Mutable Abstract Datatypes
— or —
How to Have Your State and Munge It Too

Paul Hudak

Yale Research Report YALEU/DCS/RR-914
Yale University
Department of Computer Science
New Haven, CT 06520
hudak@cs.yale.edu

December 1992
(revised May 1993)

Abstract

A *mutable abstract datatype*, or MADT, is any ADT whose rewrite semantics permits "destructive re-use" of one or more of its arguments while still retaining confluence in a general rewrite system. MADT's have the obvious advantage of guaranteed, predictable performance for every operation, yet algebraic reasoning ("referential transparency") is preserved. A spectrum of MADT designs is presented, including *direct*, *continuation* (CPS), and *monadic* designs. It is shown that given any simple ADT, along with an axiomatization possessing a certain linearity property, an equivalent MADT in direct, CPS, or monadic style may be derived automatically. A simple graph rewrite semantics is given which implies an efficient implementation. Constant-time and constant-space lookup and update of arrays, for example, is an immediate consequence of this result.

1 Introduction

It's been said many times before: "Functional languages are *great*, but they can't deal with *state*!" to which functional programmers often reply: "But a *compiler* that's great, will *eliminate* state!"

Although recent advances in compiler optimization techniques have eliminated many concerns over efficiency, optimizations have their own set of problems: (1) they are often expensive (in terms of compilation resources), (2) they aren't always good enough, (3) they are often hard to reason about, and (4) they are implementation dependent (and thus programs that depend on them are not portable). Perhaps more importantly, compiler optimizations aren't *explicit*, and in this sense are not "expressive" enough.

In broad terms, to deal with state effectively it seems that one must have operations that *update* and *query* the state with acceptable efficiency, along with a way to *sequence* those operations to achieve deterministic behavior. Imperative languages can be (perversely) viewed as (rather heavy) syntactic sugar for doing just these sorts of things. Is there any hope to do the same in a "pure" functional language such as Haskell, Hope, Miranda,[1] or even a pure subset of ML or Scheme?

1.1 Previous Work

One approach to solving this problem is to build a fancy type system that rejects programs for which safe and efficient state manipulations cannot be guaranteed. In other words, replace the BNF syntactic constraints in an imperative language with the static semantics imposed by a type system. Examples include the "linear" type systems of Lafont/Wadler/Holmstrom [8, 4, 17], the "single-threaded" type system of Guzman/Hudak [2], and the "stratified" type system of Swarup/Reddy/Ireland [14]. Although this line of research is promising, the complexity of the resulting type system can be rather daunting, and some negative results in both programming and performance have been reported [19].

In a different setting, advances in *I/O systems* for functional languages have yielded designs in which arbitrary operations on objects located *outside* of a functional program could be expressed in a purely functional way [6]. Haskell's I/O design [5], for example, integrates both a stream-based model of I/O [3, 15] and a continuation-based model [7]. Using these approaches, all of the power and efficiency of conventional operating systems can be placed in the hands of a functional programmer.

More recently, through his work on using *monads* to structure functional programs, Wadler discovered an abstract datatype for an *array* that can be implemented safely using in-place, destructive updates, without resorting to any special kind of type system [16, 18]. Whearas the I/O solutions mentioned above dealt with the state residing outside of a program, this was the first example of an efficient treatment of state *within* a functional program. Related efforts can be found in [9, 14].

1.2 Overview

Many questions were left unanswered, however, in the various approaches to I/O and monadic solutions to state mentioned above. When do these ideas work, and when do they not? How can we be sure that referential transparency is not destroyed? Is the monadic solution to "internal" state the only one possible? Can other objects of state besides arrays be captured in this way? And what is the proper operational semantics that captures the behavior that we desire?

We informally define a *mutable abstract datatype*, or MADT, as any ADT whose rewrite semantics permits "destructive re-use" of one or more of its arguments while still retaining confluence in a general rewrite system. MADT's do not depend on a type system (beyond say, a conventional Hindley-Milner type system) or program analysis technique to determine when updates can be done destructively. They thus have the obvious advantage of guaranteed, predictable performance for

[1] Miranda is a trademark of Research Software Ltd.

every operation, yet algebraic reasoning ("referential transparency") is preserved. This greatly aids reasoning about program efficiency, even when using a lazy language, when such reasoning can be especially difficult.

In this paper we summarize the following results about MADT's:

1. There are *many* MADT designs. We highlight two designs of particular interest: a *direct* MADT and a *CPS* MADT (which are related to the notions of direct and continuation semantics, respectively). We also present a *monadic* MADT that falls somewhere between these two.

2. Given any simple ADT, along with an axiomatization possessing a certain linearity property, an equivalent MADT in direct, CPS, or monadic style may be derived *automatically*. This derivation is achieved by a translation of the axiomatization of the original ADT operators into a new one involving the operators of the target MADT.

   (The translation of the CPS MADT is of particular interest, since it amounts to a *CPS conversion of equations*, and reveals the duality between constructors and selectors; namely, they "switch roles" as a result of the translation.)

3. A simple graph rewrite semantics can be given for any of these MADT's which guarantees efficient implementation in the sense described earlier. Constant-time and constant-space lookup and update of arrays, for example, is an immediate consequence of this result.

(We will use Haskell [5] syntax for our examples throughout; occasional comments on syntax are included to aid the reader. Although only one version of MADT (the CPS version for an array) has been fully implemented, most of the code in this document has been executed under the Yale Haskell implementation to ensure at least functional correctness.)

2 A Spectrum of MADT Designs

Definition: Given a type of interest `T`, a *simple ADT* is one in which each operation can be classified as either a *generator* of type `X1->T`, a *mutator* of type `X2->T->T`, or a *selector* of type `X3->T->X4`, where the `X`$i \neq$`T` are auxiliary datatypes.

This is a fairly standard classification for ADT's, and most conventional ADT's such as arrays, lists, stacks, queues, and graphs can be expressed as simple ADT's. For sake of argument, suppose a particular simple ADT is defined by the following three operators:

```
g :: X1 -> T                        -- a generator
m :: X2 -> T -> T                   -- a mutator
s :: X3 -> T -> X4                  -- a selector
```

For example, for a fixed-size integer array, these might correspond to:

```
newArr :: (Int,Int,...,Int) -> Array   -- the tuple contains initial values
update :: (Ix,Int) -> Array -> Array   -- Ix is the array index type
select :: Ix -> Array -> Int
```

This is a simplified version of typical array operations that might be found in a purely functional language. Although we will ignore for now the axiomatization of this ADT, note that an efficient (meaning constant-time, constant-space selection and update) implementation is likely to be difficult, since the intermediate results of a series of updates have indefinite lifetimes, preventing the immediate "re-use" of old arrays. This is the classic "array update" problem for which many solutions have been proposed, most along the lines of either static or dynamic optimization techniques.

We will pursue a different approach. Rather than design an optimization strategy, we wish to *redesign the ADT* in such a way that *every* update is guaranteed to execute in constant time and space. The trick to doing this is to somehow limit access to the array (or, in general, the type-of-interest, or *state*) to prevent more than one copy of it from being accessed at the same time. In doing so we will uncover three solutions: a *direct MADT*, a *CPS MADT*, and a *monadic MADT*. In each case we will suffix the above operators with `D`, `C`, and `M`, respectively, to distinguish them.

2.1 A Direct MADT

We can limit access to values in a functional language through the use of an abstract datatype. But simply hiding the state does not get us very far (the reader may wish to try this on his/her own), because the operations needed to compute anything will result in re-exposure of the state.

What we need instead is a way to hide *functions* that *operate* on the state. Starting with the general ADT described above, perhaps the most *direct* way to do this is as follows:

```
gD :: X1 -> D0 T              -- e.g.: newArrD :: (Int,...,Int) -> D0 Array
mD :: X2 -> D1(T -> T)        --       updateD :: (Ix,Int) -> D1(Array->Array)
sD :: X3 -> D2(T -> X4)       --       selectD :: Ix -> D2(Array->Int)
```

The only difference between these types and the original ones is that three type constructors — `D0`, `D1`, and `D2` — have been added. These new types are abstract, and serve the purpose of limiting access to the state to a special set of combinators which allow composing the state operations in only a "single-threaded" manner. Two such combinators are needed, `($)` and `get`, along with a function `begin` to initialize a computation, and a function `val` to return a value from a computation. Their functionality is given by:[2]

```
($) :: D1(a -> b) -> D1(b -> c) -> D1(a -> c)         -- $ is an infix operator
get :: D2(a -> b) -> (b -> D1(a -> c)) -> D1(a -> c)
begin :: D0 a -> D1(a -> b) -> b
val :: b -> D1(a -> b)
```

Here are the hidden definitions of these combinators (written as a Haskell module to emphasize what is hidden and what is exported):

[2] `$` is written here with a tad more polymorphism than is actually needed, but it's easier to do that than to explain why a less polymorphic version is good enough!

```
module DirectMADT (D0, D1, D2, ($), get, begin, val)  -- export list
where

data D0 a = MkD0 a                                    -- new datatypes,
data D1 a = MkD1 a                                    --    each having a
data D2 a = MkD2 a                                    --    single constructor

(MkD1 f) $ (MkD1 g) = MkD1 (g.f)                      -- . is function composition
get (MkD2 f) g = MkD1 (\a -> let MkD1 h = g (f a) in h a)
begin (MkD0 a) (MkD1 f) = f a
val b = MkD1 (\a -> b)                                -- /a->... is lambda abstraction
```

The tagging and untagging of functions is a bit confusing, but note that `($)` is essentially reverse composition, `begin` is reverse application, and `get`, when stripped of the tags, could be defined as:

```
get f g a = g (f a) a
```

Note that the constructors `MkD0`, `MkD1`, `MkD2` are *not exported* — `D0`, `D1`, and `D2` are *abstract* — and thus one cannot operate on the type-of-interest directly.

As an example, here is a simple program operating on arrays:

```
begin (newArrD (0,...,0))              -- allocate new array
 (updateD (1,1)   $                    -- set 1st element to 1
  get (selectD 1) $ \x->               -- read first element and bind it to x
  updateD (1,x+1) $                    -- set 1st element to x+1
  get (selectD 1) $ \y->               -- read first element and bind it to y
  val y                  )             -- return y
```

Note the "imperative feel" of this program, even though it is purely functional. Later we will show that every `select` and `update` operation in a program using this MADT can be implemented in constant time and space. This is not true of the original ADT.

2.2 A CPS MADT

In the above program, note that `begin/newArrD`, `($)/updateD`, and `get/selectD` always appear in pairs. This raises the possibility of merging `begin`'s functionality into `newArrD`, `($)` into `updateD`, and `get` into `selectD`. Doing so amounts to pushing the sequencing behavior of the combinators into the operators themselves (as continuation arguments), and yields the following design:

```
gC   :: X1 -> (T -> a) -> a
mC   :: X2 -> (T -> a) -> (T -> a)
sC   :: X3 -> (X4 -> (T -> a)) -> (T -> a)
valC :: a -> (T -> a)
```

Note the recurring type `(T -> a)`. This is the only type involving the type-of-interest `T`, and is thus the only one that needs to be hidden. Using a type synonym to simplify things we thus arrive at:

```
type W a = C(T -> a)              -- type synonym

gC :: X1 -> W a -> a              -- e.g.: newArrC :: (Int,...,Int) -> W a -> a
mC :: X2 -> W a -> W a            --       updateC :: (Ix,Int) -> W a -> W a
sC :: X3 -> (X4 -> W a) -> W a    --       selectC :: Ix -> (Int -> W a) -> W a
```

Intuitively, these operations behave as follows (using the array version on the right as an example): (1) `newArrC` allocates a new array and passes it to its continuation argument which represents the "rest of the program," (2) `updateC` modifies the current array and passes it on to its continuation argument, and (3) `selectC` reads from the current array, passing the result and the unchanged array to its continuation argument. We will formalize all this shortly.

This is what we call an MADT in full *continuation-passing style* — a *CPS MADT*. Note now that no special composition operator is needed at all; ordinary composition will do. However, for convenience we would still like to write things in a "sequential" style, so we include a definition of `($)` along with `valC ::  a -> W a`, below:

```
module CPSMADT (W, C, ($), valC)  where

data C a = MkC a
type W a = C(T -> a)

f $ a = f a                       -- infix application (note: nothing hidden)
valC a = MkC (\t->a)              -- this is as before
```

This design allows us to write the previous example as:

```
newArrC (0,...,0)
 (updateC (1,1)   $
  selectC 1       $ \x->
  updateC (1,x+1) $
  selectC 1       $ \x->
  valC x                )
```

which is arguably a bit more aesthetic than the direct version.

2.3 A Monadic MADT

As a third possibility, suppose we decide to hide the state in the original ADT in such a way that the hidden type is more or less the same in each operator. One way to do this is as follows:

```
gM :: X1 -> A(T -> (T,a)) -> a
mM :: X2 -> A(T -> (T,()))
sM :: X3 -> A(T -> (T,X4))
```

Here note that `gM` is in CPS style, whereas `mM` and `sM` are in direct style. This "hybrid" design requires the following set of combinators along with their definitions:

```
module MonadicMADT (A, ($), valM)  where

data A a = MkA a
($) :: A(T -> (T,a)) -> (a -> A(T -> (T,b))) -> A(T -> (T,b))
valM :: a -> A(T -> (T,a))
(MkA f) $ g = MkA (\t -> let (t',a) = f t
                             MkA g' = g a
                         in g' t'        )
valM a = MkA (\t -> (t,a))
```

To see that this is a monad, first note that through the use of a type synonym the operator functionalities can be written more succinctly as:

```
type M a = A(T -> (T,a))

gM   :: X1 -> M a -> a            -- e.g.: newArrM :: (Int,...,Int) -> M a -> a
mM   :: X2 -> M ()                --       updateM :: (Ix,Int) -> M ()
sM   :: X3 -> M X4                --       selectM :: Ix -> M Int
($)  :: M a -> (a -> M b) -> M b
valM :: a -> M a
```

Now note that the triple `(M,($),valM)` is precisely Wadler's monad triple `(M,bindM,unitM)`, and the monadic laws hold for the definitions given above. The running example in monadic form is:

```
newArrM (0,...,0)
 (updateM (1,1)   $ \()->
  selectM 1       $ \x ->
  updateM (1,x+1) $ \()->
  selectM 1       $ \x ->
  valM x                 )
```

3 MADT Axiomatizations and Operational Semantics

The previous section showed several MADT designs, including full definitions of the combinators, but did not give precise definitions, or *axiomatizations*, for any particular set of operators. In this section we will do so for the array MADT, giving first an equational axiomatization, but then showing how the equations may be implemented efficiently using a graph rewrite semantics. We

choose a graph rewrite system to express the operational semantics since it is suitably abstract, yet adequately captures notions of sharing, updating, and sequencing — three key ingredients to dealing with state — and is easily related to traditional implementation techniques used for functional languages.

Recall the original array ADT defined earlier:

```
newArr :: (Int,...,Int) -> Array
update :: (Ix,Int) -> Array -> Array
select :: Ix -> Array -> Int
```

One of the most common axiomatizations of this ADT is given by:

```
select i (newArr (x1,...,xi,...,xn)) = xi
select i (update (j,x) a)            = if i==j then x
                                       else select i a
```

where for simplicity we have ignored error conditions such as index out-of-bounds. Unfortunately, this axiomatization has one serious drawback: array lookup requires time proportional to the number of previous updates (which may in fact be much larger than the size of the array itself)! An MADT design using this axiomatization will have the same performance characteristic (we aren't miracle workers).

Thus, we choose instead this alternative axiomatization:

```
select i (newArr (x1,...,xi,...,xn))     = xi
update (i,y) (newArr (x1,...,xi,...,xn)) = newArr (x1,...,y,...,xn)
```

In this context `newArr` can be thought of as a primitive that allocates a contiguous block of storage for the array, and `select` uses a constant-time indexing operation to access its element in one reduction step (which in fact conforms more closely to one's intuitions about arrays as used in practice).

But what about `update`? Clearly it can *find* the element to be updated in constant-time, but if this ADT is to be embedded in a functional language, in the general case it will have to create an entirely new *copy* of the array to ensure confluence. This takes time and space proportional to the size of the array. Fortunately, however, all three of the MADT designs presented earlier admit simple implementations that always permit *resue* of the old array when creating the new one. This means that `update` can be implemented in constant time and space, and is the key point of this paper. The details of this for each MADT are given below.

3.1 Graph Rewrite Semantics for the Direct MADT

Consider first the direct MADT for an array:

```
newArrD :: (Int,...,Int) -> D0 Array
updateD :: (Ix,Int) -> D1(Array->Array)
selectD :: Ix -> D2(Array->Int)
```

Since the arrays themselves are "hidden", the new axiomatization has to be given in conjunction with the combinators:

```
begin (newArrD (...,xi,...)) (get (selectD i) g) = begin (newArrD (...,xi,...)) (g xi)
begin (newArrD (...,xi,...)) (updateD (i,y) $ d) = begin (newArrD (...,y,...)) d
begin (newArrD (...,xi,...)) (val b)             = b
```

Such an axiomatization can be derived automatically as described in Section 4, and its correctness with respect to the original axiomatization is given in Section 4.2.

With this new axiomatization we are now in a position to argue for an efficient implementation. To make this argument most convincing, we wish to expose the hidden representation of the array more directly in the axioms. To do so, we rewrite the above axioms in terms of the hidden representation, and include a step in which the initial hidden array is explicitly created. Using the syntax `<<x1,x2,...,xn>>` as a representation for the hidden array, we add the following rule for `begin`:

```
begin (newArrD (x1,...,xn)) d = begin (MkDO <<x1,...,xn>>) d
```

If this rule is used to expand each occurrence of `begin` in the above axioms, we arrive at a version of the axiomatization that exposes the array representation:

```
begin (MkDO <<...,xi,...>>) (get (selectD i) g) = begin (MkDO <<...,xi,...>>) (g xi)
begin (MkDO <<...,xi,...>>) (updateD (i,y) $ d) = begin (MkDO <<...,y,...>>) d
begin (MkDO <<...,xi,...>>) (val b)             = b
```

We can now think of these four equations as the specification of a graph rewrite system, as shown in Figure 1. Clearly the time and space required to implement the first rule (that involving `newArrD`) is directly proportional to the size of the array. Using this fresh, but otherwise "hidden" array, the operators `selectD` and `val` require only constant time and space. But what about `updateD`? If the old array could be reused, then clearly it also could be implemented in constant time and space. In fact, this can be done, as implied by the following result.

Theorem: The graph rewrite semantics shown in Figure 1, in which the rule for `updateD` reuses the array argument, is confluent.

Proof: Assuming the system is confluent when copying the array, it is sufficient to show that the reduction rule for `update` can only occur in a context where there is exactly one pointer to the old array. We do this by induction: The base case is the rule for `newArrD`, which creates the initial, *single* pointer to the array. The induction step assumes that there is exactly one pointer to the array at the time one of the other rules is to be applied. But each of these other rules adds no additional pointers, so by induction there can never be more than one pointer to the array. □

Generalizing this argument to an arbitrary direct MADT is straightforward, although it relies on a linearity constraint on the axiomatization, to be presented in a later section.

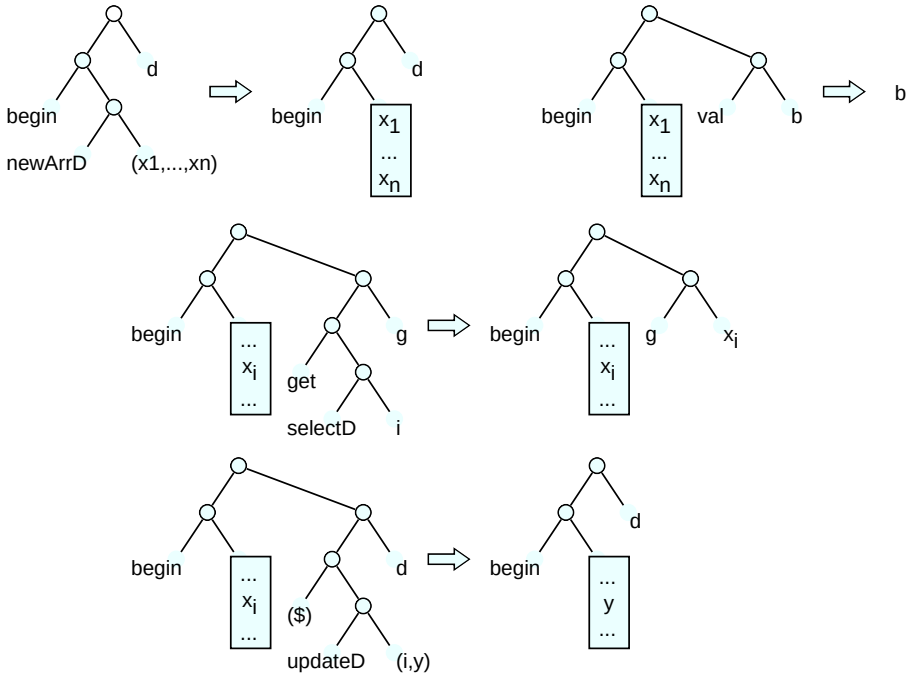

Figure 1: Graph Rewrite Semantics for Direct Array MADT

3.2 Graph Rewrite Semantics for CPS MADT

Consider now the CPS MADT for an array:

```
type W a = C(T -> a)

newArrC :: (Int,...,Int) -> W a -> a
updateC :: (Ix,Int) -> W a -> W a
selectC :: Ix -> (Int -> W a) -> W a
```

A suitable axiomatization (in which the array representation is exposed as we did earlier) can be given by:

```
newArrC (x1,...,xn) c                    = newArrC <<x1,...,xn>>  c
newArrC <<...,xi,...>> (selectC i k)     = newArrC <<...,xi,...>> (k xi)
newArrC <<...,xi,...>> (updateC (i,y) c) = newArrC <<...,y,...>>  c
newArrC <<...,xi,...>> (valC b)          = b
```

These rules are shown graphically in Figure 3.2, and look surprisingly similar to those for the direct MADT. The technique for deriving them automatically is given in Section 4.1, and involves a variation of traditional CPS conversion. In addition, the proof that `updateC` can be implemented in constant time and space is a trivial variation of the proof given for `updateD`.

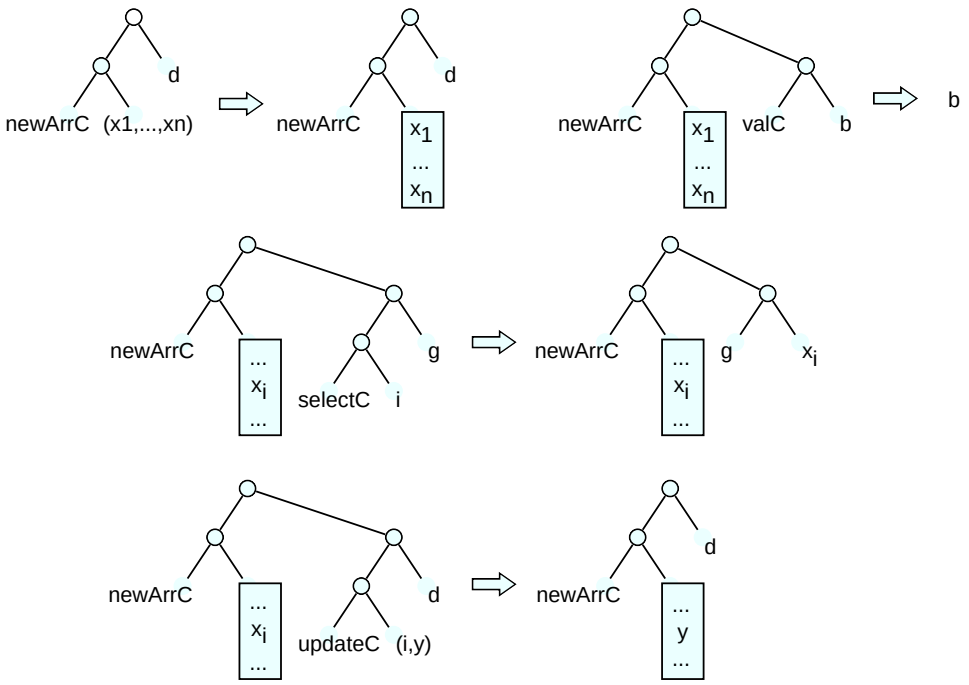

Figure 2: Graph Rewrite Semantics for CPS Array MADT

4 Derivation of MADT Axioms

In this section we present a translation from the equations defining the original ADT into ones defining each MADT. We give only the most difficult translation: that for the CPS MADT. The translations for both the direct and monadic MADT's are similar, but simpler.

Definition: Given a simple ADT with a set of generators G, mutators M, and selectors S, an *axiomatization* distinguishes a set $D = S \cup T$, where $T \subseteq M$. Each operator `d` $\in D$ has one or more defining equations of the form "`d x t = exp`", where `exp` is an ordinary expression involving operators on the domains `X`i, the conditional, and the operators of the ADT. If `d` is a mutator, `exp` will of course denote a value of the type-of-interest `T`, whereas for a selector `exp`'s type will be one of the auxiliary domains `X`$i \neq$ `T`.

Recall that our goal is to define *mutable* ADT's. In order to do this, we must constrain the axioms sufficiently to guarantee *linearity* of the type-of-interest, thus permitting efficient in-place update in the derived MADT's.

Definition: An axiomatization is *linear* if the type-of-interest satisfies Schmidt's *single-threadedness* condition [13].

The use of Schmidt's characterization of single-threadedness is arbitrary — others would do as well. An advantage of Schmidt's condition is that it is *simple*, even though it may not be as general as some others. The point here is not to define the most general class of axiomatizations possible, but rather a sufficiently rich class into which most axiomatizations can at least be translated. As mentioned in the introduction, common ADT's such as arrays, graphs, and queues can easily be

given linear axiomatizations. As an example, both axiomatizations of the array ADT given earlier are linear.

4.1 CPS MADT Translation

The translation scheme from a linear ADT axiomatization to an axiomatization for a CPS MADT looks, not surprisingly, somewhat like a conventional CPS conversion. (In the following, we use $\lceil \cdots \rceil$ and $\langle \cdots \rangle$ to quote and un-quote, respectively, the text being translated.)

First we define a translation $\mathcal{T}$ for expressions that, given an expression e and a continuation k, returns an equivalent expression using the CPS MADT:

$$\begin{array}{lcl}
\mathcal{T}\ \lceil \texttt{s}\ \langle x \rangle\ \langle t \rangle \rceil\ k & \Rightarrow & \mathcal{T}\ t\ \lceil \texttt{sC}\ \langle x \rangle\ \langle k \rangle \rceil \\
\mathcal{T}\ \lceil \texttt{m}\ \langle x \rangle\ \langle t \rangle \rceil\ k & \Rightarrow & \mathcal{T}\ t\ \lceil \texttt{cM}\ \langle x \rangle\ \langle k \rangle \rceil \\
\mathcal{T}\ \lceil \texttt{g}\ \langle x \rangle \rceil\ k & \Rightarrow & \lceil \texttt{gC}\ \langle x \rangle\ \langle k \rangle \rceil \\
 & & \\
\mathcal{T}\ \lceil \texttt{if}\ \langle p \rangle\ \langle c \rangle\ \langle a \rangle \rceil\ k & \Rightarrow & \lceil \texttt{if}\ \langle p \rangle\ \langle \mathcal{T}\ c\ k \rangle\ \langle \mathcal{T}\ a\ k \rangle \rceil \\
\mathcal{T}\ \lceil \texttt{x} \rceil\ k & \Rightarrow & \lceil \langle k \rangle\ \texttt{x} \rceil \\
\mathcal{T}\ \lceil \texttt{t} \rceil\ k & \Rightarrow & k \\
\mathcal{T}\ \lceil \texttt{con} \rceil\ k & \Rightarrow & \lceil \langle k \rangle\ \texttt{con} \rceil
\end{array}$$

Here `s`, `m`, and `g` represent an arbitrary selector, mutator, and generator, respectively; `sC`, `mC`, and `gC` are their CPS MADT counterparts; `t` is an identifier whose type is the type-of-interest; `x` is an identifier having auxiliary type; and `con` is a constant having auxiliary type.

Given $\mathcal{T}$, we can now define a translation $\mathcal{S}$ for equations defining selectors, and a translation $\mathcal{C}$ for equations defining mutators:

$$\begin{array}{lcl}
\mathcal{S}\ \lceil \texttt{s}\ \langle x_1 \rangle\ \langle t \rangle = \langle x_2 \rangle \rceil & \Rightarrow & \lceil\ \langle \mathcal{T} \lceil \texttt{s}\ \langle x_1 \rangle\ \langle t \rangle \rceil\ \lceil \texttt{k} \rceil \rangle = \langle \mathcal{T}\ x_2\ \lceil \texttt{\textbackslash y -> }\langle \mathcal{T}\ t\ \lceil \texttt{k y} \rceil \rangle \rceil \rangle\ \rceil \\
\mathcal{C}\ \lceil \texttt{m}\ \langle x \rangle\ \langle t_1 \rangle = \langle t_2 \rangle \rceil & \Rightarrow & \lceil\ \langle \mathcal{T} \lceil \texttt{m}\ \langle x \rangle\ \langle t_1 \rangle \rceil\ \lceil \texttt{k} \rceil \rangle = \langle \mathcal{T}\ t_2\ \lceil \texttt{k} \rceil \rangle\ \rceil
\end{array}$$

(We assume suitable α-renaming to avoid name clashes with `y` and `k`.) Note that $\mathcal{S}$ translates the right-hand-side in the context (i.e. continuation) of the nested type-of-interest on the left-hand-side. $\mathcal{C}$, on the other hand, does not — the type-of-interest in some sense "disappears," as it should, since we are trying to limit access to it. Indeed, note how it also disappears in the translation $\mathcal{T}$ for expressions.

Note also that we began with equations defining each operator in the set D, with the other operators being defined implicitly. However, the translation inverts this: elements in D are defined implicitly, with the others (which we collect into the set $\hat{D}$) defined directly.

To complete the translation scheme, one additional rule is required for each operator $d \in \hat{D}$ to capture its interaction with `valC`:

```
d x (valC y) = y                -- if d is a generator
d x (valC y) = valC y           -- if d is a mutator
```

The translation is now completely defined. As an example, applying it to the *first* array axiomatization given in Section 3 yields the following equations:

```
newArrC (...,xi,...) (selectC i k) = newArrC (...,xi,...) (k xi)
newArrC (...,xi,...) (valC x)      = x
updateC (j,x) (selectC i k) = if i==j then updateC (j,x) (k x)
                              else selectC i (\y-> updateC (j,x) (k y))
updateC (j,x) (valC y)      = valC y
```

We can read the first equation as saying: "initializing an array where the first operation in the 'rest of the program' is a `selectC`, is the same as not doing the select and just passing the initial value `xi` to the rest of the program." A similar reading can be made for each of the other equations. Overall, each `selectC` "bubbles up" to the most recent `updateC` of the same element (or to the initial `newArrC`), at which point it disappears; meanwhile `updateC`s are pushed "downward", and when they reach `val` they too disappear.[3]

Applying the translation to the *second* array axiomatization given in Section 3 yields:

```
newArrC (...,xi,...) (selectC i k)     = newArrC (...,xi,...) (k xi)
newArrC (...,xi,...) (updateC (i,y) c) = newArrC (...,y,...)  c
newArrC (...,xi,...) (val y)           = y
```

The structure of this translation is different from the one above because now $D = \{\texttt{select}, \texttt{update}\}$ rather than just $\{\texttt{select}\}$. The resulting axioms are, however, identical to the rewrite rules given in Section 3.2, except that there we exposed the hidden array representation using one extra rule for `newArrC`. If that rule were used to expand each occurrence of `newArrC` above, as we did for the direct MADT derivation given in Section 3.1, the precise rewrite rules of Section 3.2 would result.

4.2 Correctness

The correctness of the various axiomatizations can be proven by establishing a relationship between the new operators and the original ones.

Theorem: Given the original array ADT and its second axiomatization in Section 3, the following implementation of the direct MADT satisfies the axiomatization given in Section 3.1:

```
newArrD xs      =   MkD0 (newArr xs)
updateD (i,x)   =   MkD1 (update (i,x))
selectD i       =   MkD2 (select i)
```

Intuitively this is the strongest relationship we can expect; namely, that the direct MADT behaves just like the original ADT except that the state is hidden. The proof of this theorem amounts to a proof of each direct MADT axiom in turn, using the definitions of `begin`, `get`, and `$`, along with the original axioms for `newArr`, `select`, and `update`.

[3] We note an interesting aspect of the CPS conversion scheme: mutators and selectors trade roles! I.e., the CPS transformation uncovers a "duality" between constructors and selectors, much like the duality noted by Filinski in [1]. Performing the translation on other ADT's yields interesting results. For example, a CPS MADT for a list datatype has `car` and `cdr` as constructors, and `cons` and `nil` as selectors, as we will see in Section 5.3.

Proof:

```
(1) begin (newArrD (..xi..)) (get (selectD i) g)
    = begin (MkD0 (newArr (..xi..))) (get (MkD2 (select i)) g)
    = begin (MkD0 (newArr (..xi..))) (MkD1 (\a-> let MkD1 h = g (select i a) in h a
    = (\a-> let MkD1 h = g (select i a) in h a) (newArr (..xi..))
    = let MkD1 h = g xi in h (newArr (..xi..))
    = begin (MkD0 (newArr (..xi..))) (g xi)
    = begin (newArrD (..xi..)) (g xi)

(2) begin (newArrD (..xi..)) (updateD (i,y) $ d)
    = begin (mkD0 (newArr (..xi..))) (MkD1 (update (i,y)) $ d)
    = begin (mkD0 (newArr (..xi..))) (MkD1 (let MkD1 g = d in g . update (i,y)))
    = (let MkD1 g = d in g . update (i,y)) (newArr (..xi..))
    = (\a-> let MkD1 g = d in g (update (i,y) a)) (newArr (..xi..))
    = let MkD1 g = d in g (newArr (..y..))
    = begin (MkD0 (newArr (..y..))) d
    = begin (newArrD (..y..)) d

(3) begin (newArrD (..xi..)) (val b)
    = begin (MkD0 (newArr (..xi..))) (MkD1 (\a-> b))
    = (\a -> b) (newArr (..xi..))
    = b
```

□

A version of this theorem can also be stated for the general translation schemes. The one for the direct MADT translation is similar to the above, whereas for the CPS MADT it takes the following form:

Theorem: Given an ADT $\langle \mathtt{g}, \mathtt{m}, \mathtt{s} \rangle$ with axiomatization A, and its CPS MADT translation $\langle \mathtt{gC}, \mathtt{mC}, \mathtt{sC} \rangle$ with derived axiomatization A', the following implementation of the MADT satisfies A':

```
gC x c   =   let C f = c in f (g x)
mC x c   =   C (\t-> let C f = c in f (m x t))
sC x k   =   C (\t-> let C g = k (s x t) in g t)
```

5 Variations on a Theme

Many variations and extensions of the methodology introduced here are possible. In this section we outline a few of them.

5.1 Haskell Arrays

Haskell arrays are *monolithic*: the elements are specified all at once via an *array comprehension* that is similar to a list comprehension. The basic array-former, called `array`, has type:

```
data Assoc a b  =  a := b
array :: (Ix a) => (a,a) -> [Assoc a b] -> Array a b
```

(Note: "`Ix`" here is the *class* `Ix` as defined in Haskell, as opposed to some index type `Ix` as used earlier in this paper.) `a` is the index type and `b` is the element type. The first argument is a pair of upper and lower bounds, and the second argument is a list of index/value pairs, usually written as a list comprehension. An immutable array of type `Array a b` is returned. For example, a vector of squares of the first 10 integers is given by:

```
array (1,10) [ i := i*i | i <- [1..10] ]  -- [1..10] is the list of
                                          -- integers from 1 to 10
```

It is possible to define Haskell arrays in terms of a more primitive array MADT. This has several advantages when optimizing programs. For example, it makes it easier to express optimizations such as turning the list comprehension argument into some kind of "loop". In addition, it allows us to express Haskell's incremental array update operator in an efficient way. This operator takes the array to be updated and a list of index/value pairs representing the desired updates. Using a MADT we can express the fact that the resulting operation may incur an initial copy of the old array, but then each update is done in constant time and space.

The array CPS MADT that we need to accomplish this is shown below:

```
data C a b
data PrimArr a

newArrC ::  Int -> a -> C a b -> b
updateC ::  Int -> a -> C a b -> C a b
selectC ::  Int -> (a -> C a b) -> C a b
valC    ::  a -> C b a
returnC ::  C a (PrimArr a)
select  ::  PrimArr a -> Int -> a
copyArrC::  PrimArr a -> C a b -> b
```

This design permits *polymorphic* arrays; thus instead of the continuation type "`W a`" used earlier, we have the type "`C a b`", which is the continuation type for arrays containing elements of type `a` and returning final result of type `b`. But note that although polymorphic in the element type, these arrays are still monomorphic in the index type (`Int`). In contract, a Haskell array can be defined using any type as index as long as it is in the class `Ix`. The implementation is simplified, however, by mapping any element of such a type into the `Int` domain. In fact, Haskell provides pre-defined functions to do just that.

The type "`PrimArr a`" is the abstract type of pure polymorphic arrays indexed by integers. The operation `returnC` is intended to end a computation on arrays (as does `valC`), but it returns the array itself as the final value. `select` is a pure function which reads from a pure array. Finally, `copyArrC` is like `newArrC` in that it begins a computation on an array, but it creates a copy of its pure array argument as the intial value for the hidden array.

Given this MADT, Haskell's array operators are easily defined, and are given in Appendix A. Included there is the Common Lisp code needed to make this work using Yale Haskell's general Common Lisp interface mechanism.

5.2 Exceptions

Programming with continuations has become more comfortable within certain communities (in particular, the Scheme and denotational semantics communities). It is well-known and straightforward to define the equivalent of loops, procedure calls, exceptions, etc. using continuations. For example, consider handling an exception such as index out-of-bounds for an array. To introduce errors into a conventional ADT one would typically add an error element to the domain of answers, along with rules that not only give rise to the error, but also *propagate* them once they occur. With continuations, however, we can do better: we can realize the essence of exceptions via "non-local exits." To see how in the case of arrays, we first define a datatype `Maybe`:

```
data Maybe a = OK a | OutOfBounds
type W a = C (Array -> Maybe a)           -- hidden
```

and then redefine our CPS MADT as follows:

```
newArrE :: (Int,Int,...,Int) -> W a -> Maybe a
updateE :: (Ix,Int) -> W a -> W a
selectE :: Ix -> (Int -> W a) -> W a
```

The resulting MADT can be implemented such that when `updateE` or `selectE` encounters an out-of-bounds condition, it can simply *ignore its continuation*, thus effecting a "non-local exit." This also prevents cluttering up the types of intermediate results with error values — for example, note that the second argument to `selectE` has type `Int -> W a` rather than `Maybe Int -> W a`.

To carry this a step further, one might argue that it would be better to have some kind of *exception handling* mechanism. Indeed, this is easily done by supplying an *error continuation* as an argument to each operator, to act as a local exception handler:[4]

```
type W a = C (Array -> a)           -- hidden
type EC a = W a
newArrH :: (Int,Int,...,Int) -> W a -> a
updateH :: (Ix,Int) -> EC a -> W a -> W a
selectH :: Ix -> EC a -> (Int -> W a) -> W a
```

Note that the `maybe` datatype is no longer fixed into the design. The user might wish to utilize it or not, depending on the application; i.e. depending on what the desired return type is. Below is an example where it *is* used: two potential error sites are handled in different ways; one returns an error message, the other simply aborts and (effectively) ignores the error, returning a default value `d`.

[4] This use of both a success and error continuation is reminiscent of Haskell's continuation-based I/O.

```
newArrH (1,2,3)                $
...
updateH 4 0 (valH OutOfBounds) $
...
selectH 4 0 (valH (OK d))      $
...
```

In closing, we note that the array ADT only has one kind of error: out-of-bounds. In an ADT design where more than one type of error is possible, the error continuation could take an error message as an argument.

5.3 Lists, Stacks, and Queues

Consider this standard axiomatization of a *polymorphic list* datatype:

```
nil  :: List a
cons :: a -> List a -> List a
car  :: List a -> a
cdr  :: List a -> List a

car (cons x l) = x
car  nil       = error
cdr (cons x l) = l
cdr  nil       = error
```

The following CPS MADT is derived from the above axioms using the translation scheme given in Section 4.1 (note: `car` is treated as a selector, whereas `cdr` is treated as a *mutator*):

```
type W a = List a -> a          -- hidden
nilC  :: W a -> a
consC :: a -> W a -> W a
carC  :: (a -> W a) -> W a
cdrC  :: W a -> W a
val   :: a -> W a

consC x (carC k) = consC x (k x)
consC x (cdrC c) = c
consC x (val y)  = val y
nilC (carC k)    = error
nilC (cdrC k)    = error
nilC (val y)     = y
```

Note that the original ADT exhibits constant-time complexity for each of its operations. But the CPS MADT exhibits the same complexity. So it seems as if there is no advantage to the MADT

version. But note further that the hidden list in the MADT version is still handled "linearly", and thus we gain the following benefit: an implementation is free to immediately "reclaim" cons cells that are "discarded" by cdr. In other words, no garbage collection is needed to implement the list! This may be important in certain real-time applications, for example.

To realize another advantage, we can extend the ADT with two other operations, `nth-sel` and `nth-upd`, that select and update, respectively, the nth element of a list:[5]

```
nth-sel :: Int -> List a -> a
nth-upd :: Int -> a -> List a -> List a

nth-sel 0 (cons x xs) = x
nth-sel n (cons x xs) = nth-sel (n-1) xs
nth-sel n  nil        = error

nth-upd 0 a (cons x xs) = cons a xs
nth-upd n a (cons x xs) = cons x (nth-upd (n-1) a xs)
nth-upd n a  nil        = error
```

Their CPS MADT versions, again derived using the results in Section 4.1, are given by:

```
nth-sel :: Int -> (a -> W a) -> W a
nth-upd :: Int -> a -> W a -> W a

consC x (nth-selC 0 k)   = consC x (k x)
consC x (nth-selC n k)   = nth-selC (n-1) (\y-> consC x (k y))
consC x (nth-updC 0 a c) = consC a c
consC x (nth-updC n a c) = nth-updC (n-1) a (consC x c)
consC x (val a)          = val a

nilC (nth-selC n k)   = error
nilC (nth-updC n a c) = error
nilC (val a)          = a
```

Both these and the original operators take time proportional to `n`, but as with the earlier list operators, they can be implemented without the need for garbage collection; in particular, the updates can be done "in-place", and no new "consing" is incurred.

By taking advantage of the well-known isomorphism between lists and *stacks*, we can associate `cons` with `push`, `car` with `top`, and `cdr` with `pop`, thus implementing a stack MADT. The use of such a stack is very similar to its use in an imperative langauge.

[5]Similar operators were suggested by George Nelan in a message to the Haskell mailing list in April, 1993.

To realize *queues*, we could start with a standard axiomatization as given by:

```
newQ :: Queue a
addQ :: a -> Queue a -> Queue a
remQ :: Queue a -> Queue a
fstQ :: Queue a -> a

fstQ (addQ a newQ) = a
fstQ (addQ a q)    = fstQ q
fstQ  newQ         = error
remQ (addQ a newQ) = newQ
remQ (addQ a q)    = addQ a (remQ q)
remQ  newQ         = error
```

It is straightforward to derive a MADT from these axioms, and it will share the garbage-collection-free property of the list MADT discussed earlier. However, as for the first array axiomatization that we considered in Section 3, we cannot reduce the linear-time complexity of `fstQ` and `remQ`. To accomplish contant-time complexity for both insertion and deletion in the *conventional* way,[6] we would need some mechanism for having two pointers, one to the front and one to the end of the queue. This is best achieved by having a general mechanism for building arbitrary graphs; such a mechanism is described in the next section.

5.4 General Heaps

A *store* is a structure which associates locations with values. In denotational semantics it is considered to be a function; in a simple interpreter it may be a list of index/value pairs; and in a real computer it is the memory, and is best thought of as a large vector indexed by location and updated in constant time and space. Thus the array MADT described earlier can be used to simulate a store efficiently in a pure functional language.

A *heap* is somewhat more abstract – it also associates locations (but now usually called *pointers* or *references*) with values, but is assumed to be (at least conceptually) *unbounded* in size.[7] A heap also provides an abstract mechanism for allocating new, fresh references. Heaps are usually implemented by an underlying store with constant-time access and update, and a mechanism to perform garbage collection to give the impression of infiniteness.

With a heap one can build general, dynamic graphs in a straightforward manner. Thus if we could devise a linear axiomatization for a heap we would be able to build general, mutable graphs in a pure functional language by using a MADT version of the heap. This axiomatization should abstract away the details of the underlying implementation. One way to do this is to represent the heap as a pair: an infinite list of unique references, and an infinite store. The resulting set of operations would have functionality:

[6] It is possible, of course, to achieve constant-time complexity for these operations in a functional way not relying on MADT's, although it is somewhat unconventional.

[7] This notion of heap is that used in programming language circles, *not* the (usually tree-shaped) heap described in data structures textbooks.

```
mkHeap :: List Refs -> Store a -> Heap a     -- basic heap former
alloc  :: Heap a -> (Ref, Heap a)            -- allocated a new pointer
update :: Ref -> a -> Heap a -> Heap a       -- updates a cell
lookup :: Ref -> Heap a -> a                 -- retrieves contents of a cell
```

An appropriate (linear) axiomatization is given by:

```
alloc (mkHeap (cons ref refs) store) = (ref, mkHeap refs store)
update ref y (mkHeap refs (...(ref,x)...)) = mkHeap refs (...(ref,y)...)
lookup   ref (mkHeap refs (...(ref,x)...)) = x
```

We assume that `update` and `lookup` can be implemented in constant time and space. (In the context of an implementation using garbage collection, an ammortization argument is needed, but that does not concern us here.)

The CPS MADT version of these axioms is given by:

```
mkHeapC :: List Refs -> Store a -> W a -> a
allocC  :: (Ref -> W a) -> W a
updateC :: Ref -> a -> W a -> w a
lookupC :: Ref -> (a -> W a) -> W a

mkHeapC (cons ref refs) store (allocC k)       = mkHeapC refs store (k ref)
mkHeapC refs (...(ref,x)...) (updateC ref y c) = mkHeapC refs (...(ref,y)...) c
mkHeapC refs (...(ref,x)...) (lookupC ref k)   = mkHeapC refs (...(ref,x)...) (k x)
mkHeapC refs store (valC x)                    = x
```

Graphs can be created by assuming the existence of two auxiliary values: `allRefs`, an infinite list of unique references; and `newStore`, a store each of whose elements is bound to *error* (or $\perp$). Then define:

```
newGraph = mkHeapC allRefs newStore
```

For example, the two-node circular graph with node labels `"a"` and `"b"` shown in Figure 5.4 can be created by the following code:

```
newGraph           $
allocC             $ \r1->
allocC             $ \r2->
updateC r1 ("a",r2) $
updateC r2 ("b",r1) $
...
```

This graph has "hidden" type `Graph (String,Ref)`.

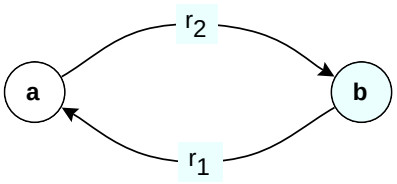

Figure 3: Simple Circular Graph

5.4.1 The Power of References

The idea of a reference is rather pervasive. Aside from using references to build arbitrary graphs, as described above, they can be used simply as assignable cells, as in ML. Furthermore, they can be used as handles to more than one array (or other contiguous data structure). More specifically, the array MADT's presented thus far can only define one array at a time, which is a serious limitation. By introducing the notion of an *array reference*, however, we can redesign the polymorphic array CPS MADT, for example, to handle more than one array:

```
newArrCR :: W b -> b
allocCR  :: (a,a,...,a) -> (ArrRef a -> W b) -> W b
selectCR :: ArrRef a -> Ix -> (a -> W b) -> W b
updateCR :: ArrRef a -> Ix ->  a -> W b  ->  W b
valCR    :: b -> W b
```

`allocCR` allocates a new array with the given values, and returns a *reference* to that array. For example:

```
newArrCR            $
allocCR (True,True) $ \a1->
allocCR ('a','a',)  $ \a2->
updateCR a1 1 False $
updateCR a2 1 'b'   $
...
```

Note that the two arrays are *independently* polymorphic, as are the array references.

An alternative way to deal with more than one array is as follows: Assume a pure array ADT (such as pre-defined in Haskell), and a way to create single references such as described earlier. The pure array can then be initialized with a set of unique vectors. To update the array in constant time and space, select the desired reference from the pure array and update it!

5.4.2 Are References Safe?

The astute reader may be concerned that we may suddenly lose confluence with the graph and multiple-array MADT's just presented, since array references are now "first class", and thus may

be returned as a final result via `valC`. Such an "escaping" reference might then be used in some other context in an insidious manner![8] However, by returning to the axioms we see that no fundamental harm is done: That escaping reference is simply an element of `allRefs`, and indeed it *may* be used in some other context, even though that use may have been unintentional on the user's part. This is analogous to using the wrong index into an array in an imperative language.

Nevertheless, one might ask if it is possible to improve on this situation. Abstractly, what we desire is a solution that creates a new, fresh supply of unique names for every invocation of `mkHeap`, rather than always using the fixed supply `allRefs`. A reference created in one context (i.e. in one instantiation of `mkHeap`) will then cause a run-time error if used in some other context.

Alternatively, in a concrete implementation, one could generate a single unique "label" for every invocation of `mkHeap`. This label is initially attached to the hidden store. Newly allocated references are also tagged with this label. When an update or selection occurs, the tag of the reference is compared with the tag of the hidden store, and a run-time error results if they are not the same.

Unfortunately, although this is a pragmatic solution, the only way to axiomatize it is to "thread" some kind of name supply through the whole program. This seems undesirable. A more promising solution might be found in the recent work on adding unique name generation (ala Lisp's "gensym") to the lambda calculus (see, for example, [10] and [12]).

6 Final Comments

Although using MADT's requires a shift in programming style, they have the advantages of guaranteed, predictable performance, and simple equational reasoning. When used in a functional language such as Haskell, a function's conventional type alone indicates the kinds of "effects" that it will induce.

Regarding functionality, we make the final observation that the reduction rules for all forms of the array MADT imply that both `select` and `update` are *strict* in their index arguments, but that `update` is *not* strict in its array-element argument. Thus the resulting arrays are non-strict in their elements (which is the best that any current functional language can do).

Their most serious drawback is the need to program in a "continuation passing style" (even the direct and monadic MADT's are somewhat continuation-oriented). On the other hand, this requirement does not permeate an entire program: it is only necessary in those portions of a program dealing directly with the state. Furthermore, both functional programmers and Lisp programmers have gained a fair amount of experience programming with and reasoning about continuations in recent years, most notably as a result of continuation-based I/O in languages such as Haskell and Hope, the use of `call/cc` in Scheme, and the use of continuations in denotational semantics.

A question to consider is which of the various MADT's is best? They all seem to be equal in expressive power, yet stylistically have different features to offer. The direct and CPS MADT's represent extremes in at least one dimension of the solution space, the former using continuation-based combinators, the latter embedding the continuation-like behavior into the operators themselves. In

[8] This same problem arises in [9] and [14].

this sense the monadic MADT appears to be a hybrid between these extremes; indeed, other hybrids are also possible. In a recent paper Peyton Jones and Wadler show that our CPS version of an array MADT can be implemented with a monadic version [11]. The reverse simulation can also done, but, curiously, not in a language using a Hindley-Milner type system.

7 Acknowledgements

Thanks to Dan Rabin, Martin Odersky, Uday Reddy, Phil Wadler, Simon Peyton Jones, and George Nelan for comments on previous drafts of this paper. Thanks also to Chih-Ping Chen for implementing CPS MADT's in Yale Haskell.

Appendix A: Implementation of Haskell Arrays in Terms of CPS MADT

```
--------------- Haskell Array Implementation --------------------------------
module PreludeArray (Array, Assoc((:=)), array, (!), listArray, bounds,
                     indices, elems, assocs, (//), accumArray, accum, amap,
            ixmap) where

import CPSMADTArray (PrimArr, newArrC, updateC, selectC,
                     returnC, select, copyArrC)

infixl 9 !
infixl 9 //
infix  1 :=

data Assoc a b = a := b deriving (Eq, Ord, Ix, Text, Binary)
data (Ix a) => Array a b = MkArray (a,a) (PrimArr b)

-- Function signatures
array        :: (Ix a) => (a, a) -> [Assoc a b] -> Array a b
(!)          :: (Ix a) => Array a b -> a -> b
listArray    :: (Ix a) => (a,a) -> [b] -> Array a b
bounds       :: (Ix a) => Array a b -> (a, a)
indices      :: (Ix a) => Array a b -> [a]
elems        :: (Ix a) => Array a b -> [b]
assocs       :: (Ix a) => Array a b -> [Assoc a b]
(//)         :: (Ix a) => Array a b -> [Assoc a b] -> Array a b
accumArray   :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
accum        :: (Ix a) => (b -> c -> b) ->Array a b -> [Assoc a c] -> Array a b
amap         :: (Ix a) => (b -> c) -> Array a b -> Array a c
ixmap        :: (Ix a, Ix b) => (a,a) -> (a -> b) ->Array b c -> Array a c

unInitializedError = error "(!) uninitialized element!"
```

```
-- Function definitions
array bounds@(low,high) ivs =
         let g (i := x) next = updateC (index bounds i) x $ next
             size = (index bounds high) + 1
             operations = foldr g returnC ivs
         in MkArray bounds (newArrC size unInitializedError $ operations)

(MkArray bounds vec) ! i = select vec ind
                           where ind = index bounds i

listArray bounds@(low,high) vs     =
          let size = (index bounds high) + 1
              g x next j = updateC j x $ next (j+1)
              operations = foldr g (\_ -> returnC) vs 0
          in MkArray bounds (newArrC size unInitializedError $ operations)

bounds (MkArray bounds _) = bounds

indices          = range . bounds

elems (MkArray bounds@(low,high) vec) =
          let sizeMinus1 = index bounds high
          in foldr (\ind rest -> select vec ind : rest) [] [0..sizeMinus1]

assocs (MkArray bounds vec) =
          let g vind rest j = (vind := select vec j) : rest (j+1)
          in foldr g (\_ -> []) (range bounds) 0

(MkArray bounds vec) // us =
         let g (i := x) next = updateC (index bounds i) x $ next
             operations = foldr g returnC us
         in MkArray bounds (copyArrC vec $ operations)

accumArray f z bounds@(low,high) as =
         let size = index bounds high + 1
             g (a := b) next =
               let ind = index bounds a
               in  selectC ind $ \x -> updateC ind (f x b) $ next
             operations = foldr g returnC as
         in MkArray bounds (newArrC size z $ operations)

accum f (MkArray bounds vec) as =
         let g (a := b) next =
               let ind = index bounds a
```

```
              in  selectC ind $ \x -> updateC ind (f x b) $ next
            operations = foldr g returnC as
         in MkArray bounds (copyArrC vec $ operations)

amap f (MkArray bounds@(low,high) vec) =
              let size = (index bounds high) + 1
          g ind next =
                      let x = select vec ind
                      in  updateC ind (f x) $ next
                  operations = foldr g returnC [0..size-1]
              in MkArray bounds (newArrC size unInitializedError $ operations)

ixmap b f a      = array b [i := a ! f i| i <- range b]

-- Declare Array as an instance of class Eq.
instance  (Ix a, Eq b)  => Eq (Array a b)  where
    a == a'             =  assocs a == assocs a'

-- Declare Array as an instance of class Ord.
instance  (Ix a, Ord b) => Ord (Array a b)  where
    a <=  a'            =  assocs a <=  assocs a'

-- Declare Array as an instance of class Text.
instance  (Ix a, Text a, Text b) => Text (Array a b)  where
    showsPrec p a = showParen (p > 9) (
                    showString "array " .
                    shows (bounds a) . showChar ' ' .
                    shows (assocs a)                 )

    readsPrec p = readParen (p > 9)
           (\r -> [(array b as, u) | ("array",s) <- lex r,
                                     (b,t)       <- reads s,
                                     (as,u)      <- reads t   ]
                  ++
                  [(listArray b xs, u) | ("listArray",s) <- lex r,
                                         (b,t)           <- reads s,
                                         (xs,u)          <- reads t ])

--------------- Haskell/Lisp Interface ---------------------------------------
interface CPSMADTArray where

data C a b
data PrimArr a

newArrC ::  Int -> a -> C a b -> b
```

```
updateC ::  Int -> a -> C a b -> C a b
selectC ::  Int -> (a -> C a b) -> C a b
valC    ::  a -> C b a
returnC ::  C a (PrimArr a)
select  ::  PrimArr a -> Int -> a
copyArrC::  PrimArr a -> C a b -> b

{-#
selectC :: LispName("prim.selectc")
updateC :: Strictness("S,N,S"),
                   LispName("prim.update")
newArrC :: Strictness("S,N,S"),
                   LispName("prim.make")
valC    :: LispName("prim.value")
returnC :: LispName("prim.return")
select  :: LispName("prim.select")
copyArrC:: LispName("prim.copy")
#-}

--------------- Lisp Array MADT Implementation ------------------------------
(define-integrable (prim.selectc i kont)
  (lambda (array)
    (let ((g (funcall kont (vector-ref array i))))
      (funcall g array))))

(define-integrable (prim.update i newval cont)
  (lambda (array)
    (setf (vector-ref array i) newval)
    (funcall cont array)))

(define-integrable (prim.make n initval cont)
  (funcall cont (make-vector n initval)))

(define-integrable (prim.value val)
  (lambda (array) val))

(define-integrable (prim.select array i)
  (force (vector-ref array i)))

(define-integrable (prim.copy array cont)
  (funcall cont (vector-copy array)))

(define-integrable prim.return
  (lambda (array) array))
```

References

[1] A. Filinski. Declarative continuations: An investigation of duality in programming language semantics. In D. H. Pitt, editor, *Category Theory and Computer Science*, pages 224–249. Springer-Verlag, Berlin, 1989. (Lect. Notes in Comp. Science Vol. 389).

[2] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of Symposium on Logic in Computer Science*, pages 333–343. IEEE, June 1990.

[3] P. Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 177–189. Cambridge University Press, 1982.

[4] S. Holmstrom. A linear functional language. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages, PMG Report 53*, pages 13–32, 1988.

[5] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[6] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional i/o systems. Technical Report YALEU/DCS/RR-665, Yale University Department of Computer Science, December 1988.

[7] K. Karlsson. Nebula, a functional operating system. Technical report, Chalmers University, 1981.

[8] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[9] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda-calculus. Research Report YALEU/DCS/RR-929, Department of Computer Science, Yale University, New Haven, Connecticut, October 1992. To appear in Proc. 20th ACM Symposium on Principles of Programming Languages.

[10] Martin Odersky. A syntactic theory of local names. Technical Report TR-965, Yale University, May 1993.

[11] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993. (to appear).

[12] A. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-968, June 1993.

[13] D. A. Schmidt. Detecting global variables in denotational specification. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[14] V. Swarup, U. Reddy, and E. Ireland. Assignments for applicative languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM/IFIP, 1991.

[15] David Turner. *Functional Programming and Communicating Processes*, volume 259 of *Lecture Notes in Computer Science*, pages 54–74. Springer Verlag, 1987.

[16] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

[17] P. Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 255–273. ACM/IFIP, 1991.

[18] P. Wadler. The essence of functional programming. In *Proceedings 19th Symposium on Principles of Programming Languages*, pages 1–14. ACM, January 1992.

[19] D. Wakeling and C. Runciman. Linearity and laziness. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM/IFIP, 1991.