# Mutable Abstract Datatypes

Paul Hudak
Research Report YALEU/DCS/RR-914
December 1992

# Mutable Abstract Datatypes

Paul Hudak

Yale University
Department of Computer Science
New Haven, CT 06520
hudak@cs.yale.edu

December 1992

## 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 "purely" functional language such as Haskell, Hope, Miranda, or even a pure subset of ML or Scheme?

One way 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 [6, 3, 12], the "single-threaded" type system of Guzman/Hudak [2], and the "stratified" type system of Swarup/Reddy/Ireland [10]. 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 [13].

In contrast, 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 [11]. This was the first example of a purely functional treatment of state for which efficient performance could be guaranteed without resorting to an extended type system. However, many questions were left unanswered in Wadler's work (the primary emphasis being on the monadic

style of programming). For example, is a monadic solution to 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 for which an efficient implementation exists that does 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. MADT's have the obvious advantage of guaranteed, predictable performance for every operation. 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 [4] syntax for our examples throughout; occasional comments on syntax are included to aid the reader. Although only one version of the CMADT's (the CPS version) has been fully implemented, all of the code in this document has been executed under the Yale Haskell implementation to ensure at least functional correctness.[1]

## 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 `Xi` $\neq$`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:

[1] Although we do not preclude the possibility of typographical errors during manuscript preparation!

```
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]

[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!

```
($) :: 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):

```
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)
val :: 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 `val :: a -> W a`, below:

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

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

f $ a = f a                         -- infix application (note: nothing hidden)
val 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->
  val x                 )
```

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

### 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. Using the syntax `<<x1,x2,...,xn>>` for this purpose, we introduce one additional rule:[3]

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

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

```
get (selectD i) g   <<...,xi,...>> = g xi <<...,xi,...>>
(updateD (i,y) $ d) <<...,xi,...>> = d    <<...,xi,...>>
val b               <<...,xi,...>> = 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 `newArrD` is directly proportional to the size of the array. Using this fresh, but otherwise "hidden" array, the operators `selectD` and `valD` 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 state argument, is confluent.

**Proof:** 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

[3] This rule is actually ill-typed (the right-hand-side being an application of type `D1(Arr->Arr)` to type `Arr`), but because this is a hidden implementation no harm is done. It is straightforward to regain well-typedness, but doing so only clutters the exposition.

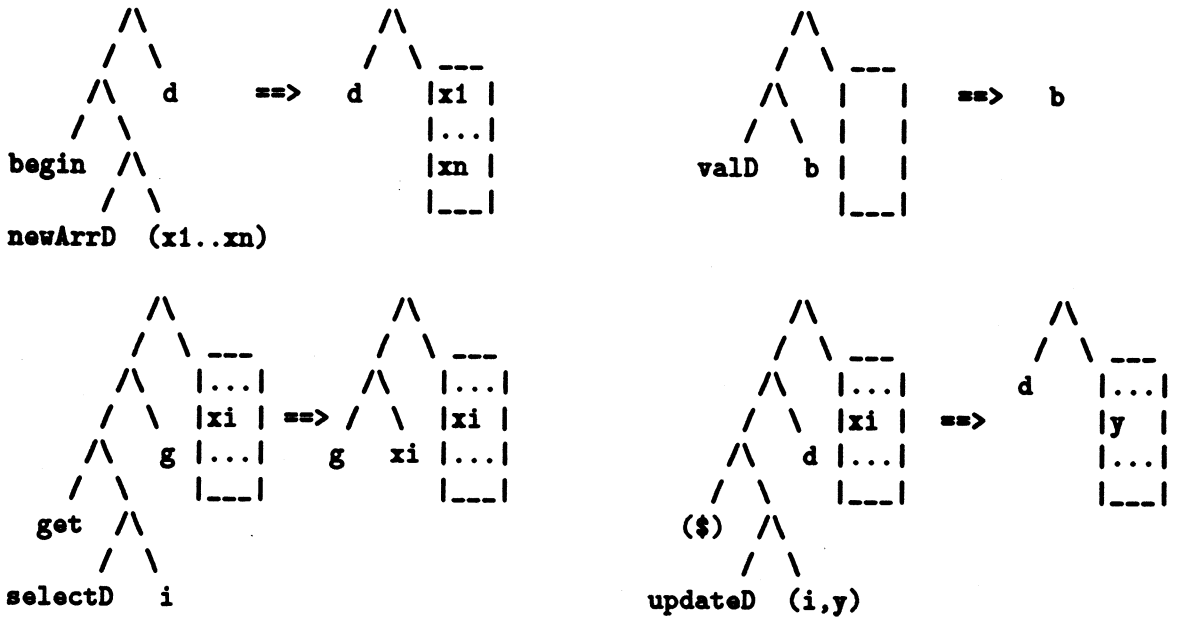


Figure 1: Graph Rewrite Semantics for Direct Array MADT

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.

## 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 above) can be given by:

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

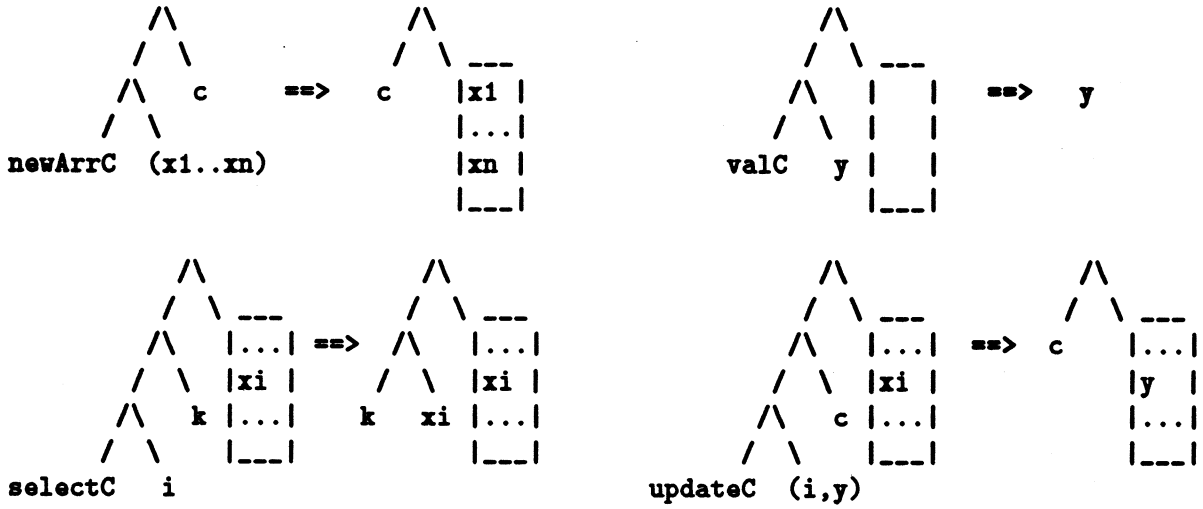


Figure 2: Graph Rewrite Semantics for CPS Array MADT

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**.

# 4 Derivation of MADT Axioms

In this section we present a translation from the equations defining the original ADT into ones defining each MADT. [Because of lack of space, 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 [9].

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:

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

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:

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

(We assume suitable $\alpha$-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 **val**:

```
d x (val y) = y          -- if d is a generator
d x (val y) = val 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 (selectC i k)        = newArrC (k 0)
newArrC (val 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) (val y)        = val 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 0 to the rest of the program." A similar reading can be made 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.[4]

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

```
newArrC (...,xi,...) (selectC i k)      = newArrC (...,xi,...) (k 0)
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 $\{\texttt{select}\}$. It is also slightly different from the rewrite rules given in Section 3.2, but only because there we exposed the hidden array representation using the rule:

```
newArrC (x1,...,xn) c          = c <<x1,...,xn>>
```

If this rule is 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 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. For example:

**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)
```

[4]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 as selector!

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**.

**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. It is quite easy to make our arrays polymorphic, for example. We have also derived MADT's for hash tables, queues, graphs, and search trees. Although using MADT's requires a shift in programming style, they have