

Rolling Your Own Mutable ADT

— A Connection between Linear Types and Monads —

Chih-Ping Chen and Paul Hudak

Yale University
Department of Computer Science
New Haven, CT 06520
{chen-chih-ping, hudak}@cs.yale.edu

Abstract

A methodology is described whereby a linear ADT may be rigorously encapsulated within a state monad. A CPS-like translation from the original ADT axioms into monadic ones is also described and proven correct, so that reasoning can be accomplished at the monadic level without exposing the state. The ADT axioms are suitably constrained by a linear type system to make this translation possible. This constraint also allows the state to be “updated in place,” a notion made precise via a graph-rewrite operational semantics.

1 Introduction

In recent years, numerous proposals for I/O, destructive updates to data structures, mutable variables, non-determinism, and concurrency have been put forth, all using *monads* to structure programs in such a way that details of the computation are effectively hidden and encapsulated [18, 20]. One of the most important uses of monads is in dealing with *state*, resulting in a style of programming referred to appropriately by Peyton Jones and Wadler as *imperative functional programming* [16]. In this style, a *state monad* is used to encapsulate the state in an abstract datatype, thus avoiding the “plumbing” usually associated with pure functional programs that manipulate state. Access to the hidden state is achieved by abstract operations that are sequenced via a small set of monadic combinators, typically called *bind* and *unit*. This style of programming has been adopted wholeheartedly by the Haskell [8] community, and its

utility has been verified through many non-trivial applications.

Surprisingly, however, little work has been done in establishing a formal connection between monads and state. In particular:

- There are few formal connections between monads and *single-threaded* state, where the property of being single-threaded is precisely what is needed to guarantee safe, “in-place” update. The severity of this problem is easily demonstrated by an example: For any state-transformer monad with type $M a = \mathcal{S} \rightarrow (\mathcal{S}, a)$, there is a trivial operation $getState :: M \mathcal{S}$ that will instantly destroy any hope for single-threadedness:

$$getState\ s = (s, s)$$

- Nor are there many formal connections between monads and the state-manipulating operations themselves. That is, the standard monad operations tell us how the state-manipulating operations may be sequenced, but say nothing about what they do, or how they are derived.
- Finally, although laws for reasoning about several specific monads exist [18, 15], there is no general way to derive these laws.

To solve these problems, we outline a methodology that allows one to begin with a conventional axiomatization of a type of interest, rigorously encapsulate it within a monad, and reason about it abstractly in a monadic fashion. This is only possible if the axiomatization satisfies a certain linearity condition, established using linear types [6, 9, 4, 19, 5]. This same condition allows us to prove that the type of interest (i.e. the state) may be updated in place.

Our combination of linear types and monads unveils a connection between two very different methods for

dealing with state in a functional language. It is interesting that a CPS-like conversion, with strong ties to notions of sequentiality, formalizes this connection.

From a pragmatic perspective, the methodology allows one to use conventional ADT techniques to design a type of interest and operations on it. The use of the linear type system is confined *just to the ADT axioms*. Once the axioms type-check, the linear type system never needs to be dealt with again. In other words, the monad encapsulates not just the ADT, but its linearity as well.

The methodology is also easily implementable (although we have not yet done so). The graph-rewrite semantics is quite similar to conventional graph reduction, the basis of many functional language implementations. Moreover, the monadic axioms can be derived automatically, yielding a source of program transformation rules for compile-time optimization. These rules may also be used by semantics-directed compilers and interpreters that are based on monads [12].

An Example. Consider a simple *integer list* ADT, with operations:

```

nil      :: IntList
cons     :: Int → IntList → IntList
nth-select :: Int → IntList → Int
nth-update :: (Int, Int) → IntList → IntList

```

These operations are axiomatized by:

```

nth-select 0 (cons x xs)      ⇒ x
nth-select (i + 1) (cons x xs) ⇒ nth-select i xs
nth-update (0, v) (cons x xs) ⇒ cons v xs
nth-update (i + 1, v) (cons x xs)
  ⇒ cons x (nth-update (i, v) xs)

```

This axiomatization is well typed with respect to the linear type system described in Section 3. Consequently, a mutable version of this ADT can be derived automatically, with operations encapsulated in a monad having types:

```

nilM      :: M Int → Int
consM     :: Int → M ()
nth-selectM :: Int → M Int
nth-updateM :: (Int, Int) → M ()

```

The following small program which manipulates this mutable ADT:

```

nilM
(consM 2      >>
 consM 1      >>
 nth-updateM (1, 5) >>
 nth-selectM 1      )

```

returns the value 5. Furthermore, using the graph-rewrite semantics of Section 5, *nth-updateM* will update the list *in-place* (in contrast with the original ADT, which must recreate the first *n* elements). In this example the benefits are only constant-factor amortized improvements in time and space, but in other examples the improvements are more dramatic (see Appendix A for the design of an array ADT, where a linear-factor improvement is realized). Even constant-factor improvements may be important in certain situations, for example if the improvement can reduce load on the garbage collector (we will show later that *nth-updateM* induces only a constant number of heap or stack allocations).

Finally, a new set of axioms for the ADT can be automatically obtained by applying the translation scheme in Section 4.3 to the original axioms above, yielding:

```

consM x >> nth-selectM 0 >> λz → m
  ⇒ consM x >> m[x/z]
consM x >> nth-selectM (i + 1) >> λz → m
  ⇒ nth-selectM i >> λz → consM x >> m
consM x >> nth-updateM (0, v)
  ⇒ consM v
consM x >> nth-updateM (i + 1, v)
  ⇒ nth-updateM (i, v) >> consM x
consM x >> unit y ⇒ unit y
nilM (unit y) ⇒ y

```

(\gg and \gg are the monadic *bind* and *seq* operators, respectively.) Note that these axioms allow reasoning about integer lists at the level of monads, without exposing the underlying representation.

2 Preliminaries

Define the language \mathcal{L} as the lambda calculus extended with constants, which include primitive datatypes and operations on them, such as the unit type $()$, integers, booleans, and pairs. Its syntax is given in Figure 1. In addition, for convenience we will often use a non-recursive *let* expression defined by:

$$\text{let } x = e_1 \text{ in } e_2 \equiv (\lambda x \rightarrow e_2) e_1$$

The only reduction rules that interest us at this stage are the conventional β rule:

$$(\lambda x \rightarrow e) y \Rightarrow_{\beta} e[y/x]$$

and the δ rules that govern the behavior of conditional expressions, primitive datatypes, and recursion. For the conditional and pairing, we have:

$$\begin{aligned}
\text{if } \text{True} \ e_1 \ e_2 &\Rightarrow_{\delta} e_1 \\
\text{if } \text{False} \ e_1 \ e_2 &\Rightarrow_{\delta} e_2 \\
\text{fst } (e_1, e_2) &\Rightarrow_{\delta} e_1 \\
\text{snd } (e_1, e_2) &\Rightarrow_{\delta} e_2
\end{aligned}$$

$$\begin{array}{l}
e ::= x \\
\quad | k \\
\quad | e_1 e_2 \\
\quad | (e_1, e_2) \\
\quad | \lambda x \rightarrow e
\end{array}
\quad
\begin{array}{l}
x \in \text{Var} \quad (\text{variables}) \\
k \in \text{Con} \quad (\text{primitive constants})
\end{array}$$

Figure 1: Syntax of \mathcal{L}

Note that the above rules imply a non-strict semantics.

For recursion, we assume $\text{fix} \in \text{Con}$ with δ rule:

$$\text{fix } f \Rightarrow_{\delta} f (\text{fix } f)$$

Define the relation \Rightarrow as the union of \Rightarrow_{β} and \Rightarrow_{δ} . We write $e_1 \Downarrow_{CBN} e_2$ to denote the reduction of e_1 to e_2 using \Rightarrow under a call-by-name (leftmost, outermost) reduction strategy. *Evaluation* of an expression e is defined as reduction of e to a *value* v , defined as:¹

$$v ::= k \mid (v_1, v_2)$$

Evaluation is a partial function.

2.1 Abstract Datatypes (ADTs)

Definition 1 Given a type of interest T , a *simple ADT* is one in which each operation can be classified as either a *generator* of type $X_1 \rightarrow T$, a *modifier* of type $X_2 \rightarrow T \rightarrow T$, or a *selector* of type $X_3 \rightarrow T \rightarrow X_4$, where the X_i are arbitrary auxiliary datatypes.

This is a standard classification for ADTs, and most conventional ADTs such as arrays, lists, stacks, and queues can be expressed in this way. However, it does not include *tree-shaped* ADT's, since a modifier for such an ADT would need a type such as $X \rightarrow T \rightarrow T \rightarrow T$. This limitation is discussed further in Section 6.

We allow \mathcal{L} to be extended with simple ADTs by extending the base types as necessary and using the ADT axioms as δ rules in the calculus. We assume that adding new axioms does not destroy confluence (which can also be achieved by choosing a determinate order on the delta rules, as is done in most modern functional languages). For an ADT \mathcal{A} we refer to the extended language as $\mathcal{L}^{\mathcal{A}}$.

Definition 2 Given a simple ADT with a set of generators G , modifiers M , and selectors S , an *axiomatization* distinguishes a set $D = S \cup U$, where $U \subseteq M$. Each operator $d \in D$ has one or more defining axioms whose syntax is given in Figure 2.

¹Evaluation can be generalized to include abstractions as values, but for simplicity we choose not to here.

This definition subdivides modifiers into two categories: U is the set of *mutators*, each having defining axioms; and the rest ($M \setminus U$) are called *constructors*, whose behavior is defined implicitly by the axioms. For example, for the integer list ADT in the introduction, *nil* is a generator, *cons* is a constructor, *nth-update* is a mutator, and *nth-select* is a selector.

Note that the syntax in Figure 2 restricts the axioms to be first-order.

3 The Linearity Condition

Not every ADT can be converted to monadic form; it must be “linear,” or “single-threaded,” in some sense. In this section we describe a *linear type system* which constrains the ADT sufficiently to allow the conversion. It is inspired by the linear type system with read-only access in [19], which in turn was influenced by the single-threadedness condition in [17].

A *type* in our system can be a basic type, a pair of auxiliary types, a function type, or the type of interest T . In addition, it could be a *linear* type of interest, denoted $!T$. Formally:²

$$\begin{array}{l}
u, v ::= \text{Int} \quad | \quad \text{Bool} \quad | \quad \dots \\
\quad | (u, v) \quad | \quad (u \rightarrow v) \\
\quad | T \quad | \quad !T
\end{array}$$

Facilitated with this finer-grained type system, we are able to type each class of ADT operators to better reflect the intended use of T :

$$\begin{array}{l}
g \quad :: \quad X_1 \rightarrow !T \\
m \quad :: \quad X_2 \rightarrow !T \rightarrow !T \\
s \quad :: \quad X_3 \rightarrow T \rightarrow X_4
\end{array}$$

Generators and modifiers manipulate $!T$ to ensure that at any time they either create or have the sole pointer to the type of interest. On the other hand, selectors take a non-linear T as an argument because they only read the type of interest and, as a result, the linearity condition can be relaxed.

²In this work, the linearity of the type of interest is all that we care about, so it would be a waste of ink if we follow the convention of linear logic and tag each *non-linear* type with $!$. Instead we tag the linear type of interest with a $!$ as is done in [19]. We pronounce $!T$ as “squirt T ” (gentler than a bang).

$\text{axiom} ::= \text{lhs} \Rightarrow \text{rhs}$ $\text{lhs} ::= m \ x \ \text{pat} \mid s \ x \ \text{pat}$ $\text{pat} ::= t \mid g \ x \mid c \ x \ \text{pat}$ $\text{rhs} ::= g \mid s \mid m \mid c \mid x \mid t \mid k$ $\quad \mid (rhs_1, rhs_2) \mid rhs_1 \ \text{rhs}_2$ $\quad \mid \text{if } rhs_1 \ \text{then } rhs_2 \ \text{else } rhs_3$	$g \in G \quad (\text{generators})$ $s \in S \quad (\text{selectors})$ $m \in U \quad (\text{mutators})$ $c \in M \setminus U \quad (\text{constructors})$ $x \in \text{Var} \quad (\text{identifiers of auxiliary type})$ $t \in \text{Var} \quad (\text{identifiers of type of interest})$ $k \in \text{Con} \quad (\text{constants})$
---	--

Figure 2: Axiom Syntax Scheme

An assumption list associates variables and constants with types:

$$A ::= x_1 : u_1, \dots, x_n : u_n$$

where x_1, \dots, x_n are distinct variables or constants. The *base assumption list* $B_{\mathcal{A}}$ for an ADT \mathcal{A} associates all the operators of \mathcal{A} , primitives, and constants with their types. We write A, B to denote the concatenation of two assumption lists; an implicit side condition of a concatenation is that A and B contain distinct variables.

$R(A)$ is an assumption list identical to A except that all the associations of mutators are elided, and those of generators, constructors, and variables of type $\text{!}T$ are “non-linearized.” That is, the ! ’s in their type signatures are removed. Intuitively, R prepares a “read-only” assumption list for an expression which may do multiple reads on the type of interest.

The typing rules are listed in Figure 3. The predicate $NL(u)$ (“non-linear u ”) checks that u does not have a linear type of interest as a component. It is defined by:

$$NL(u) \iff u \neq \text{!}T \quad \text{and} \\ u = (v, w) \implies NL(v) \text{ and } NL(w)$$

The predicate $HY(u)$ (“hygienic u ”) checks that u is not a function type and does not have *any* type of interest, either linear or nonlinear, as a component. It is defined by:

$$HY(u) \iff u \neq v \rightarrow w \text{ and} \\ u \neq \text{!}T \quad \text{and} \\ u \neq T \quad \text{and} \\ u = (v, w) \implies HY(v) \text{ and } HY(w)$$

Note that our system allows (Weakening) on any variable because the “no discarding” property of a linear type of interest is not of our concern. This design in turn enables the use of only one (Identity) rule. On the other hand, (Contraction) can only be applied to a variable of non-linear type because the “no duplicate” property of a linear type of interest is the theme of the whole type system. There are three application rules: ($\text{!}T$ -Application) is identical to the application

rules in other linear type system except that here we constrain the resulting type of the application to be $\text{!}T$; (T -Application) allows read-only access in the argument term whose type is T , and the hygienic condition ensures that no type of interest can be smuggled out (a side-effect of this constraint is that the resulting type can not be a function type); (Application) handles applications whose operands are of hygienic type. Also note that read-only access is used when typing the predicate of a conditional expression.

Definition 3 An ADT \mathcal{A} is *linear* if each axiom $\text{lhs} \Rightarrow \text{rhs}$ has the following properties:

- There exists a type u and assumption list A such that:
 - $B_{\mathcal{A}}, A \vdash \text{lhs} : u$, and
 - If $u = \text{!}T$, then $B_{\mathcal{A}}, A \vdash \text{rhs} : u$; otherwise, $R(B_{\mathcal{A}}, A) \vdash \text{rhs} : u$.
- Each free variable has exactly one occurrence in lhs .

As an example of a well-typed linear ADT, consider the integer list ADT given in the introduction. As an example of an ADT that would not be well-typed, consider adding the following (contrived) operator to the integer list ADT:

$$\text{silly } l \Rightarrow \text{let } l' = \text{nth-update } (1, 2) \ l \\ \text{in } \text{nth-select } 1 \ l'$$

This is not linear because *silly* is a selector, but there is an update to the type of interest on the right-hand side, which our type system does not allow.

4 Monadic ADTs

To say that an ADT is linear is to say something about its axioms. But if we naively implement a linear ADT in a functional language based on a Hindley-Milner type system, the linear axioms won’t buy us much: we still will not be able to do in-place updates on the type of

$$\begin{array}{c}
\frac{}{x : u \vdash x : u} \text{ (Identity)} \quad \frac{A, y : u, z : u, B \vdash e : v \quad NL(u)}{A, x : u, B \vdash e[x/y, x/z] : v} \text{ (Contraction)} \\
\\
\frac{A \vdash e : v}{A, x : u \vdash e : v} \text{ (Weakening)} \quad \frac{A, x : u, y : v, B \vdash e : t}{A, y : v, x : u, B \vdash e : t} \text{ (Exchange)} \\
\\
\frac{R(A) \vdash p : Bool \quad A \vdash e_1 : u \quad A \vdash e_2 : u}{A \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 : u} \text{ (If)} \\
\\
\frac{R(A) \vdash e_1 : u \quad R(A) \vdash e_2 : v \quad HY(u) \quad HY(v)}{A \vdash (e_1, e_2) : (u, v)} \text{ (Pair)} \\
\\
\frac{A \vdash e_1 : ;T \rightarrow ;T \quad B \vdash e : ;T}{A, B \vdash e_1 e : ;T} \text{ (;T-Application)} \\
\\
\frac{A \vdash e_1 : T \rightarrow v \quad R(B) \vdash e_2 : T \quad HY(v)}{A, B \vdash e_1 e_2 : v} \text{ (T-Application)} \\
\\
\frac{A \vdash e_1 : u \rightarrow v \quad B \vdash e_2 : u \quad HY(u)}{A, B \vdash e_1 e_2 : v} \text{ (Application)}
\end{array}$$

Figure 3: Linear Typing Rules

interest because the type system is not strong enough to guarantee that the argument to an ADT mutator is unshared. For example, the program:

```

let l1 = cons 1 (cons 2 nil)
    l2 = nth-update (1, 5) l1
in nth-select 1 l2 + nth-select 1 l1

```

should return 7. But if the update were done destructively, the result might be 10, which is clearly wrong.

We could, of course, enrich the type system with linear types throughout the language, in which case the above program would be ill-typed (at least one functional language, Clean, uses this approach [2]). We choose instead a different solution: we will encapsulate the linear ADT in a monad, and use the monadic structure to enforce linearity of the type of interest throughout the language.

4.1 Monad Basics

Definition 4 A *monad* is a triple (M, \gg, unit) where M is a type constructor, and \gg and unit are functions which satisfy the following three laws:

$$\begin{aligned}
\text{unit } a \gg \lambda b \rightarrow n &= n[a/b] \\
n \gg \text{unit} &= n \\
n_1 \gg (\lambda a \rightarrow n_2 \gg \lambda b \rightarrow n_3) &= (n_1 \gg \lambda a \rightarrow n_2) \\
&\gg \lambda b \rightarrow n_3
\end{aligned}$$

In the last law, a does not appear free in n_3 .

Operationally speaking, $M a$ is a *computation* which produces an answer of type a when executed; \gg combines two computations into a larger one which performs the two in sequence; and unit injects a single value into a computation. Of particular interest to us is the *state monad* which abstracts over a type of interest to be treated as mutable state:

$$\begin{aligned}
\mathbf{type} \ M a &= T \rightarrow (a, T) \\
\gg &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \\
\text{unit} &:: a \rightarrow M a \\
(n \gg k) t &\Rightarrow \text{let } p = n t \text{ in } k (fst p) (snd p) \\
(\text{unit } a) t &\Rightarrow (a, t)
\end{aligned}$$

It can be easily verified that these definitions satisfy the three monad laws.

For convenience, we specialize a version of \gg which we write \ggg :

$$(n_1 \ggg n_2) t \Rightarrow n_2 (snd (m_1 t))$$

\ggg and \gg are like Haskell's \ggg and \gg , respectively, and are sometimes called *bind* and *seq*.

4.2 Monadic Encapsulation of ADT's

For each modifier $m :: X_2 \rightarrow T \rightarrow T$ and selector $s :: X_3 \rightarrow T \rightarrow X_4$ in an ADT \mathcal{A} , we introduce monadic counterparts mM and sM , respectively, defined in terms of m and s :

$$\begin{aligned} mM &:: X_2 \rightarrow M () \\ sM &:: X_3 \rightarrow M X_4 \\ mM \ x \ t &\Rightarrow ((), \ m \ x \ t) \\ sM \ x \ t &\Rightarrow (s \ x \ t, \ t) \end{aligned}$$

The typing $M ()$ which results from applying mM implies that the modifier only effects the type of interest and does not return anything interesting.

In addition, for each generator $g :: X_1 \rightarrow T$ in an ADT \mathcal{A} we define a monadic generator gM by:

$$\begin{aligned} gM &:: X_1 \rightarrow M a \rightarrow a \\ gM \ x \ m &\Rightarrow \text{fst} (m (g \ x)) \end{aligned}$$

This definition reflects the use of gM as the operator which *invokes* a computation: it applies the monadic computation to the type of interest generated by the original ADT generator and returns the first element of the resulting pair of the computation while discarding the second element, the (possibly altered) type of interest.

4.3 Deriving Monadic Axioms

The encapsulation just described leads to a monadic implementation of a linear ADT, and we will see in a later section that it allows in-place update of the type of interest. But it is not an especially good mechanism for reasoning about a monadic ADT, since it reveals the details of the encapsulation, and, once the original ADT is exposed, reverts to reasoning in terms of that ADT. It would be better if we had monadic axioms, corresponding to the original linear ones, that did not expose the underlying ADT representation. In this section we describe a method for deriving such axioms from those in the original linear ADT.

Our translation scheme resembles, perhaps not surprisingly, a CPS conversion, and is related to call-by-value monad transformation [13, 18]. But it is also different from these in several ways, the two most important being: (1) the conversion is performed on *axioms* rather than just terms, and (2) the conversion is done with respect to a particular value—the *type of interest*—and in fact is not even valid for non-linear ADTs.

In the following, we use $[\dots]$ and $\langle \dots \rangle$ to quote and un-quote, respectively, the text being translated. First we define a translation function \mathcal{D} for expressions that, given an expression e and a continuation k , returns an

equivalent expression using the monadic ADT:

$$\begin{aligned} \mathcal{D} [s \langle x \rangle \langle t \rangle] k &= \mathcal{D} t (\mathcal{D} x [\lambda y \rightarrow sM y \gg \langle k \rangle]) \\ \mathcal{D} [m \langle x \rangle \langle t \rangle] k &= \mathcal{D} t (\mathcal{D} x [\lambda y \rightarrow mM y \gg \langle k \rangle]) \\ \mathcal{D} [g \langle x \rangle] k &= [gM \langle x \rangle \langle k \rangle] \\ \mathcal{D} [if \langle p \rangle then \langle c \rangle else \langle a \rangle] k &= \mathcal{D} p [\lambda x \rightarrow if \ x \ then \langle \mathcal{D} c \ k \rangle \ else \langle \mathcal{D} a \ k \rangle] \\ \mathcal{D} [\langle (e_1), \langle e_2 \rangle \rangle] k &= \mathcal{D} e_1 [\lambda y \rightarrow \langle \mathcal{D} e_2 [\lambda z \rightarrow (unit (y, z)) \gg \langle k \rangle] \rangle] \\ \mathcal{D} [f \langle e_1 \rangle \langle e_2 \rangle] k &= \mathcal{D} e_1 [\lambda y \rightarrow \langle \mathcal{D} e_2 [\lambda z \rightarrow (unit (f \ y \ z)) \gg \langle k \rangle] \rangle] \\ \mathcal{D} [t] k &= k \\ \mathcal{D} [x] (\lambda y \rightarrow n) &= n[x/y] \\ \mathcal{D} [c] (\lambda y \rightarrow n) &= n[c/y] \end{aligned}$$

Here s , m , and g represent an arbitrary selector, modifier, and generator, respectively; sM , mM , and gM are their monadic counterparts; t is an identifier whose type is the type of interest; x is an identifier having auxiliary type; f is a primitive function; and c is a constant having auxiliary type. We assume suitable α -renaming to avoid name clashes with y , z , and k .

Given \mathcal{D} , we can now define translations \mathcal{S} and M for *axioms* that define selectors and mutators, respectively.

$$\begin{aligned} \mathcal{S} [s \langle x_1 \rangle \langle t \rangle \Rightarrow \langle x_2 \rangle] &= [\langle \mathcal{D} [s \langle x_1 \rangle \langle t \rangle] [(\lambda y \rightarrow n)] \rangle \\ &\quad \Rightarrow \langle \mathcal{D} x_2 [\lambda y \rightarrow \langle \mathcal{D} t [n] \rangle] \rangle] \\ M [m \langle x \rangle \langle t_1 \rangle \Rightarrow \langle t_2 \rangle] &= [\langle \mathcal{D} [m \langle x \rangle \langle t_1 \rangle] [k] \rangle \Rightarrow \langle \mathcal{D} t_2 [k] \rangle] \end{aligned}$$

Here also we assume suitable α -renaming to avoid name clashes with y , z , and k . Note that \mathcal{S} translates the right-hand side in the context of the nested type of interest on the left-hand side, whereas M translates both sides independently. Also note that if an axiom is defined in terms of type of interest identifier t , then the type of interest *disappears* during the translation (thus completing our mission of hiding the state!).

In addition to the axioms derived from the original ADT axioms, we also need to axiomatize:

- the interactions between generators and *unit*. For each generator gM , add the rule:

$$gM (unit \ y) \Rightarrow y$$

- the interactions between constructors and \gg . For each constructor cM , add the rule:

$$cM \ x \ \gg \ unit \ y \Rightarrow unit \ y$$

An example of the overall translation is the monadic axioms for the integer list ADT given in the introduction. It is interesting to note in this translation process

that mutators and selectors trade roles! This is an artifact of the CPS-like nature of the translation, which uncovers a “duality” between constructors and selectors, much like the duality noted by Filinski in [3].

Calling the above translation scheme \mathcal{T} , its correctness is captured by its relationship to the monadic encapsulation previously defined:

Theorem 1 *The monadic encapsulation of a linear ADT \mathcal{A} satisfies the monadic axiomatization derived from \mathcal{A} using translation \mathcal{T} .*

Proof: By applying the translation scheme to a generic axiom of each kind, then using equational reasoning to unfold the monadic encapsulation on each side and reduce them to equivalent terms. A proof for one of the three axioms is given in Appendix B.

5 GRS Operational Semantics

In [7] we informally defined:

Definition 5 A *mutable* ADT, or MADT, is any ADT whose operational semantics permits *in-place update* (i.e. *destructive reuse*) of the type of interest, while still retaining confluence.

We will now make this definition more precise. To do so, we need an operational semantics that captures sharing and identifies all relevant costs of execution. The evaluation function \Downarrow_{CBN} is not suitable for this purpose since it is essentially a *term*-rewriting system. Instead, we choose a *graph*-rewriting system (GRS) to capture operational semantics. Although other approaches to operational semantics abound—such as recent efforts at modeling lazy evaluation [10, 1, 14]—we find a GRS as particularly clear (a picture is worth a thousand words), abstract (no heap, for example), expressive (capturing notions of sharing, updating, and sequencing), and intuitive (resembling conventional graph reduction).

Figure 4 defines a function *lift* which maps terms of \mathcal{L} into their corresponding graphs (called *\mathcal{L} -graphs*). For each CBN δ rule, a corresponding GRS rule can be obtained by lifting both sides of the rule. For example, Figure 5 shows the GRS rules for *fst* and *snd*. The GRS rule corresponding to the CBN β rule is also shown in Figure 5, where the notation $e\{y/x\}$ denotes the result of redirecting the edges which point to the bound variable x in the subgraph e to the subgraph y ; thus y is shared (also, as in [21], we assume that a fresh copy of the function body e is created before the substitution is made).

We write $e_1 \Downarrow_{GRS} e_2$ to denote the reduction of e_1 to e_2 using the GRS-lifted reduction \Rightarrow under a call-by-name (leftmost, outermost) reduction strategy. The

evaluation function \Downarrow_{GRS} can be seen as the *call-by-need* equivalent of the call-by-name evaluator \Downarrow_{CBN} .

Theorem 2 *For every term e in \mathcal{L} ,*

$$e \Downarrow_{CBN} v \iff \text{lift}(e) \Downarrow_{GRS} v$$

where v is a value.

Proof: See [21].

5.1 Lifting of ADT’s

The lifting operation can also be extended to an ADT by lifting each side of each axiom. For example, the GRS rules for the two *nth-update* axioms given in the introduction are shown in Figure 6.

When implementing GRS rules—for example using conventional graph reduction techniques—all of the reductions are assumed to be “pure.” That is, other than the top vertex on each of the left- and right-hand sides, no vertices are assumed to be shared. For example, in the first *nth-update* rule of Figure 6, the top vertex on the left is assumed to be the same as the top vertex on the right, but the vertices that comprise the *cons* operation on the right are assumed to be “fresh” vertices, distinct from those on the left. This is necessary, of course, to preserve confluence, as we have discussed earlier. But this is also exactly where we would like to do better: if some vertices on the left are discarded as a result of applying the rule, there is no reason why they can’t be immediately re-used on the right, thus achieving “in-place update.” In those cases where we abandon the “pure” reduction strategy in a rule, we will annotate the vertices with names to indicate which are being reused, and how.

5.2 GRS Rules for Encapsulated ADT

The monad laws can also be lifted into the GRS, as shown in Figure 7, where t denotes the subgraphs representing state. Note in the rule for *unit* that the state t is single-threaded; i.e. t is reachable from the root of each side of the rule in exactly one way. On the other hand, the rule for $\underline{\geq}$ is *not* single-threaded, at least not in this simplistic way: in reality it depends on the behavior of k and n . This observation hints that if:

- a linear ADT operator is only used “monadically,” i.e. having type $M a$ and only combined with other monadic computations via $\underline{\geq}$, *and*
- a fresh (unshared) copy of the type of interest is passed into a monadic computation when it is invoked,

$$\begin{aligned}
\text{lift}(x) &= x & \text{lift}(k) &= k & \text{lift}(\lambda x \rightarrow e) &= \begin{array}{c} \lambda x \\ | \\ \text{lift}(e) \end{array} \\
\text{lift}(e1\ e2) &= \begin{array}{c} \circ \\ / \quad \backslash \\ \text{lift}(e1) \quad \text{lift}(e2) \end{array} & \text{lift}((e1,e2)) &= \begin{array}{c} \circ \\ / \quad \backslash \\ \text{Pair} \quad \text{lift}(e2) \\ | \\ \text{Pair} \quad \text{lift}(e1) \end{array}
\end{aligned}$$

Figure 4: Lifting Terms to Graphs

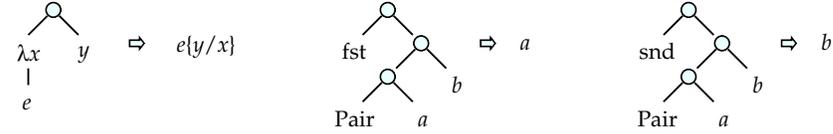


Figure 5: GRS Rules for Application and Pair Selections

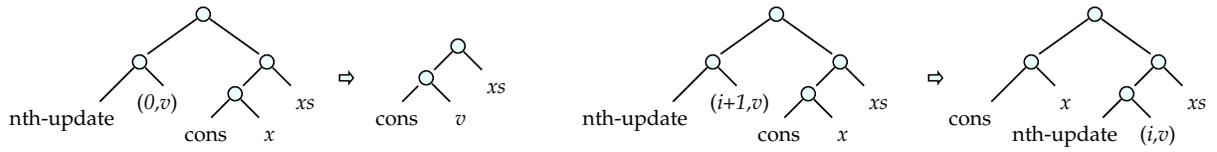


Figure 6: GRS Rules for *nth-update*

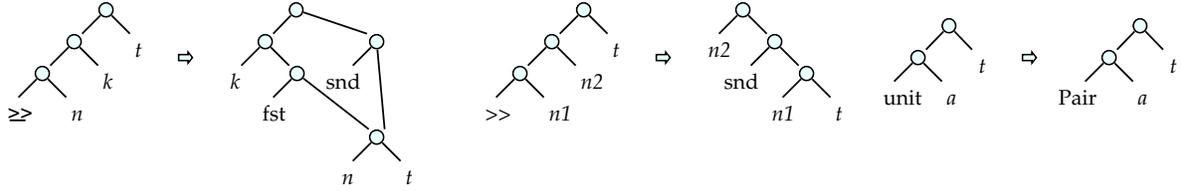


Figure 7: GRS Rules for Monadic Combinators

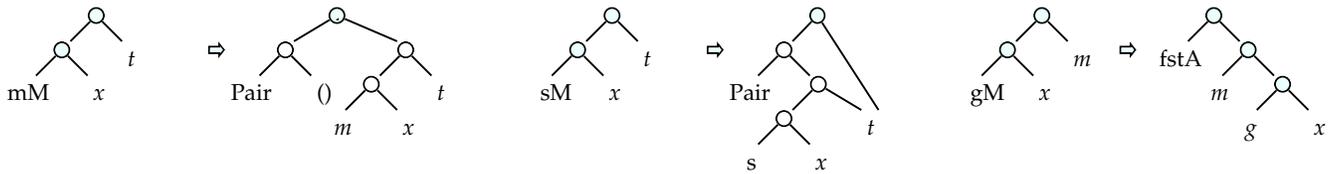


Figure 8: GRS Rules for Monadic ADT Operators

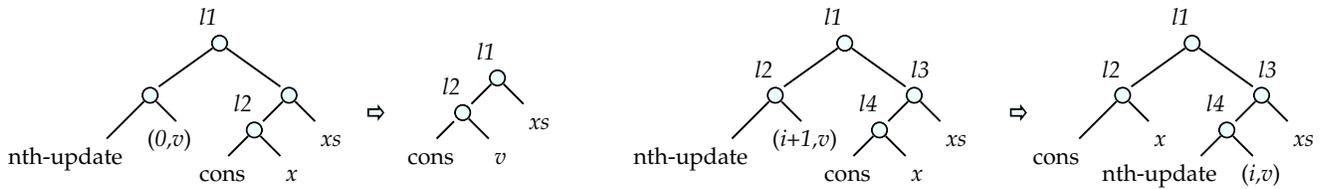


Figure 9: GRS Rules for *nth-update* Reusing Vertices

then the single-threadedness of the type of interest can be guaranteed throughout the entire computation.

We can achieve this effect by first lifting the encapsulation strategy of Section 4.2 into the GRS, as shown in Figure 8. However, the GRS rule for mM will have one important twist: the mutator m that it exposes will be allowed to reuse the type of interest; i.e. in-place update is allowed. To see this concretely, Figure 9 shows the rules for $nth\text{-update}$, using the convention of vertex reuse described earlier. It is interesting to note that the second rule implies that no vertices are needed to realize the recursive call structure of $nth\text{-update}$; that is, it is fully tail-recursive, or iterative. Thus, in our operational semantics, an application of $nth\text{-update}M$ requires allocation of exactly two vertices, regardless of the length of the list.

There is one problem, however: the right-hand side of the rule for sM introduces sharing in t , but using CBN reduction there is no guarantee that the lifted graph of $(s\ x\ t)$ on the right-hand side is reduced before some subsequent mutation to t . We can fix this problem by making fst and snd hyper-strict in the first component of their arguments. This is the only sensible way to ensure that there are no dangling references to the type of interest, and is similar to the use of a hyper-strict *let* expression in [19]. Thus we define new versions of the pair selectors, $fstA$ and $sndA$:

$$\begin{aligned}fstA\ (v, e) &\Rightarrow_{\delta} v \\sndA\ (v, e) &\Rightarrow_{\delta} e\end{aligned}$$

where v is a value. We then use these to redefine \gg and \ggg :

$$\begin{aligned}(m \gg k)\ t &\Rightarrow \text{let } p = m\ t \text{ in } k\ (fstA\ p)\ (sndA\ p) \\(m \ggg n)\ t &\Rightarrow n\ (sndA\ (m\ t))\end{aligned}$$

The GRS versions of these rules are exactly the same as those in Figure 7 except that fst is replaced by $fstA$ and snd by $sndA$.

We can see that using hyper-strict pair selectors restores single-threadedness by reasoning as follows:

- If $fstA\ (sM\ x\ t)$ is reduced before $sndA\ (sM\ x\ t)$, then t becomes unshared because $fstA$ forces $(s\ x\ t)$ to be fully evaluated.
- Similarly, if $sndA\ (sM\ x\ t)$ is reduced first, $sndA$ likewise forces $(s\ x\ t)$ to be fully evaluated, so t becomes unshared before it is further used.

The GRS system was necessarily made more strict in order to achieve single-threadedness. Thus it is possible that some computations which do not contribute to the evaluation of the final answer are evaluated (and may

not terminate). For example:

$$\begin{aligned}gM\ c_1 \\(sM\ c_2 \gg \lambda x \rightarrow \\sM\ c_3 \gg \lambda y \rightarrow \\unit\ y\end{aligned}$$

where x does not appear in c_3 . The GRS would evaluate the subgraph representing $(s\ c_2\ (g\ c_1))$, whose value is obviously irrelevant to the final answer. Nevertheless, GRS still offers the beauty of lazy evaluation in many cases. For example:

$$\begin{aligned}gM\ c_1 \\(mM\ c_2 \gg \lambda x \rightarrow \\mM\ c_3 \gg \lambda y \rightarrow \\unit\ 1\end{aligned}$$

Here the GRS returns 1 without creating even the initial copy of the type of interest, let alone performing the two mutations.

5.3 Correctness of In-place Update

Using our somewhat stricter notion of GRS evaluation, we can prove a correctness result for in-place update. Define \mathcal{L}^M as the language \mathcal{L} extended with any monadically encapsulated, linear ADT \mathcal{M} .

Theorem 3 *For every term e in \mathcal{L}^M ,*

$$e \Downarrow_{\text{CBN}} v \iff \text{lift}(e) \Downarrow_{\text{GRS}} v$$

where v is a value.

Proof: See Appendix C.

6 Discussion

In this section we discuss limitations and possible extensions of our methodology.

Although we believe that the added strictness introduced in Section 5.2 is minimal, and that some added strictness is inherent in any single-threaded reduction system, we are investigating ways to improve this, such as dynamically tagging closures having no references to the type of interest. But it is not clear that such a small improvement is worthwhile.

Polymorphism should be easy to handle, at the expense of carrying around more type information. For example, the integer list ADT could be turned into a polymorphic list, but the monad type would have to be extended to something like $M\ a\ b$, where a is the type returned from the computation, and b is the type of the list elements.

The linear type system proposed in Section 3 is limited to first-order values, so that, for example, a selector cannot return a function as its result. We believe, however, that the type system can be extended to handle abstractions, and we have worked out the preliminary ideas of such an extension.

A more serious limitation of the current system is that it does not handle more than one mutable ADT. One possible solution to this is to introduce *references*, so that individual objects can be named and manipulated, and use the extension to the Hindley-Milner type system proposed in [11] to prevent the escape of references from the monadic computation.

Related to this issue is the inability to handle tree-like structures, ruled out by our definition of a simple ADT. Rather than using references, we have been working on a different solution to this problem, based on the following idea.

Suppose we are designing a database-like ADT, whose axiomatization is given by:

$$\begin{aligned}
new &:: DB \\
insert &:: Int \rightarrow DB \rightarrow DB \\
remove &:: Int \rightarrow DB \rightarrow DB \\
\\
remove\ a\ new &= new \\
remove\ a\ (insert\ b\ x) &= \text{if } a = b \text{ then } x \\
&\quad \text{else } insert\ b\ (remove\ a\ x)
\end{aligned}$$

This is clearly linear, so our methodology can be applied, yielding a mutable ADT with monadic axioms, etc. However, it is not very efficient. So, as in a traditional ADT design, we may wish to implement the ADT using binary search trees. The implementation details should be hidden, of course, and we do so below using Haskell module syntax:

```

module DB (DB, new, insert, find, remove) where
data DB = Leaf | Branch Int Tree Tree
new      = Leaf
insert a Leaf      = Branch a Leaf Leaf
insert a (Branch b lt rt) =
    if a<=b then Branch b (insert a lt) rt
    else Branch b lt (insert a rt)
remove a Leaf      = Leaf
remove a (Branch b lt rt) =
    if a==b
    then if lt==Leaf then rt else
         if rt==Leaf then lt else
         let pred = largest lt
             lt'  = remove pred lt
         in Branch pred lt' rt
    else if a<=b then Branch b (remove a lt) rt
         else Branch b lt (remove a rt)
where largest (Branch x l r) =
    if r==Leaf then x else largest r

```

The interesting thing about this solution is that the tree is used linearly in the implementation viewed as

a set of axioms, only our type system is currently not strong enough to determine this. If it could be suitably strengthened, then this encapsulation of the axioms should allow in-place update of the tree.

7 Conclusions

Using state monads to achieve safe destructive reuse of the state is by no means an innovation; several earlier efforts have already demonstrated its suitability. What have been missing in these previous efforts are a systematic way to recognize when monads work and when they don't, an operational semantics to formally reason about the correctness of in-place updates, a method for designing efficient state monads, and a way to reason about them abstractly. Our contributions take positive steps toward resolving these deficiencies.

8 Acknowledgements

Thanks to Phil Wadler and the anonymous referees for their helpful comments on previous versions of this paper. This research was supported in part by NSF under Grant CCR-9404786.

References

- [1] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 233–241, New York, January 1995. ACM Press.
- [2] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In *Proc. 13th Conference on the Foundations of Software Technology & Theoretical Computer Science*, Bombay, India, 1993.
- [3] 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).
- [4] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proc. Symposium on Logic in Computer Science*, pages 333–343. IEEE, June 1990.
- [5] J.C. Guzman. *On Expressing the Mutation of State in a Functional Programming Language*. PhD thesis, Yale University, 1993.

- [6] S. Holmstrom. A linear functional language. In *Proc. the Workshop on the Implementation of Lazy Functional Languages, PMG Report 53*, pages 13–32, 1988.
- [7] P. Hudak. Mutable abstract datatypes – or – how to have your state and munge it too. Research Report YALEU/DCS/RR-914, Yale University, Department of Computer Science, December 1992.
- [8] 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.
- [9] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
- [10] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 144–154, New York, January 1993. ACM Press.
- [11] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Proc. Symposium on Programming Language Design and Implementation '94*, Orlando, June 1994. ACM.
- [12] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *European Symposium on Programming '96*, Linkoping, Sweden, April 1996.
- [13] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*. IEEE, June 1989.
- [14] G. Morriset, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, New York, June 1995. ACM/IFIP, ACM Press.
- [15] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda-calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.
- [16] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th Symposium on Principles of Programming Languages*. ACM, January 1993. (to appear).
- [17] D. A. Schmidt. Detecting global variables in denotational specification. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.
- [18] P. Wadler. Comprehending monads. In *Proc. Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.
- [19] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, Sea of Galilee, Isreal, April 1990. IFIP.
- [20] P. Wadler. The essence of functional programming. In *Proc. 19th Symposium on Principles of Programming Languages*, pages 1–14. ACM, January 1992.
- [21] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

A A Complete Example: Array ADT

Consider a very simple, fixed-size integer array ADT, with operations:

$$\begin{aligned} \text{newArr} &:: (\text{Int}, \dots, \text{Int}) \rightarrow \text{Array} \\ \text{update} &:: (\text{Ix}, \text{Int}) \rightarrow \text{Array} \rightarrow \text{Array} \\ \text{select} &:: \text{Ix} \rightarrow \text{Array} \rightarrow \text{Int} \end{aligned}$$

whose *array axiomatization* is given by:

$$\begin{aligned} \text{select } i \ (\text{newArr } (x_1, \dots, x_i, \dots, x_n)) &\Rightarrow x_i \\ \text{update } (i, y) \ (\text{newArr } (x_1, \dots, x_i, \dots, x_n)) &\Rightarrow \text{newArr } (x_1, \dots, y, \dots, x_n) \end{aligned}$$

Intuitively, $\text{newArr } xs$, where xs is an n -tuple, creates an array of size n , each of whose elements is initialized to its corresponding component in xs ; $\text{update } (i, v) a$ returns an array identical to a except that the value of the i th element is v ; and $\text{select } i a$ returns the value of the i th element of a .

It can be verified that these axioms satisfy our linearity condition. The GRS rules for this ADT are given in Figure 10. Monadically encapsulating newArr , update , and select , we get:

$$\begin{aligned} \text{type } M a &= \text{Array} \rightarrow (a, \text{Array}) \\ \text{newArrM} &:: (\text{Int}, \dots, \text{Int}) \rightarrow M a \rightarrow a \\ \text{updateM} &:: (\text{Ix}, \text{Int}) \rightarrow M () \\ \text{selectM} &:: \text{Ix} \rightarrow M \text{Int} \end{aligned}$$

Their GRS rules are shown in Figure 11; and a programming example in this MADT is given in Figure 12. The correctness of destructively reusing the array follows from Theorem 3.

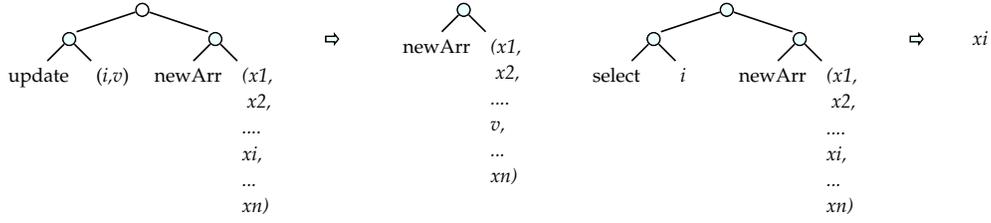


Figure 10: GRS Rules for Array ADT

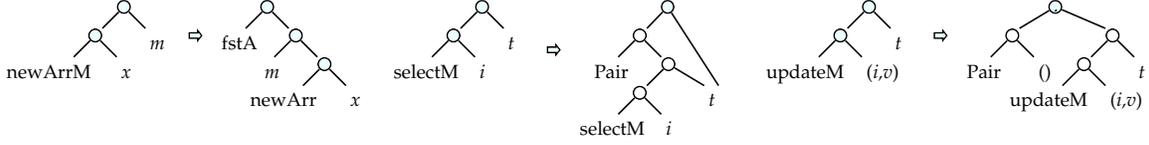


Figure 11: GRS Rules for Monadic Array MADT

Finally, we apply the axiom translation scheme to obtain the following monadic MADT axiomatization:

$$\begin{aligned}
& \text{newArrM } (\dots x_i \dots) (\text{selectM } i \ggg g) \\
& \Rightarrow \text{newArrM } (\dots x_i \dots) (g \ x_i) \\
& \text{newArrM } (\dots x_i \dots) (\text{updateM } (i, y) \ggg d) \\
& \Rightarrow \text{newArrM } (\dots y \dots) d \\
& \text{newArrM } (\dots x_i \dots) (\text{unit } b) \Rightarrow b
\end{aligned}$$

B The Proof of Theorem 1

The proof is a tedious proof-by-cases on the formation of both sides of a linear axiom. In some cases, an induction argument is necessary. We illustrate the main idea by going through one sample case. Consider a selector axiom $s \ x_1 \ t = x_2$. To omit the uninteresting details, let us assume x_1 and x_2 are two identifiers of proper auxiliary types. By inspecting the typing rules in Figure 3 and the definition of simple ADTs, we know that t can either be an identifier of type of interest, an application of some constructor (this case is proved inductively), or an application of some generator. If it is the last case, the axiom looks like

$$s \ x_1 \ (g \ x_3) \Rightarrow x_2.$$

Applying \mathcal{S} to this axiom, we get

$$gM \ x_2 \ (sM \ x_1 \ \ggg \ \lambda y \rightarrow n) \Rightarrow gM \ x_2 \ n[x_2/y].$$

Unfolding gM , sM , and \ggg on the left-hand side of the axiom:

$$\begin{aligned}
& gM \ x_3 \ (sM \ x_1 \ \ggg \ \lambda y \rightarrow n) \\
& \Rightarrow_{gM} \text{fst} \ (sM \ x_1 \ \ggg \ \lambda y \rightarrow n) \ (g \ x_3) \\
& \Rightarrow_{\ggg} \text{fst} \ ((\lambda p \rightarrow (\lambda y \rightarrow n) \ (\text{fst} \ (sM \ x_1 \ p))) \\
& \quad (\text{snd} \ (sM \ x_1 \ p))) \ (g \ x_3) \\
& \Rightarrow_{\beta} \text{fst} \ ((\lambda y \rightarrow n) \ (\text{fst} \ (sM \ x_1 \ (g \ x_3)))) \\
& \quad (\text{snd} \ (sM \ x_1 \ (g \ x_3))) \\
& \Rightarrow_{\beta} \text{fst} \ (n[\text{fst} \ (sM \ x_1 \ (g \ x_3))/y] \\
& \quad (\text{snd} \ (sM \ x_1 \ (g \ x_3)))) \\
& \Rightarrow_{sM} \text{fst} \ (n[\text{fst} \ (s \ x_1 \ (g \ x_3), \ (g \ x_3))/y] \\
& \quad (\text{snd} \ (sM \ x_1 \ (g \ x_3)))) \\
& \Rightarrow_s \text{fst} \ (n[x_2/y] \ (\text{snd} \ (sM \ x_1 \ (g \ x_3)))) \\
& \Rightarrow_{sM} \text{fst} \ (n[x_2/y] \ (\text{snd} \ (s \ x_1 \ (g \ x_3), \ (g \ x_3)))) \\
& \Rightarrow_{sM} \text{fst} \ (n[x_2/y] \ (g \ x_3))
\end{aligned}$$

Unfolding gM on the right-hand side:

$$\begin{aligned}
& gM \ x_2 \ n[x_2/y] \\
& \Rightarrow_{gM} \text{fst} \ (n[x_2/y] \ (g \ x_3))
\end{aligned}$$

This term is identical to the unfolding of the left-hand side and we are done. \square

C The Proof of Theorem 3

Following the development in [5], it can be shown that for any term e in $\mathcal{L}^{\mathcal{M}}$:

$$e \Downarrow_{CBN} v \iff \text{lift}(e) \Downarrow_{GRS} v$$

where the monadic mutators in GRS do updates non-destructively.

So if we can prove that it is always safe to do in-place updates in GRS as an “optimization,” it follows that for a $\mathcal{L}^{\mathcal{M}}$ term e , its CBN evaluation yields the same result as its in-place update GRS evaluation.

$newArrM (0, \dots, 0)$		--	allocate new array and invoke a computation
$(updateM (1, 1)$	\gg		-- set 1st element to 1
$selectM 1$	\gg	$\lambda x \rightarrow$	-- read first element and bind it to x
$selectM 1$	\gg	$\lambda y \rightarrow$	-- read first element and bind it to y
$unit (x + y)$	$)$		-- return the sum of x+y (the value 2)

Figure 12: A Simple Program Using Array MADT

In order to prove that in-place update on a type of interest is safe, it is sufficient to show that a mutator can only update an *unshared* type of interest. More precisely, we want to prove that for a monadic term:

$$gM e_1 (n_1 \gg n_2 \dots n_i \dots \gg n_k)$$

the type of interest after reduction of $sndA (n t)$ at any step i , where n is the composition of computations n_1 through n_i , is unshared.

Base case: $i = 0$: the type of interest is supplied by the rule for gM , which by definition is always unshared.

Induction hypothesis: the type of interest after reduction of $sndA (n t)$ at step i is unshared.

Induction step: prove that the type of interest is unshared after reduction of $sndA (n t)$ at step $i + 1$. We proceed by case analysis of the operation at step $i + 1$:

- *unit*: The rule for *unit* shows that it simply passes on the type of interest without adding any pointers. So by the induction hypothesis, it is still unshared.
- *mM*: The rule for *mM* shows that it applies m to the unshared type of interest and passes the result on. But our linearity condition guarantees that m handles the type of interest single-threadedly. So the type of interest is still unshared.
- *sM*: The rule for *sM* adds a pointer to t in the first component of the pair; i.e. the reference to t in $s x t$. But the reduction rule for $sndA$ reduces this value to normal form, and since it is also linear, it cannot have any dangling reference to t . *sM* also simply passes the type of interest on, but this is now the only reference, so the type of interest is still unshared.

Thus by induction, the type of interest is always unshared just after the reduction of $sndA (n t)$. It follows that *mM* is always handed an unshared copy of the type of interest, and *mM* can safely perform an in-place update. This further implies that:

$$e \Downarrow_{CBN} v \iff lift(e) \Downarrow_{GRS} v$$

where v is a value and GRS employs rules doing in-place update on the type of interest. \square