

# Monadic Encapsulation in ML

Miley Semmelroth      Amr Sabry

Department of Computer and Information Science  
University of Oregon  
Eugene, OR 97403

{miley,sabry}@cs.uoregon.edu

## Abstract

In a programming language with procedures and assignments, it is often important to isolate uses of state to particular program fragments. The frameworks of type, region, and effect inference, and monadic state are technologies that have been used to state and enforce the property that an expression has no visible side-effects. This property has been exploited to justify the deallocation of memory regions despite the presence of dangling pointers.

Starting from an idea developed in the context of monadic state in Haskell, we develop an ML-like language with full assignments and an operator that enforces the encapsulation of effects. Using this language, we formalize and prove the folklore connection between effect masking and monadic encapsulation. Then, by employing a novel set of reductions to deal with dangling pointers, we establish the soundness of the type-based encapsulation with a proof based on a standard subject reduction argument.

## 1 Introduction

Two of the recurring themes in programming language research have been:

1. controlling the interaction of procedures and assignments, and
2. analyzing the lifetimes of heap-allocated objects.

Although their original introduction in the programming language world focused on neither topic [Mog89], *monads* nowadays address both problems.

**State in Haskell** In the programming language Haskell, monads separate the functional sublanguage (of values) from the imperative sublanguage (of computations) [Sab98, Wad90]. The state monad [LP95] comes equipped with four functions that map from values to computations: `returnST` coerces a value to a trivial computation that returns that value, `ref` maps a value to a computation that allocates a reference to that value, `deref` maps a value denoting a reference

to a computation that reads the contents of that reference, and `setref` maps two values to a computation that updates the reference denoted by the first with the value denoted by the second. An additional combinator `thenST` (or `letST`) is used to express the sequencing of computations. Finally `runST` is used to deliver a value from a computation by forcing the computation to be performed in an empty initial store.

By providing different types for values and computations, the separation of the functional language and the imperative language is *almost* neatly achieved by the standard Haskell type system [LP95]. The addition of `runST` requires a non-standard extension to the type system to ensure that `runST` actually *encapsulates* the computation represented by its subexpression. Intuitively, since this computation should be evaluated in an empty store, it cannot import references, and since this computation is supposed to return a value, it cannot export references.

In other words, the addition of `runST` and its associated typing rule extend the monadic framework to, not only separate the functional and imperative sublanguages, but also to reason about the lifetimes of references. All this is achieved in an elegant and simple framework that requires a modest extension to the type system, a minor change to the implementation of the type inference module, and no runtime cost. Unfortunately, a recent attempt [LS97] to formalize this “modest extension of the type system” and the intuitive operational meaning of `runST` as a construct that bounds the lifetimes of references allocated within its subexpression is now known to be incorrect.

The problem in the attempted proof (pointed out by Claus Reinke and Peter Thiemann in a private communication to John Launchbury and Amr Sabry) can be traced to the complicated semantics of lazy state threads [LP94]. Further attempts to fix the proof have repeatedly stumbled against these complications. Unable to adapt the proof technique to lazy state threads, we leave that problem as an open one, and instead investigate monadic encapsulation in the context of an ML-like call-by-value language with strict store primitives.

**Memory Encapsulation in ML** Besides being a first step towards formalizing the encapsulation of state in Haskell, the encapsulation of state in ML is of independent interest as witnessed by the large amount of previous, current, and planned work on the topic. Much of this work has been carried out in the context of effect and region inference [LG88, JG91, TJ92, TT97]. In contrast to the monadic ap-

proach, effect systems do not restrict the interactions between the functional and imperative sublanguages at all. Expressions with global effects are allowed but are given different type descriptions than expressions with no effects. The type system composes effects generated by imperative operations to annotate enclosing expressions with their effects. A special rule, called *effect masking*, may be used at any time to restrict the set of visible effects by eliminating local (*i.e.*, encapsulated) ones.

In some systems [LG88, JG91, TJ92], effect masking is purely an artifact of the type system and has no operational significance. For example, the absence of visible effects provides a simple criterion for generalizing type variables that avoids the subtle problems that occur when mixing polymorphism and assignments [Tof90, Wri95]. The encapsulation of memory *regions* can also be used to optimize storage allocation and deallocation in ML programs [TT94]. In systems designed to exploit regions for optimization purposes, effect masking is used to guide deallocation of regions at runtime. As Tofte and Talpin argue, the correctness of this deallocation step is far from obvious and requires a proof [TT93, p.15]. Using natural semantics, they provide such a proof for a pure language, *i.e.*, where the only effects are those which exist in the underlying implementation. In contrast, the present study pertains only to user-allocated heap structures.

The recent work on monads raises another obvious alternative to memory encapsulation in ML, and suggests a connection with effect inference technology. Since the semantics of ML-like languages is easily expressed via a translation to monadic style, it should be possible to reason about memory encapsulation of ML programs by translating to monadic style and using monadic encapsulation. Indeed, Wadler recently explored part of this idea [Wad98]. He shows that effect systems (without masking) can be translated to a monadic framework (without encapsulation) but with additional annotations for effects and regions. In contrast, we relate a restricted effect system (with masking) to the standard monadic framework of Launchbury and Peyton Jones [LP95].

**Results** We introduce an ML-like language with references and one construct that is used to encapsulate references within arbitrary program expressions. The type system for this language uses a simplified form of effect inference and effect masking. We define the semantics using a translation to a monadic language in which encapsulation is enforced by `runST`. Building on Wadler’s result [Wad98], we give a transformation from the first language to the second which preserves typability. This confirms a folklore statement that the traditional monadic framework implements a cheap form of region inference and effect masking. We then revisit the problem of monadic encapsulation and show that our monadic language satisfies a type soundness theorem with respect to a semantics in which any attempt to read or assign to a “garbage” location would result in a stuck state.

From a technical perspective, the design of this semantics that makes deallocation explicit and for which subject reduction holds is our most interesting and challenging result. Indeed, the difficulty of satisfying subject reduction in systems that include procedures and assignments and that perform some kind of encapsulation has been repeatedly noted in various contexts. As early as in 1978, Reynolds points out that his syntactic criteria for control-

ling interference among distinct phrases are not robust with respect to  $\beta$ -reduction [Rey78]. Later, attempts to devise type systems that encapsulate effects for ILC [SRI91] and  $\lambda_{var}$  [ORH93, CO94] are now known to be incorrect as they rely on conditions that are not invariant under reduction [Rab96]. A final blow is the failed proof of subject reduction for monadic state by Launchbury and Sabry [LS97].

To gain some intuition about the problem, and to later understand our solution, we illustrate the subtle interaction of encapsulation and subject reduction using an example. Consider the following term:

```
runST( runST( letST x = ref 0
              in returnST (letST d = returnST x
                           in returnST 2)))
```

The first `runST` creates an outer state thread in which its subexpression is evaluated. This subexpression immediately establishes an inner state thread that allocates a reference to 0, binds it to the name  $x$ , and returns a computation to be executed in the outer state thread. When exiting the scope of the inner `runST`, all references allocated in the inner thread (including the reference named  $x$ ) are garbage-collected, so the computation:

```
letST d = returnST x in returnST 2
```

must be performed in a context where  $x$  is a *dangling pointer*. Luckily, when executed, this computation binds  $x$  to a dummy variable  $d$ , and returns the value 2.

To prove type soundness using subject reduction [WF94], one must find a set of reductions that preserve typability. Unfortunately the “natural” choice of reductions typically introduces dangling pointers which is problematic for type preservation. For example, the reductions suggested by Launchbury and Sabry [LS97] would rewrite the above term as follows:

```
runST( letST d = runST ( letST x = ref 0
                        in returnST (returnST x))
      in runST (letST x = ref 0
                in returnST (returnST 2)))
```

which fails to typecheck because the computation bound to  $d$  establishes a partition which exports a local reference.

One way to deal with the problem is to restrict the monadic language to prohibit a state thread from returning a computation. In his thesis, Rabin pursues this approach and proves that the subject reduction property holds for the “natural” choice for reductions [Rab96]. Kieburz pursues, without problems, a related approach that enforces that all effects are lexically scoped [Kie99]. Despite its success, this approach severely limits the expressiveness of the target monadic language to the extent that simple source level functions that return computations cannot be encapsulated.

Another way to deal with the problem, that does not restrict the language, is to use a proof technique other than subject reduction. Tofte and Talpin pursue such an approach [TT93]. To motivate their more complicated proof technique, they conduct an analysis much like the one we just presented and conclude:

It appears, however, that no matter how we try to define consistency, consistency is *not* preserved under evaluation ... This loss of consistency appears to be at variance with another syntactic

approach to proving soundness, namely proving soundness by proving a subject reduction property. Subject reduction is useful for proving that typing is *preserved* under reduction.

Our solution neither restricts the language nor requires a proof technique other than subject reduction. The idea is to devise more sophisticated notions of reductions that maintain several “garbage regions” in addition to an active live region. Reading and updating is only allowed to pointers in the live region. Dangling pointers are represented as pointers to one of the “garbage regions” and cannot be read or updated.

## 2 Effect Masking and RunST

Our starting point is the language Monadic-ML (MML) that uses an explicit state monad to enforce the locality of references. Its syntax and typing rules are given in Figure 1. Expressions include values, applications, and encapsulated expressions. Values include variables, abstractions, and monadic commands. Intuitively, these commands are “waiting” for a store and hence can be considered as values. Only when the commands occur in the context of runST are they executed.

The type system is essentially an adaptation of Haskell typing which is tailored to accommodate ML-style value polymorphism. Monadic computations are given types of the form  $(\text{ST } \rho \tau)$ . The type indicates that the computation delivers a value of type  $\tau$  and that it occurs in a “partition” indexed by the type variable  $\rho$ . Locations are given a type of the form  $(\text{eref } \rho \tau)$  which indicates that the location contains a value of type  $\tau$  and that it was allocated in the partition indexed by  $\rho$ .

The notation used in our typing rules is standard. Type environments, ranged over by  $\Gamma$ , are finite maps from variables to type schemes. The free type variables of types, type schemes, and type environments have standard inductive definitions and we write  $FV(\tau)$ ,  $FV(\sigma)$ , and  $FV(\Gamma)$  respectively. A substitution,  $S$ , is a partial function from type variables to types. We define the relation  $\succ$  between type schemes and types as  $\forall \alpha_i, \rho_i. \tau \succ \tau'$  iff there exists a substitution  $S$  with  $\text{Dom}(S) = \{\alpha_i, \rho_i\}$  such that  $S\tau = \tau'$ .

The typing rules propagate and unify the indices of memory partitions as follows. A runST marks the beginning of new encapsulated partition: all operations that are determined to occur within that partition are infected with the same index. The type system will only accept the encapsulation of the partition if the index variable is universally quantifiable and is not exported in the result type. Intuitively, the first condition implies that the evaluation of the partition makes no demands on its environment to provide, say, a location to be read or written. If it did, the index of the partition would have been unified with the index of the location in the environment, and universal quantification could not take place. The second condition guarantees that the result of evaluating a partition cannot be used outside the partition to access local locations.

### 2.1 EML

Naturally, we are not interested in forcing ML programmers to write in monadic style. Instead we devise a language Encapsulated-ML (EML) whose syntax and typing

rules are given in Figure 2. Terms in this language are just like the ones in an ML-subset but include the additional form  $(\text{encap } e)$  which should intuitively correspond to runST.

To formalize such an intuition, we design a simplified effect system (see Figure 2) and show it matches the monadic typing discipline. The definitions of type environments, free variables, and the relation  $\succ$  carry over directly from MML. Contrasting our system to Talpin and Jouvelot’s [TJ92], we see a much simplified syntax of effects. Rather than distinguishing among effects corresponding to the initialization, reading, and writing of locations, and keeping track of the types of values involved in these operations, we’re only interested in knowing the region in which an effect occurs. Furthermore, instead of allowing an expression to have effects on different regions, we force region variables to be unified in the same way that the typing rules for the monadic language force  $\rho$  to be the same. For example, the rule for application requires that the effect of evaluating the function, the effect of evaluating the argument, and the latent effect of the function all match.

Some expressions have no visible effects: values and encapsulated expressions. These effects are usually represented using the empty effect  $\emptyset$ . However, as a convenience, we do not allow the empty effect to appear on an arrow in the type of a function. Instead, if the body of a function has the empty effect, then some region variable must be chosen using the inference rule (*does*). These simplifications allow us to do entirely without effect variables as quantification over such a variable amounts to merely an indirect way of quantifying over a region variable. The inference rule for effect masking uses essentially the same observation criterion given by Talpin and Jouvelot; it’s just simplified to our smaller syntax of effects.

Finally, as is well-known, the interaction between polymorphism and references is delicate and we again adopt the simple assumption of value-restricted polymorphism [Wri95]. As noted by Wadler [Wad98], this assumption also allows the translation to monadic form to work out cleanly.

### 2.2 Translation and Correctness

The semantics of EML is given by a standard translation to monadic style. The translation on types, type schemes, type environments, and terms is given in Figure 3.

The following proposition establishes that our EML design including the simplified effect system corresponds to monadic typing. In particular, effect masking in this simple context implies monadic encapsulation.

**Proposition 2.1 (Translation preserves types)**

- (i) If  $\Gamma \vdash_{\text{eff}} e : \tau ! \rho$  then  $\mathcal{T}[\Gamma] \vdash_{\text{mon}} \mathcal{S}[e] : \text{ST } \rho \mathcal{T}[\tau]$ .
- (ii) If  $\Gamma \vdash_{\text{eff}} v : \tau ! \emptyset$  then  $\mathcal{T}[\Gamma] \vdash_{\text{mon}} \mathcal{S}[v] : \text{ST } \rho \mathcal{T}[\tau]$ .
- (iii) If  $\Gamma \vdash_{\text{eff}} (\text{encap } e) : \tau ! \emptyset$  then  $\mathcal{T}[\Gamma] \vdash_{\text{mon}} (\text{runST } \mathcal{S}[e]) : \mathcal{T}[\tau]$ .

The proof is by induction on the structure of type derivations and is mostly straightforward. Beware though, the  $\rho$  mentioned on the right hand side of implication (ii) above

---

$e \in MonExp$	$e ::= v \mid e e' \mid \text{runST } e$
$v \in MonVal$	$v ::= c \mid \lambda x. e \mid x$
$c \in Command$	$c ::= \text{letST } x = e \text{ in } e' \mid e \gg e' \mid \text{returnST } e \mid$
$x \in MonVar$	$\text{ref } e \mid \text{deref } e \mid \text{setref } e e'$

$\tau \in MonType$	
$\alpha \in MonTypeVar$	$\tau ::= \alpha \mid \tau \rightarrow \tau' \mid \text{eref } \rho \tau \mid \text{ST } \rho \tau$
$\rho \in PartitionVar$	

$\pi \in MonTypeScheme$	$\pi ::= \tau \mid \forall \alpha_i, \rho_i. \tau$
-------------------------	--

$$\begin{array}{c}
(\text{var}) \frac{\Gamma(x) \succ \tau}{\Gamma \vdash_{\text{mon}} x : \tau} \\
(\text{abs}) \frac{\Gamma[x \mapsto \tau'] \vdash_{\text{mon}} e : \tau}{\Gamma \vdash_{\text{mon}} \lambda x. e : \tau' \rightarrow \tau} \\
(\text{app}) \frac{\Gamma \vdash_{\text{mon}} e : \tau' \rightarrow \tau \quad \Gamma \vdash_{\text{mon}} e' : \tau'}{\Gamma \vdash_{\text{mon}} e e' : \tau} \\
(\text{ref}) \frac{\Gamma \vdash_{\text{mon}} e : \tau}{\Gamma \vdash_{\text{mon}} (\text{ref } e) : \text{ST } \rho \text{ (eref } \rho \tau)} \\
(\text{deref}) \frac{\Gamma \vdash_{\text{mon}} e : \text{eref } \rho \tau}{\Gamma \vdash_{\text{mon}} (\text{deref } e) : \text{ST } \rho \tau} \\
(\text{setref}) \frac{\Gamma \vdash_{\text{mon}} e : \text{eref } \rho \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash_{\text{mon}} (\text{setref } e e') : \text{ST } \rho \tau} \\
(\text{then}) \frac{\Gamma \vdash_{\text{mon}} e : \text{ST } \rho \tau' \quad \Gamma \vdash_{\text{mon}} e' : \tau' \rightarrow \text{ST } \rho \tau}{\Gamma \vdash_{\text{mon}} (e \gg e') : \text{ST } \rho \tau} \\
(\text{let}) \frac{\Gamma \vdash_{\text{mon}} e : \text{ST } \rho \tau' \quad \Gamma[x \mapsto \text{gen}(\tau', \Gamma, e)] \vdash_{\text{mon}} e' : \text{ST } \rho \tau}{\Gamma \vdash_{\text{mon}} \text{letST } x = e \text{ in } e' : \text{ST } \rho \tau} \\
(\text{return}) \frac{\Gamma \vdash_{\text{mon}} e : \tau}{\Gamma \vdash_{\text{mon}} (\text{returnST } e) : \text{ST } \rho \tau} \\
(\text{run}) \frac{\Gamma \vdash_{\text{mon}} e : \text{ST } \rho \tau}{\Gamma \vdash_{\text{mon}} \text{runST } e : \tau} \quad \rho \notin FV(\Gamma, \tau)
\end{array}$$

$$\text{gen}(\tau, \Gamma, e) = \begin{cases} \forall \alpha_i, \rho_i. \tau & , \text{ if } e = (\text{returnST } x) \text{ or } e = (\text{returnST } \lambda x. e') \\ \tau & , \text{ otherwise} \end{cases}$$

where  $\{\alpha_i, \rho_i\} = FV(\tau) - FV(\Gamma)$

Figure 1: Monadic-ML

---

---

$e \in Exp$        $e ::= v \mid e e' \mid \text{let } x = e \text{ in } e' \mid \text{encap } e \mid$   
 $x \in Id$              $\text{ref } e \mid \text{deref } e \mid \text{setref } e e'$   
 $v \in Val$           $v ::= x \mid \lambda x. e$

$\sigma \in Effect$        $\sigma ::= \emptyset \mid \rho$   
 $\rho \in RegionVar$

$\tau \in Type$           $\tau ::= \alpha \mid \tau \xrightarrow{\rho} \tau' \mid \text{ref}_{\rho} \tau$   
 $\alpha \in TypeVar$

$\pi \in TypeScheme$     $\pi ::= \tau \mid \forall \alpha_i, \rho_i. \tau$

$$\begin{array}{c}
(\text{var}) \frac{\Gamma(x) \succ \tau}{\Gamma \vdash_{\text{eff}} x : \tau ! \emptyset} \\
(\text{abs}) \frac{\Gamma[x \mapsto \tau'] \vdash_{\text{eff}} e : \tau ! \rho}{\Gamma \vdash_{\text{eff}} \lambda x. e : \tau' \xrightarrow{\rho} \tau ! \emptyset} \\
(\text{app}) \frac{\Gamma \vdash_{\text{eff}} e : \tau' \xrightarrow{\rho} \tau ! \rho \quad \Gamma \vdash_{\text{eff}} e' : \tau' ! \rho}{\Gamma \vdash_{\text{eff}} e e' : \tau ! \rho} \\
(\text{ref}) \frac{\Gamma \vdash_{\text{eff}} e : \tau ! \rho}{\Gamma \vdash_{\text{eff}} (\text{ref } e) : \text{ref}_{\rho} \tau ! \rho} \\
(\text{deref}) \frac{\Gamma \vdash_{\text{eff}} e : \text{ref}_{\rho} \tau ! \rho}{\Gamma \vdash_{\text{eff}} (\text{deref } e) : \tau ! \rho} \\
(\text{setref}) \frac{\Gamma \vdash_{\text{eff}} e : \text{ref}_{\rho} \tau ! \rho \quad \Gamma \vdash_{\text{eff}} e' : \tau ! \rho}{\Gamma \vdash_{\text{eff}} (\text{setref } e e') : \tau ! \rho} \\
(\text{let}) \frac{\Gamma \vdash_{\text{eff}} e : \tau' ! \rho \quad \Gamma[x \mapsto \text{gen}'(\tau', \Gamma, e)] \vdash_{\text{eff}} e' : \tau ! \rho}{\Gamma \vdash_{\text{eff}} \text{let } x = e \text{ in } e' : \tau ! \rho} \\
(\text{does}) \frac{\Gamma \vdash_{\text{eff}} e : \tau ! \emptyset}{\Gamma \vdash_{\text{eff}} e : \tau ! \rho} \\
(\text{mask}) \frac{\Gamma \vdash_{\text{eff}} e : \tau ! \rho}{\Gamma \vdash_{\text{eff}} \text{encap } e : \tau ! \emptyset} \rho \notin FV(\Gamma, \tau)
\end{array}$$

$$\text{gen}'(\tau, \Gamma, e) = \begin{cases} \forall \alpha_i, \rho_i. \tau & , \text{ if } e = x \text{ or } e = \lambda x. e \\ \tau & , \text{ otherwise} \end{cases}$$

where  $\{\alpha_i, \rho_i\} = FV(\tau) - FV(\Gamma)$

Figure 2: Encapsulated-ML

---

$$\begin{aligned}
T[\alpha] &= \alpha \\
T[\tau \xrightarrow{\rho} \tau'] &= T[\tau] \rightarrow \text{ST } \rho \ T[\tau'] \\
T[\text{ref } \rho \ \tau] &= \text{eref } \rho \ T[\tau] \\
T[\forall \alpha_i, \rho_i. \tau] &= \forall \alpha_i, \rho_i. T[\tau] \\
T[[x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n]] &= [x_1 \mapsto T[\tau_1]] \dots [x_n \mapsto T[\tau_n]] \\
S[x] &= \text{returnST}(x) \\
S[\lambda x. e] &= \text{returnST}(\lambda x. S[e]) \\
S[e e'] &= S[e] \gg= (\lambda f. (S[e'] \gg= (\lambda a. (f a)))) \\
S[\text{let } x = e \text{ in } e'] &= \text{letST } x = S[e] \text{ in } S[e'] \\
S[\text{encap } e] &= \text{returnST}(\text{runST } S[e]) \\
S[\text{ref } e] &= S[e] \gg= (\lambda x. (\text{ref } x)) \\
S[\text{deref } e] &= S[e] \gg= (\lambda x. (\text{deref } x)) \\
S[\text{setref } e e'] &= S[e] \gg= (\lambda x. (S[e'] \gg= (\lambda y. (\text{setref } x y))))
\end{aligned}$$

Figure 3: Translation

is unconstrained (unlike implication (*i*)). This nondeterminism follows naturally from the cases in EML where we are forced to choose a region variable for expressions having the empty effect before proceeding in the derivation. On the monadic side, such expressions are translated using `returnST` whose typing rule allows us to choose an arbitrary partition variable.

### 3 MML Semantics

The semantics of MML (and hence EML) is given by a syntactic theory in the style of Felleisen *et al.* [FH92]. As is customary in this style of syntactic theory, one needs a syntactic representations of the store and its locations. Unlike the usual representation that uses one global store, we divide the store in several partitions, only the first of which is live. The extended syntax of the language is given in Figure 4.

The semantics maps programs to answers. An MML *program* is a closed term of the form  $(\text{sto } \Delta \ e)$  and an *answer* is a term of the form  $(\text{sto } \Delta \ (\text{returnST } v))$ . Note that the result of translating an EML program to monadic style is not an MML program. To remedy this situation, we implicitly surround the result of the translation with `runST(...)`. Our answers contain unused store bindings much like the answers for the call-by-need calculus [AFM<sup>+</sup>95].

The syntactic theory is formalized using the contexts and reductions given in Figure 5. Contexts express the relative sequencing of pure and imperative operations. Pure computations are sequenced by the usual call-by-value contexts. Store operations take place only in command contexts and only when initiated by a `sto` expression. Thus, evaluation contexts are defined to pursue ordinary call-by-value evaluation until a `sto` expression is reached, in which case we start executing commands inside. Of course, in place of a command we may find an expression that evaluates to a computation. But during this evaluation, the expression may require evaluation of another subpartition, etc. Using the evaluation contexts, the inference rules given in Figure 5 define the stepping relation  $\mapsto$ , and  $\mapsto^*$  as its reflexive and transitive closure.

The first reduction is the familiar  $\beta_v$  rule where, as usual, the notation  $e[v/x]$  stands for the capture-avoiding substitution of the value  $v$  for the variable  $x$  in the expression  $e$ . The second reduction is just an administrative change of syntax which intuitively provides an empty store to be used by a partition. The next two reductions express two of the monad laws [Mog89] needed for evaluation. The next three describe the semantics of references. Other than the fact that the reductions are restricted to use the first partition of the store, the reductions are standard. The last reduction deserves some discussion.

According to our informal intuition about encapsulation, the last axiom could have been:

$$\text{sto } \Delta \ (\text{returnST } v) \longrightarrow v$$

Unfortunately, such an axiom does not preserve typability. The problem is that although the locations contained in  $\theta$  should be inaccessible, the value  $v$  in the above example may still need assumptions about the contents of  $\theta$  to type-check. To demonstrate this, let's informally trace through the execution of an example with a nested partition:

```

runST(
  runST(letST x = ref 0
        in returnST(letST d = ref x
                    in returnST 2)))
 $\mapsto$ 
sto {} (
  sto {(p1,0)} (letST x = returnST p1
                in returnST(letST d = ref x
                            in returnST 2)))
 $\mapsto$ 
sto {} (
  sto {(p1,0)}(returnST(letST d = ref p1
                        in returnST 2)))

```

Here, we have reached a point where the inner partition has completed evaluation during which it has created the

---


$$\begin{array}{ll}
e \in \text{MonExp} & e ::= \dots \mid \text{sto } \Delta e \\
v \in \text{MonVal} & v ::= \dots \mid p \\
p \in \text{Location} & \\
\theta \in \text{Partition} & \theta ::= \{(p_1, v_1), \dots, (p_n, v_n)\} \\
\Delta \in \text{Store} & \Delta ::= \theta \mid \theta, \Delta
\end{array}$$

Figure 4: Extended syntax for MML

---

$$P \in \text{CommandContext} \quad P ::= [] \mid P \gg e \mid \text{letST } x = P \text{ in } e$$

$$A \in \text{ApplicativeContext} \quad A ::= [] \mid A e \mid v A$$

$$R \in \text{ReturnContext} \quad R ::= P[A] \mid P[\text{returnST } A] \mid P[\text{ref } A] \mid P[\text{deref } A] \mid P[\text{setref } A e] \mid P[\text{setref } v A]$$

$$\begin{array}{ll}
(\lambda x. e) v & \longrightarrow e[v/x] \\
\text{runST } e & \longrightarrow \text{sto } \emptyset e \\
\text{sto } \Delta P[(\text{returnST } v) \gg e] & \longrightarrow \text{sto } \Delta P[(e v)] \\
\text{sto } \Delta P[\text{letST } x = (\text{returnST } v) \text{ in } e] & \longrightarrow \text{sto } \Delta P[e[v/x]] \\
\text{sto } \theta, \Delta P[(\text{ref } v)] & \longrightarrow \text{sto } \theta \cup \{(p, v)\}, \Delta P[(\text{returnST } p)] \quad p \text{ fresh} \\
\text{sto } \theta \cup \{(p, v)\}, \Delta P[(\text{deref } p)] & \longrightarrow \text{sto } \theta \cup \{(p, v)\}, \Delta P[(\text{returnST } v)] \\
\text{sto } \theta \cup \{(p, v')\}, \Delta P[(\text{setref } p v)] & \longrightarrow \text{sto } \theta \cup \{(p, v)\}, \Delta P[(\text{returnST } v)] \\
\text{sto } \Delta R[\text{sto } \Delta' (\text{returnST } v)] & \longrightarrow \text{sto } \Delta, \Delta' R[v]
\end{array}$$

$$E \in \text{EvaluationContext} \quad E ::= [] \mid E e \mid v E \mid \text{sto } \Delta P[E] \mid \text{sto } \Delta P[\text{returnST } E] \mid \text{sto } \Delta P[\text{ref } E] \mid \text{sto } \Delta P[\text{deref } E] \mid \text{sto } \Delta P[\text{setref } E e] \mid \text{sto } \Delta P[\text{setref } v E]$$

$$\frac{e \longrightarrow e'}{E[e] \mapsto E[e']} \qquad \frac{}{e \mapsto e} \qquad \frac{e \mapsto e' \quad e' \mapsto e''}{e \mapsto e''}$$

Figure 5: MML semantics

---

location p1. Notice though that although this location is garbage, we cannot simply throw away the resulting store leaving p1 unbound. In fact, the computation delivered by the child partition and executed by the parent will go on to manipulate this garbage location by creating a pointer to it. Since p1 itself is never actually dereferenced or assigned to, this operation doesn't violate encapsulation.

To accommodate this situation, when terms of the form

$$\text{sto } \theta (\text{returnST } v)$$

have been evaluated in the context of a parent partition that demands the value  $v$ , the axiom simply performs the return but appends the current store to the list of stores maintained by the parent. As a technical convenience, the return contexts used in the final reduction rule are defined to enforce that partitions return to their *immediate* parent. Thus, we must account for any position within a monadic expression from which a subpartition can return. This situation is complicated by the fact that pure computations

can produce commands. For this reason, the applicative contexts in Figure 5 are used in the definition of return contexts. The resulting definition closely follows that of evaluation contexts, except that no new encapsulated expressions are entered.

We can now proceed with our evaluation by applying the last reduction rule:

$\longrightarrow$

$$\text{sto } \{\}, \{(p1, 0)\} (\text{letST } d = \text{ref } p1 \text{ in } (\text{returnST } 2))$$

$\longrightarrow$

$$\text{sto } \{(p2, p1)\}, \{(p1, 0)\} (\text{letST } d = \text{returnST } p2 \text{ in } (\text{returnST } 2))$$

$\longrightarrow$

$$\text{sto } \{(p2, p1)\}, \{(p1, 0)\} (\text{returnST } 2)$$

Now, the parent partition has produced a value and since the parent is also the outermost partition, the resulting expression is an answer.

## 4 Type Soundness

Proving type soundness relative to rewriting semantics involves the following standard steps [WF94]:

- Show that every program that isn't an answer already is either reducible according to the semantics, or is faulty.
- Show that reducing a typable program produces another program with the same type.
- Show that faulty programs are untypable.

Once these facts are established, it's easy to see that a typable program must either diverge (reduce forever), or produce an answer of the expected type. Faulty expressions characterize stuck states on which the semantics is undefined.

**Definition 4.1** *The following expressions are faulty:*

1.  $\text{sto } \theta, \Delta P[(\text{deref } p)]$  where  $p \notin \text{Dom}(\theta)$   
(illegal dereference),
2.  $\text{sto } \theta, \Delta P[(\text{setref } p \ e)]$  where  $p \notin \text{Dom}(\theta)$   
(illegal assignment),
3.  $\text{sto } \theta, \Delta b$  where  $b$  is  $p$ , or  $\lambda x.e$   
(illegal encapsulated value),
4.  $(\text{letST } x = b \text{ in } e)$ ,  $(b \gg e)$  where  $b$  is  $p$ , or  $\lambda x.e$   
(illegal value in command context),
5.  $(\text{deref } w)$ ,  $(\text{setref } w \ e)$  where  $w$  is  $c$ , or  $\lambda x.e$   
(non-location in location position), and
6.  $(r \ e)$  where  $r$  is  $p$ , or  $c$   
(non-function in function position).

The first two faulty expressions are the most interesting ones. They state that attempts to read or write from any partition other than the first one is illegal. The remaining faulty expressions do not directly involve references and are standard. Once type soundness is established, the first two cases in the definition above will guarantee that encapsulated expressions really do have the desired behavior.

The first step in the proof is to establish the following proposition.

**Proposition 4.1** *Every MML program is either an answer or can be decomposed into the form  $E[T]$  where  $T$  is either a redex or a faulty term.*

But given the complexity of our contexts and reduction axioms, it is necessary to adopt a stronger induction hypothesis and prove instead that:

**Proposition 4.2** *Every MML expression  $e$  can be decomposed into one of the following forms:*

- (i)  $P[W]$  where  
 $W$  is of the form  $E[T]$ ,  $(\text{returnST } E[T])$ ,  $(\text{ref } E[T])$ ,  $(\text{deref } E[T])$ ,  $(\text{setref } E[T] \ e')$ , or  $(\text{setref } p \ E[T])$  and where  $T$  is a redex, a faulty expression, or a variable not bound in  $e$
- (ii)  $P[X]$  where  $X$  is of the form  $(\text{ref } v)$ ,  $(\text{deref } p)$ ,  $(\text{setref } p \ v)$ ,  $((\text{returnST } v) \gg e')$ , or  $(\text{letST } x = (\text{returnST } v) \text{ in } e')$
- (iii)  $R[Y]$  where  $Y$  is of the form  $\text{sto } \Delta (\text{returnST } v)$
- (iv)  $[Z]$  where  $Z$  is of the form  $\lambda x.e'$ ,  $p$ , or  $(\text{returnST } v)$ .

Although seemingly complex, the statement above is merely being more specific about the kinds of redexes we may find within a term and in what specific contexts they may occur. The proof is a straightforward but tedious structural induction and Proposition 4.1 now follows as an easy corollary.

In order to preserve typability for our intermediate states of evaluation, we need to add the inference rules given in Figure 6. The rule for the `sto` construct may be summarized by saying that we insist that each garbage partition is assigned a different partition variable for its locations and that the partition variable associated with the encapsulated expression matches that of the first partition listed.

**Proposition 4.3 (Subject Reduction)**

*If  $\Gamma \vdash_{\text{mon}} e : \tau$  and  $e \longrightarrow e'$  then  $\Gamma \vdash_{\text{mon}} e' : \tau$ .*

The proof is by case analysis on the reduction step. Subject reduction is easy to establish for our first six reductions from Figure 5. The last rule requires more work to show that the invariants given by the conditions on our inference rule for `sto` are still met when we “garbage collect” the locations of a partition which is finished executing. In fact, the conditions given could be relaxed somewhat, but they have been defined in a way that makes this part of the proof easier. A corollary of Subject Reduction is that standard reduction steps are type preserving.

Finally, showing that the given faulty expressions are untypable is straightforward but it requires that we treat locations differently from ordinary variables, thus the extra typing rule in Figure 6. This merely insures that locations cannot be assigned type schemes, for otherwise we could construct hypothetical intermediate states involving open terms which violate encapsulation.

**Proposition 4.4 (Faulty expressions are untypable)**

*If  $e$  is faulty then there are no  $\Gamma, \tau$  such that  $\Gamma \vdash_{\text{mon}} e : \tau$ .*

The proof is by case analysis of the definition of faulty expressions. We say a program is faulty if it contains a faulty subexpression. An easy corollary is that faulty programs are untypable.

We write  $e \uparrow$  if there exists an infinite standard reduction sequence from  $e$ . Let  $\mathcal{P}$  be an MML program and  $\mathcal{A}$  an answer. We now have the following corollary:

**Proposition 4.5 (Type soundness)**

*If  $\vdash_{\text{mon}} \mathcal{P} : \tau$  then either  $\mathcal{P} \uparrow$  or  $\mathcal{P} \longmapsto \mathcal{A}$  and  $\vdash_{\text{mon}} \mathcal{A} : \tau$ .*

$$\begin{array}{c}
(\text{loc}) \quad \Gamma \vdash p : \tau \text{ if } \Gamma(p) = \tau \\
\\
\text{(sto)} \quad \frac{\forall i. \forall j. \Gamma [p_{1j} \mapsto \text{eref } \rho_1 \tau_{1j}]_{j=1}^{s_1} \cdots [p_{kj} \mapsto \text{eref } \rho_k \tau_{kj}]_{j=1}^{s_k} \vdash v_{ij} : \tau_{ij} \quad \Gamma [p_{1j} \mapsto \text{eref } \rho_1 \tau_{1j}]_{j=1}^{s_1} \cdots [p_{kj} \mapsto \text{eref } \rho_k \tau_{kj}]_{j=1}^{s_k} \vdash e : \text{ST } \rho_1 \tau}{\Gamma \vdash \text{sto } \{(p_{1j}, v_{1j})_{j=1}^{s_1}\}, \dots, \{(p_{kj}, v_{kj})_{j=1}^{s_k}\} \quad e : \tau} P_1 \wedge P_2 \wedge P_3 \\
\\
\begin{array}{l}
(\text{P1}) \quad \rho_i \text{ distinct} \\
(\text{P2}) \quad \rho_i \notin FV(\Gamma) \\
(\text{P3}) \quad \rho_i \notin FV(\tau)
\end{array}
\end{array}$$

where  $1 \leq i \leq k$  and  $1 \leq j \leq s_i$

Figure 6: Additional typing rules

## 5 Conclusions and Future Work

We have conducted an investigation of monadic encapsulation of user-allocated heap structures in ML. We have confirmed folklore that monadic state and encapsulation implements cheap region inference and effect masking. The converse of this statement is, however, less obvious and more work is needed to show that the full expressiveness of `runST` is captured by the direct region inference system. The problem is that the translation from MML to EML should eliminate the monadic combinators but preserve the assignments. Filinski’s technique of representing monads does not provide a “natural” mapping since it would encode the assignments [Fil94]. This idea is, however, reminiscent of the implementation strategy used in the `ghc` compiler to implement monadic state efficiently [LP95]. In this implementation the monadic combinators are inlined as functions that manipulate the store, but then a trick is used to implement these functions without actually passing the store. An analysis of this idea was given by Ariola and Sabry [AS98] and might be the basis for a natural translation from MML to EML that preserves encapsulation.

Our semantics for encapsulation allows us to deal with the garbage collection of inaccessible locations in a simple way which is consistent with type preservation. Moreover, by translating EML programs to monadic form it is possible to implicitly enforce the invariant represented by monadic encapsulation with only minor changes to the type system. This lightweight encapsulation may be useful in concurrent language extensions. Finally, using this result as infrastructure, we intend to revisit the problem in the original setting of Haskell.

## Acknowledgments

We thank the anonymous reviewers for their valuable comments. Both authors were supported by the National Science Foundation under the title: “Career: Controlling Space Properties of Higher-Order Typed Programs”, grant number CCR-9733088.

## References

[AFM<sup>+</sup>95] Zena M. Ariola, Matthias Felleisen, John

Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM Press, New York, 1995.

- [AS98] Zena M. Ariola and Amr Sabry. Correctness of monadic state: An imperative call-by-need calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–74. ACM Press, New York, 1998.
- [CO94] K. Chen and M. Odersky. A type system for a lambda calculus with assignment. In *Theoretical Aspects of Computer Software*. Springer Verlag, LNCS 789, 1994.
- [FH92] Matthias Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.*, 102:235–271, 1992. Tech. Rep. 89-100, Rice University.
- [Fil94] Andrzej Filinski. Representing monads. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, 1994.
- [JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 303–310. ACM Press, January 1991.
- [Kie99] Richard Kieburtz. Taming effects with monadic typing. *ACM SIGPLAN Notices*, 34(1):51–62, January 1999.
- [LG88] Jon M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, ACM Press, January 1988.
- [LP94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.

- [LP95] John Launchbury and Simon L Peyton Jones. State in Haskell. *Lisp Symbol. Comput.*, 8:193–341, 1995.
- [LS97] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In the *ACM SIGPLAN International Conference on Functional Programming*, pages 227–238. ACM Press, New York, 1997.
- [Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In the *IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989. Also appeared as: LFCS Report ECS-LFCS-88-86, University of Edinburgh, 1988.
- [ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56. ACM Press, New York, January 1993.
- [Rab96] Dan Rabin. *Calculi for Functional Programming Languages with Assignments*. PhD thesis, Yale University, 1996. Technical Report YALEU/DCS/RR-1107.
- [Rey78] John C. Reynolds. Syntactic control of interference. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, January 1978.
- [Sab98] Amr Sabry. What is a purely functional language? *J. Functional Programming*, 8(1):1–22, January 1998.
- [SRI91] V. Swarup, Uday Reddy, and E. Ireland. Assignments for applicative languages. In the *Conference on Functional Programming and Computer Architecture*, pages 192–214, 1991.
- [TJ92] Jean-Pierre Talpin and P. Jouvelot. The type and effect discipline. In the *IEEE Symposium on Logic in Computer Science*, pages 162–173, June 1992.
- [Tof90] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, November 1990.
- [TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report 93/15, Department of Computer Science, Copenhagen University, July 1993.
- [TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value calculus using a stack of regions. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, 1994.
- [TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.
- [Wad90] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.
- [Wad98] Philip Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74, Baltimore, September 1998. ACM.
- [WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.
- [Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995. Preliminary Version is Polymorphism for Imperative Languages without Imperative Types, Rice Technical Report TR93-200.