightarrow \textbf{case } \alpha_3 \textbf{ of} \\
&\qquad\qquad\qquad [\,] \rightarrow [\,] \\
&\qquad\qquad\qquad (\_:\_) \rightarrow \textsc{matching error} \\
&\qquad\qquad (\alpha_4{:}\alpha_5) \rightarrow \textbf{case } \alpha_5 \textbf{ of} \\
&\qquad\qquad\qquad [\,] \rightarrow \textbf{let } \alpha_6 = \mathit{first}^1\ \alpha_1 \\
&\qquad\qquad\qquad\qquad\qquad \alpha_7 = \mathit{dequeue}^2\ \alpha_1 \\
&\qquad\qquad\qquad\qquad \textbf{in case } \alpha_6 \textbf{ of} \\
&\qquad\qquad\qquad\qquad\quad EmptyTree \rightarrow loop\ \alpha_7 \\
&\qquad\qquad\qquad\qquad\quad Node\ \alpha_8\ \alpha_9\ \alpha_{10} \rightarrow \alpha_9 : loop\ (enqueue\ (enqueue\ \alpha_7\ \alpha_8)\ \alpha_{10}) \\
&\qquad\qquad\qquad (\alpha_6{:}\alpha_7) \rightarrow \textbf{let } \alpha_8 = \mathit{first}^1\ \alpha_1 \\
&\qquad\qquad\qquad\qquad\qquad \alpha_9 = \mathit{dequeue}^1\ \alpha_1 \\
&\qquad\qquad\qquad\qquad \textbf{in case } \alpha_8 \textbf{ of} \\
&\qquad\qquad\qquad\qquad\quad EmptyTree \rightarrow loop\ \alpha_9 \\
&\qquad\qquad\qquad\qquad\quad Node\ \alpha_{10}\ \alpha_{11}\ \alpha_{12} \rightarrow \alpha_{11} : loop\ (enqueue\ (enqueue\ \alpha_9\ \alpha_{10})\ \alpha_{12})
\end{aligned}
$$

Figure 8. Compiled code of *breadthSearch*

[7] Albert Gräf. Left-to-right tree pattern matching. In *4th Rewriting Techniques and Applications, LNCS 488*, pages 323–334, 1991.

[8] Reinhold Heckmann. A functional language for the specification of complex tree transformation. In *ESOP'88, LNCS 300*, pages 175 – 190, 1988.

[9] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairburn, J. Fasel, K. Hammond, J. Hughes, T. Johnson, D. Kieburtz, R. Nikhil, S. Peyton Jones, M. Reeve, D. Wise, and J. Young. Report on the programming language Haskell: A non-strict purely functional language (version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Deparment of Computer Science, 1990.

[10] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the functional programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5):1–154, 1992.

[11] Alain Laville. Implementation of lazy pattern matching algorithms. In *ESOP'88, LNCS 300*, pages 298–316, 1988.

[12] Luc Maranget. Two techniques for compiling lazy pattern matching. Technical Report RR-2385, INRIA, October 1994.

[13] Manuel Núñez, Pedro Palao Gostanza, and Ricardo Peña. A second year course on data structures based on functional programming. In *Functional Programming Languages in Education, LNCS 1022*, pages 65–84, 1995.

[14] Pedro Palao Gostanza. Integración de ajuste de patrones y tipos abstractos de datos. Technical Report DIA-UCM 16/94, Dept. Informática y Automática, Universidad Complutense de Madrid, 1994. In Spanish.

[15] Ricardo Peña. *Diseño de Programas*. Prentice Hall, 1993. In Spanish.

[16] Laurence Puel and Ascánder Suárez. Compiling pattern matching by term decomposition. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 273–281, 1990.

[17] Christian Queinnec. Compilation of non-linear, second order patterns on S-expressions. In *PLILP'90, LNCS 456*, pages 340–357, 1990.

[18] John Reppy. A higher-order concurrent language. In *Proceedings SIGPLAN'91*, pages 294–305, 1991.

[19] Simon Thompson. Laws in Miranda. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 1–12, 1986.

[20] Simon Thompson. Lawful functions and program verification in Miranda. *Science of Computer Programming*, 13(2–3):181–218, 1990.

[21] David A. Turner. An overwiew of Miranda. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[22] Philip Wadler. Efficient compilation of pattern matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice Hall International, 1987.

[23] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings 14th Principles of Programming Languages*, pages 307 – 313, 1987.