Monadic State: Axiomatization and Type Safety

John Launchbury
Oregon Graduate Institute
P.O. Box 91000
Portland, OR 97291-1000
jl@cse.ogi.edu

Amr Sabry
Department of Computer Science
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Abstract

Type safety of imperative programs is an area fraught with difficulty and requiring great care. The SML solution to the problem, originally involving imperative type variables, has been recently simplified to the syntactic-value restriction. In Haskell, the problem is addressed in a rather different way using explicit monadic state. We present an operational semantics for state in Haskell and the first full proof of type safety. We demonstrate that the *semantic* notion of value provided by the explicit monadic types is able to avoid any problems with generalization.

1 Introduction

When Launchbury and Peyton Jones introduced encapsulated monadic state [11, 12], it came equipped with a denotational semantics and a model-theoretic proof that different state threads did not interact with each other. The encapsulation operator `runST` had a type which statically guaranteed freedom of interaction, and the guarantee relied on a parametricity proof. What the paper failed to provide was any formal reasoning principle at the syntactic level, or any proof of type safety. This paper makes up for those shortcomings. In particular we:

- *axiomatize the monadic-state operations:* this allows us to view the monad of state transformers as an abstract type and understand formally how it should behave. Previously, the choice was between an informal understanding, or a rather heavy-weight denotational description.
- *prove type safety:* if I have a variable that claims to contain a list of integers, say, will it truly do so? or might the type system have become confused? By using the axiomatization as a reduction relation we are able to use standard techniques to show type safety of this system.
- *prove syntactic non-interference:* our proof shows that if ever a state-read or -write is about to be attempted, then the type system guarantees that the reference is local.

To appear in: ACM SIGPLAN International Conference on Functional Programming, 1997

Our formal investigation reveals two subtle points that the previous informal reasoning failed to uncover. First, arbitrary beta-reduction is unsound in a compiler which encodes state-transformers as state-passing functions—any compiler that implemented the denotational semantics directly would have to take great care never to duplicate the state parameter. Second, the recursive state operator `fixST` cannot be interpreted naïvely in call-by-name, but really needs call-by-need to make sense.

2 Imperative Types

We begin by reviewing the issue of type safety. The need for a special care arises in SML because of examples like the following:

```
let val r = ref (fn x => x)
in (fn _ => !r 4) (r := not)
end
```

The variable `r` is given the type $\forall\alpha.(\alpha \to \alpha)ref$, which subsequently unifies with *both* the integer and the boolean uses. Intuitively, the problem is that the `let` has generalized over type variables that actually occur free in the state.

The solution adopted for many years is to have a class of "imperative type variables" [26] and only to generalize over these if the expression being bound by `let` is syntactically a value. Due to Wright's observations [29] this has since been simplified [15] to treat all type variables as if they were imperative type variables, and so collapsing the two tier structure.

Why do these problems not occur in Haskell when using monadic state? The precise answer comes from the proof later in the paper, of course, but we can provide intuition here. When working within the state monad (or any other explicit monad, for that matter) the facilities provided by `let` are given by the use of monadic extension (a *bind* operator in Wadler's terminology [28], `thenST` in the State in Haskell papers, or `>>=` in Haskell 1.3 notation). This takes a computation—a term of type $M\ A$ where M is the monadic type constructor—together with a function of type $A \to M\ B$. Intuitively, the first term is executed, delivering a value of type A which is then passed as an argument to the function whose subsequent computation is also executed. The value is passed according to regular function application, so the type A in $A \to M\ B$ is a regular type, not a type scheme. The intermediate values within a computation are *lambda-bound*, therefore, rather than being let-bound.

That's the intuition. The fact that it works relies on the correct interplay of a number of different aspects of the system (in what comes later, for example, if the typing judgment for runST were relaxed, then type safety would be lost). One way of viewing all this is that the Haskell type system makes a distinction between computations with no effects (which we call *semantic values*) and computations that may have effects (which we simply refer to as *computations*). Semantic values are bound with the `let` construct and their types can be generalized; results of computations are bound with the `>>=` construct and their types cannot be generalized. Hence, like SML, Haskell's `let` only performs generalization over values, not over the results of computations. However, unlike SML, the Haskell notion of value is semantic and hence richer. For example, in Haskell the expression $((\lambda x.x)\ (\lambda y.y))$ is classified as a semantic value whose type can be generalized, but not in SML.

3 State in Haskell

Our source language is an extension of the call-by-name λ-calculus with several constants and two language constructs: `let` and `runST` (later we will add a third).

Definition 1 (Syntax of Terms) *Let x range over a set Vars of variables $\{x, x_1, x_2, \ldots, y, z, \ldots\}$. The set of terms Λ is inductively defined as follows:*

Simple Constants:

$$k \quad ::= \quad () \mid 0 \mid 1 \mid \cdots \mid + \mid \cdots$$

Primitive Store Operations:

$$s \quad ::= \quad \mathtt{newVar}\ e \mid \mathtt{readVar}\ e \mid \mathtt{writeVar}\ e\ e'$$

Syntactic Values:

$$v \quad ::= \quad k \mid \lambda x.e \mid e \mathbin{\mathtt{>>=}} e' \mid \mathtt{returnST}\ e \mid s$$

Terms:

$$e \quad ::= \quad x \mid v \mid e\ e' \mid \mathtt{runST}\ e \mid \mathtt{let}\ \{x_i = e_i\}_i\ \mathtt{in}\ e'$$

We let k range over an unspecified set of simple constants like numbers and addition. The constants `newVar`, `readVar`, and `writeVar` express the usual operations on reference cells. The constants `returnST` and `>>=` are the *unit* and *bind* operations of the state monad respectively. Expressions built up from these state-transformer operations are treated as syntactic values. These syntactic values will be used to specify the operational semantics of the language, but *not* to guide generalization of type variables. The expression (`runST` e) is an eliminator for state-transformer expressions e. The operational intuition behind `runST` is that it executes e in a newly created state thread, returning the final value produced by e while discarding the final state. The state in the thread is neither accessible nor visible from outside the (`runST` e) expression.

In the examples, we sometimes use the Haskell 1.3 `do` notation [19]. This construct acts like a non-recursive `let` over computations and is definable in terms of `>>=`. For example, the following code:

```
let omega = omega
in runST (newVar (3+2) >>= \ p ->
          readVar p >>= \ v1 ->
          readVar p >>= \ v2 ->
          writeVar omega 6 >>= \_ ->
          returnST (v1+v2))
```

where we have used the Haskell notation for lambda expressions (`\x->e`) instead of the mathematical notation ($\lambda x.e$), could be rewritten as follows:

```
let omega = omega
in runST (do p <- newVar (3+2)
             v1 <- readVar p
             v2 <- readVar p
             writeVar omega 6
             returnST (v1+v2))
```

To get an informal feeling for the types and semantics of the expressions, we intuitively explain the evaluation of the above fragment. First, we bind `omega` to some (semantic) value and then create a new state thread. In this thread, we allocate a reference cell p and initialize it to $(3+2)$. The cell is then dereferenced twice and the results are added. Once the values of `v1` and `v2` are determined, no more state operations are performed as the final result is now defined. In other words we have a *lazy* store semantics in which state threads are executed on demand, and values may be returned before the computation has been completed. Hence the assignment to the uncalculable location `omega` does not affect the final result which is 10.

Why bother with lazy stores? The answer is that lazy stores have a clean interaction with the lazy semantics of the underlying language, and provide elegant ways to express imperative functional programs, as the following example illustrates.

The example is drawn from stream-based simulation, in particular simulating a local data cache within a microprocessor (for our purposes here we will assume no cache misses). The contents of the stream represent the values of the input and output wires over time. The input streams to the cache will contain addresses, data, and a boolean read/write flag. The output stream will contain the memory contents for a read, and 0 if a write was performed.

As the cache will contain thousands of randomly accessible locations, a destructive array is the obvious choice for modeling the contents. We therefore use the store primitives `readArr` and `writeArr`, which are the the obvious generalizations to arrays of the operations on single locations. Here's the code in Haskell, where the first parameter to `cache` is the size, the second parameter contains the three input streams, and the result is the output stream:

```
cache :: Int -> ([Int],[Int],[Bool]) -> [Int]
cache size ins =
  runST (do arr <- newArray (0, size-1) 0
            loop arr)

loop arr ((a:as),(d:ds),(True:bs))
  = do x <- readArr arr a
       xs <- loop arr (as,ds,bs)
       return (x:xs)

loop arr ((a:as),(d:ds),(False:bs))
  = do x <- writeArr arr a d
       xs <- loop arr (as,ds,bs)
       return (0:xs)
```

As the state is lazy, the result of each operation becomes available immediately after it is performed—there is no need to wait until *all* the state operations are performed. Indeed, if as we expect the list were infinite, there would be no end to the state operations. Conceptually, as the (infinite) loop '`loop`' executes, it defines more and more of the

$$\frac{}{\Gamma \vdash k : \tau_k} \qquad \frac{\Gamma \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \mathtt{ST}\ \tau^\circ\ \tau \qquad \Gamma \vdash e_2 : \tau \to \mathtt{ST}\ \tau^\circ\ \tau'}{\Gamma \vdash e_1\ \mathtt{>>=}\ e_2 : \mathtt{ST}\ \tau^\circ\ \tau'}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{returnST}\ e : \mathtt{ST}\ \tau^\circ\ \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{newVar}\ e : \mathtt{ST}\ \tau^\circ\ (\mathtt{MutVar}\ \tau^\circ\ \tau)} \qquad \frac{\Gamma \vdash e : \mathtt{MutVar}\ \tau^\circ\ \tau}{\Gamma \vdash \mathtt{readVar}\ e : \mathtt{ST}\ \tau^\circ\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{MutVar}\ \tau^\circ\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{writeVar}\ e_1\ e_2 : \mathtt{ST}\ \tau^\circ\ \mathtt{Unit}} \qquad \frac{}{\Gamma \cup \{x : \forall \alpha_i.\tau\} \vdash x : \tau[\tau_i/\alpha_i]}$$

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau} \qquad \frac{\Gamma \vdash e : \mathtt{ST}\ \alpha^\circ\ \tau}{\Gamma \vdash \mathtt{runST}\ e : \tau}\ \alpha^\circ \notin FV(\Gamma, \tau)$$

$$\frac{\forall j. \Gamma \cup \{x_i : \tau_i\}_i \vdash e_j : \tau_j \qquad \Gamma \cup \{x_i : \forall \alpha_{j_i}.\tau_i\}_i \vdash e' : \tau'}{\Gamma \vdash \mathtt{let}\ \{x_i = e_i\}_i\ \mathtt{in}\ e' : \tau'}\ \alpha_{j_i} \in FV(\tau_i) - FV(\Gamma)$$

Figure 1. Typing Rules

output stream. Alternatively, as more and more of the output stream is demanded by the rest of the program, more and more of the state operations execute. However, and this is an important caveat, the state operations are *linearly ordered* by the use of `>>=`, so the various reads and writes will all be performed in the correct order whatever the pattern of demand.

In summary, the cache component uses state internally purely for efficiency. Externally, however, it is simply a lazy stream transformer, which is exactly what the overall structure of the simulation requires. We have the best of both worlds.

3.1 Types

We take ST τ' τ as the abstract type of state computations. Intuitively a computation of type ST τ' τ takes a state as an argument and delivers a value of type τ together with a new state. Each state thread is indexed by a unique type. Uniqueness is given by universal quantification, and this will be seen to guarantee non-interference among different state threads. The type (MutVar τ' τ) is the type of references allocated from a state indexed by τ' and containing values of type τ.

Definition 2 (Syntax of Types) *Let α range over a set of type variables $\{\alpha, \alpha_1, \alpha_2, \alpha^\circ, \ldots\}$. The set of types is inductively defined as follows:*

$$\begin{array}{rcll} \tau^\circ, \tau & ::= & \mathtt{Unit} \mid \mathtt{Int} \mid \ldots \mid & (\textit{Types}) \\ & & \alpha \mid \tau \to \tau' \mid & \\ & & \mathtt{ST}\ \tau^\circ\ \tau' \mid \mathtt{MutVar}\ \tau^\circ\ \tau' & \\ \sigma & ::= & \forall \alpha.\sigma \mid \tau & (\textit{Type schemes}) \end{array}$$

The typing rules for our language are in Figure 1 and they use the conventions above, that variously decorated instances of α, τ and σ stand for type variable, type, and type scheme respectively. In particular, α° is just a regular type variable and τ° a regular type. The decoration is used to provide a reminder that this variable/type occurs in the state-type position of an ST or MutVar type constructor.

We let Γ scope over type environments (partial mappings from term variables to type schemes). A type judgment $\Gamma \vdash e : \tau$ means that under the assumptions in the type environment Γ, expression e has type τ. To these rules must be added the standard rules for the integer constants etc.

Of these rules, the only one to excite interest is the type of `runST`. If we were not restricted to Milner-style polymorphism [14], we might make `runST` a constant with the type:

$$\mathtt{runST} :: \forall \alpha\ .\ (\forall \alpha^\circ.\mathtt{ST}\ \alpha^\circ\ \alpha) \to \alpha$$

To fit the Hindley-Milner context, we make `runST` a language construct with a typing judgment whose side condition simulates the nested polymorphism.

The reasoning behind the type of `runST` is as follows. Every operation which manipulates a state thread is infected with the type of that state thread: when `>>=` is used to combine operations, the types of the state-thread have to be the same (*i.e.*, they become unified); every location returned by `newVar` has the same state thread type as the thread that created it; and every time a `readVar` or `writeVar` is performed its `MutVar` argument belongs to the same state thread in which the read or write is actually performed.

Then when a state thread is encapsulated by `runST` the type system will only accept the encapsulation if:

1. the type of the state is still a variable; *and*
2. that variable is universally quantifiable.

If these two conditions hold then the state thread should make no demands on its environment to provide, say, a location to be read or written. If it did, the type of the state thread would have been unified with the state type of the location in the environment, and universal quantification could not take place.

Launchbury and Peyton Jones [12] showed that the intuition pans out by using a parametricity proof over the denotational semantics presented in the next section. In particular, they proved that the result of running a state thread is independent of an arbitrary encryption of the locations generated by all other state threads.

The result we present in this paper is stronger, in that we show syntactic non-interference (which certainly implies the earlier behavioral non-interference), and in addition we show type safety.

3.2 Denotational Semantics

The denotational semantics of the language is standard [12]. We present the semantics of reference cells and the state

Domains:

$$\begin{array}{rcl} \rho \in Env & = & \Pi_\tau(Vars_\tau \rightarrow \mathcal{D}_\tau) \\ \theta \in Store & = & (\mathcal{N} \rightarrow (\bigcup_\tau \mathcal{D}_\tau))_\perp \end{array}$$

Meaning function:

$$\mathcal{E} : \Lambda \quad \rightarrow \quad \bigcup_\tau \mathcal{D}_\tau$$

$$\begin{array}{rcll} \mathcal{E}[\![\texttt{runST}\ e]\!]\rho & = & v & \text{where } (v,\theta) = \mathcal{E}[\![e]\!]\rho\ (\mathit{lift}\ \emptyset) \\ \mathcal{E}[\![\texttt{returnST}]\!]\rho\ v\ \theta & = & (v,\theta) & \\ \mathcal{E}[\![\texttt{>>=}]\!]\rho\ v\ v'\ \theta & = & v'\ v''\ \theta' & \text{where } (v'',\theta') = v\ \theta \\ \mathcal{E}[\![\texttt{newVar}]\!]\rho\ v\ \theta & = & \begin{cases} (\perp,\perp) & \text{if } \theta = \perp \\ (\ell, \theta[\ell \mapsto v]) & \text{where } \ell = new(\theta),\ \text{otherwise} \end{cases} & \\ \mathcal{E}[\![\texttt{readVar}]\!]\rho\ \ell\ \theta & = & \begin{cases} (\perp,\perp) & \text{if } \ell \notin dom(\theta) \\ (\theta\ell, \theta) & \text{otherwise} \end{cases} & \\ \mathcal{E}[\![\texttt{writeVar}]\!]\rho\ \ell\ v\ \theta & = & \begin{cases} (\perp,\perp) & \text{if } \ell \notin dom(\theta) \\ ((), \theta[\ell \mapsto v]) & \text{otherwise} \end{cases} & \end{array}$$

Figure 2. Denotational Semantics

monad combinators in Figure 2. We use this semantics as the model against which our axioms are verified.

Another way to express the denotations of the state combinators is to give them Haskell definitions and apply the standard mapping to these definitions. For example, the denotation of (e >>= e') could be derived from the following Haskell definition:

```
\s -> let (x,s') = e s in e' x s'
```

In fact this is how the Glasgow Haskell compiler implements the state combinators. (See Section 8 for more details.)

4 Axiomatic Semantics

The goal is to specify the semantics of our language via a set of local axioms that can be used anywhere inside a term (perhaps as optimizations). The interesting axioms are clearly the ones related to reference cells and the store. The axiomatization of references and stores is generally well-understood for many languages [3, 5, 7, 8, 9, 13, 23, 24], but is fairly subtle for our language.

The elegance of lazy stores has a price: it complicates the semantics of the language. Indeed, some expected axioms are unsound due to the laziness of the store. In many languages it is reasonable to expect that writing an expression e to a location x and then immediately reading the location x returns e [3, 16, 17, 23]. In our syntax, the potential axiom is:

$$\begin{array}{rl} & \texttt{newVar}\ e\ \texttt{>>=}\ \lambda x.\texttt{readVar}\ x\ \texttt{>>=}\ e' \\ = & \texttt{newVar}\ e\ \texttt{>>=}\ \lambda x.e'e \end{array}$$

Unfortunately the axiom is unsound—using the axiom we would be able to transform:

```
let omega = omega
in do d <- writeVar omega 6
      x <- newVar 0
      a <- readVar x
      returnST a
```

into the rather different:

```
let omega = omega
in do d <- writeVar omega 6
      x <- newVar 0
      returnST 0
```

Why are they different? The second makes less demand on the store than does the first. The evaluation of the first term needs access to the store in order to read the contents of location x. But the store cannot be computed since the first state transformer in the thread diverges (it cannot tell which physical location should be updated by the writeVar). In contrast the evaluation of the second term does not need the store at all, and immediately returns 0. Had the location in the writeVar subexpression been a known location, the evaluation of both terms would have been equivalent. This informal analysis suggests a way to fix the problem: only use the axiom in special contexts where the store is guaranteed to be well-defined.

Unfortunately there appears to be no easy syntactic way to represent well-defined stores without resorting to a sequence of newVars followed by a sequence of writeVars as the store can contain cyclic references. To avoid messy syntactic patterns, we introduce a special construct:

$$\texttt{sto}\ \{(p_1,e_1),\ldots,(p_n,e_n)\}\ e$$

that represents a well-defined initialization for the store. In this store, location p_i contains e_i; the expressions e_i are naturally allowed to refer to the other p_j so we can represent cyclic structures in the store. This term form is similar in structure to the $\rho\theta.e$ term form of Wright and Felleisen [8, 13, 29] though the axiomatization is rather different to take account of encapsulation.

Definition 3 (Syntax of Terms) *Let p range over a set of locations $\{p, p_1, p_2, \ldots\}$. We extend the syntax of Definition 1 as follows:*

$$\begin{array}{rcll} v & ::= & \ldots \mid p & \\ e & ::= & \ldots \mid \texttt{sto}\ \theta\ e & \\ \theta & ::= & \{(p_i,e_i)\}_i & (\textit{Store Bindings}) \end{array}$$

The typing rule and denotational meaning for the new constructs are in Figure 3. We view locations as a particular brand of term variable, with sto acting as a binding site. Thus type environments can contain assumptions about the type of locations, just as they do for the original brand of variables, but unlike for term variables, locations can only be bound to types and not type schemes [26]. It is straightforward to check that both the type and denotation of sto generalize those of runST in the sense that:

$$\texttt{sto}\ \emptyset\ e = \texttt{runST}\ e$$

Typing Rules:

$$\frac{}{\Gamma \cup \{p : \tau\} \vdash p : \tau} \qquad \frac{\begin{array}{rcl} \forall j \,.\, \Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i & \vdash & e_j : \tau_j \\ \Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i & \vdash & e : \texttt{ST}\ \alpha^\circ\ \tau \end{array}}{\Gamma \vdash \texttt{sto}\ \{p_i \mapsto e_i\}_i\ e : \tau}\ \alpha^\circ \notin FV(\tau,\Gamma)$$

Semantics:

$$\begin{array}{rcl} \mathcal{E}[\![p]\!]\rho & = & \rho(p) \\ \mathcal{E}[\![\texttt{sto}\ \theta\ e]\!]\rho & = & sto\ \{(\ell_i, \mathcal{E}[\![e_i]\!]\rho') \mid (p_i, e_i) \leftarrow \theta\}\ \mathcal{E}[\![e]\!]\rho' \\ \\ \text{where:}\ \rho' & = & \rho \cup \{(p_i, \ell_i)\} \\ sto\ s\ v & = & \pi_1\ (v\ (lift\ s)) \end{array}$$

Figure 3. Additional Typing Rules and Denotations

In order to typecheck the `sto` expression, `e` has to typecheck (the $\forall j$ is vacuous). The side condition on the type judgment follows from the corresponding side condition on the type judgment for `runST`. We therefore do not consider `runST` independently in the remainder of the paper.

Before giving the axioms, we need to formalize one last aspect of the lazy stores; we must define the position within a state thread from which it is possible to immediately return without performing the rest of the stateful computation.

Definition 4 (Return Contexts R**)** *The contexts are inductively defined as:*

$$R \ ::= \ [\,] \mid e \ \texttt{>>=}\ \lambda x.R$$

In other words, we can ignore all the computations to the left of `>>=`; these will not performed unless they are somehow explicitly demanded. In terms of the `do` notation, it means that we should skip over the list of commands and attempt to execute the last one first.

Figure 4 presents the correct axiomatization of the semantics. The correctness of the axioms is easily established by checking that the two sides of each equation are denotationally equivalent.

The first three axioms are as expected in an applied lambda-calculus. The next three axioms use the new `sto` construct as motivated above; each primitive store operation performs its intended operation on the properly initialized store fragment. The structural axioms correspond to the three monad laws. Finally the return axioms show how to compute the result of a state thread; there is an axiom for each kind of syntactic value.

5 Operational Semantics

Having defined the axioms, we need *evaluation contexts* that guide the use of the rules in a standard reduction sequence leading to the answer. Because of the *lazy* nature of our store, the definitions of the reductions is actually intertwined with the definition of evaluation contexts [1, 2]. Intuitively, "needed" variables within subterms correspond to those variables that occur in evaluation context positions. Therefore, we define evaluation contexts first.

5.1 Evaluation Contexts

Defining evaluation contexts already requires much understanding about the semantics of our language. In our case the definition is rather involved and we proceed slowly.

The definition of return contexts (Definition 4) shows that we should skip over the list of commands and attempt to execute the last one. If this last command requires a variable that results from an earlier computation, then we must attempt to perform that computation. Also if a command attempts to perform an operation that is strict in the store like `newVar`, `readVar`, or `writeVar`, then we must also step back and perform all the earlier computations. Formally we can express these chains of dependencies as follows. The definition uses the yet-to-be-defined evaluation contexts E. At this point the reader may pretend that all evaluation contexts are the empty context to get the intuition behind the concept of dependencies.

Definition 5 (Dependencies D**)** *The first three clauses express that variable x is needed by a state transformer. The last two clauses express that variable x is needed because another sequence of variables was recursively needed.*

$$\begin{array}{rcl} D & ::= & \lambda x.R[E[x]] \\ & \mid & \lambda x.R[\texttt{returnST}\ E[x]] \\ & \mid & \lambda x.R[e\ \texttt{>>=}\ E[x]] \\ & \mid & \lambda x.R[E[x]\ \texttt{>>=}\ D] \\ & \mid & \lambda x.R[s\ \texttt{>>=}\ D] \end{array}$$

The definitions of dependencies and evaluation contexts are mutually recursive.

Definition 6 (Evaluation Contexts E**)** *The set of contexts is inductively defined as:*

$$\begin{array}{rcl} E & ::= & [\,] \mid E\ e \mid k\ E \\ & \mid & \texttt{sto}\ \theta\ R[E] \\ & \mid & \texttt{sto}\ \theta\ R[\texttt{returnST}\ E] \\ & \mid & \texttt{sto}\ \theta\ R[e\ \texttt{>>=}\ E] \\ & \mid & \texttt{sto}\ \theta\ R[E\ \texttt{>>=}\ D] \\ & \mid & \texttt{sto}\ \theta\ (\texttt{readVar}\ E\ \texttt{>>=}\ D) \\ & \mid & \texttt{sto}\ \theta\ (\texttt{writeVar}\ E\ e\ \texttt{>>=}\ D) \end{array}$$

The first three clauses in the definition of evaluation contexts define the usual contexts for call-by-name languages. The remaining contexts are used when evaluated a state thread. The next three contexts combined keep demanding the right argument of `>>=` until they reach the last state transformer in an R sequence. If that state transformer is a `returnST` then we demand the value of its subexpression. If on the other hand, the last state transformer demands a variable, then we backtrack following the previously defined chains of dependencies demanding state transformers on the left of `>>=`. Finally the operations `readVar` and `writeVar` are strict in their first argument which is the location to read or write.

Computational Axioms:

$$
\begin{array}{rcl}
(\lambda x.e)e' & = & e[e'/x] \\
\texttt{let}\ \{x_i = e_i\}_i\ \texttt{in}\ e & = & e[(\texttt{let}\ \{x_i = e_i\}_i\ \texttt{in}\ e_j)/x_j]_j \\
k\ v & = & \delta(k, v) \qquad \text{if defined} \\
\texttt{sto}\ \theta\ (\texttt{newVar}\ e\ \texttt{>>=}\ e') & = & \texttt{sto}\ \theta \cup \{(p, e)\}\ (e'\ p) \\
\texttt{sto}\ \theta \cup \{(p, e)\}\ (\texttt{readVar}\ p\ \texttt{>>=}\ e') & = & \texttt{sto}\ \theta \cup \{(p, e)\}\ (e'\ e) \\
\texttt{sto}\ \theta \cup \{(p, e)\}\ (\texttt{writeVar}\ p\ e'\ \texttt{>>=}\ e'') & = & \texttt{sto}\ \theta \cup \{(p, e')\}\ (e''\ ())
\end{array}
$$

Structural Axioms:

$$
\begin{array}{rcl}
\texttt{returnST}\ e\ \texttt{>>=}\ e' & = & e'\ e \\
(e_1\ \texttt{>>=}\ e_2)\ \texttt{>>=}\ e & = & e_1\ \texttt{>>=}\ \lambda x.(e_2\ x\ \texttt{>>=}\ e) \\
e\ \texttt{>>=}\ \lambda x.\texttt{returnST}\ x & = & e
\end{array}
$$

Return Axioms:

$$
\begin{array}{rcl}
\texttt{sto}\ \theta\ R[\texttt{returnST}\ k] & = & k \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ (\lambda y.e)] & = & \lambda y.\texttt{sto}\ \theta\ R[\texttt{returnST}\ e] \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ (e\ \texttt{>>=}\ e')] & = & (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e])\ \texttt{>>=}\ (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e']) \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ (\texttt{returnST}\ e)] & = & \texttt{returnST}\ (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e]) \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ (\texttt{newVar}\ e)] & = & \texttt{newVar}\ (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e]) \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ (\texttt{readVar}\ e)] & = & \texttt{readVar}\ (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e]) \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ (\texttt{writeVar}\ e\ e')] & = & \texttt{writeVar}\ (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e])\ (\texttt{sto}\ \theta\ R[\texttt{returnST}\ e']) \\
\texttt{sto}\ \theta\ R[\texttt{returnST}\ p] & = & p \qquad \text{if } p \notin dom(\theta)
\end{array}
$$

Figure 4. Axioms

For example, using evaluation contexts and dependencies, we could rewrite the following term:

```
sto {} (newVar (3+2) >>= \x ->
        readVar x >>= \a ->
        readVar x >>= \b ->
        writeVar x (let y=y in y) >>= \_ ->
        returnST (a+b))
```

as:

```
sto {} (newVar (3+2) >>= D)
```

The reasoning is that the last state transformer demands the variable a:

```
sto {} (newVar (3+2) >>= \x ->
        readVar x >>= \a -> R[returnST E[a]])
```

where E is $([\ ] + b)$. Then, using the definition of D, the demand for a propagates to a demand for `readVar` x which demands the result of the `newVar`.

In contrast the term:

```
sto {} (writeVar (let y=y in y) 6 >>= \_ ->
        newVar 0 >>= \x ->
        readVar x >>= \a ->
        returnST a)
```

would be decomposed as follows:

```
E[(let y=y in y)]
```

This formalizes the observation in the previous section that the evaluation of the first term terminates but the evaluation of the second term diverges.

5.2 Faulty Expressions

If typechecking guarantees anything, it is that certain bad expressions never occur. Apart from the usual errors (for example, adding a boolean to a character) we are interested in avoiding a whole group of bad expressions that have to do with the state. These are expressions that attempt to read or write to a state location which is not part of the local thread, or which return a state location as the result of an encapsulated thread. The fact that the type system catches these is perhaps *the* noteworthy aspect of this formulation of state.

Definition 7 (Faulty Terms) *Let p range over locations, v range over syntactic values, and w range over the following syntactic values: k, p, or $(\lambda x.e)$. An expression e is* faulty *if it is one of the following:*

- $v\ e'$, *and v is neither a lambda expression nor a constant k (a non-function in function position),*
- $k\ v$, *and $\delta(k, v)$ is undefined (undefined basic operation),*
- `sto` θ (`readVar` v `>>=` D), `sto` θ (`writeVar` $v\ e'$ `>>=` D), *and v is not a location (a non-location in location position).*
- `sto` θ (`readVar` p `>>=` D), `sto` θ (`writeVar` $p\ e'$ `>>=` D), *and p is* not *in the domain of θ (interference between separate state threads).*
- `sto` θ R[`returnST` p], *and p is in the domain of θ (exporting private locations),*
- `sto` θ $R[w]$, `sto` θ $R[w$ `>>=` $D]$ *(a non-state-operation where one was expected),*
- `sto` θ $(R[e'$ `>>=` $v])$, *and v is not a lambda expression (a non-function in function position),*

5.3 Reductions

We now use evaluation contexts to restrict the axioms in two ways. First, the patterns of some of the axioms are restricted to avoid infinite reduction sequences that perform no useful work, and to avoid interference between the axioms. For example, we certainly would not want to repeatedly rewrite

Computational Reductions:

$$
\begin{array}{rcl}
(\lambda x.e)e' & \longrightarrow & e[e'/x] \\
\texttt{let}\ \{x_i = e_i\}_i\ \texttt{in}\ e & \longrightarrow & e[(\texttt{let}\ \{x_i = e_i\}_i\ \texttt{in}\ e_j)/x_j]_j \\
k\ v & \longrightarrow & \delta(k, v) \quad \text{if defined} \\
\texttt{sto}\ \theta\ (\texttt{newVar}\ e\ \texttt{>>=}\ D) & \longrightarrow & \texttt{sto}\ \theta \cup \{(p, e)\}\ (D\ p) \\
\texttt{sto}\ \theta \cup \{(p, e)\}\ (\texttt{readVar}\ p\ \texttt{>>=}\ D) & \longrightarrow & \texttt{sto}\ \theta \cup \{(p, e)\}\ (D\ e) \\
\texttt{sto}\ \theta \cup \{(p, e)\}\ (\texttt{writeVar}\ p\ e'\ \texttt{>>=}\ D) & \longrightarrow & \texttt{sto}\ \theta \cup \{(p, e')\}\ (D\ ())
\end{array}
$$

Structural Reductions:

$$
\begin{array}{rcl}
\texttt{sto}\ \theta\ R[\texttt{returnST}\ e\ \texttt{>>=}\ D] & \longrightarrow & \texttt{sto}\ \theta\ R[D\ e] \\
\texttt{sto}\ \theta\ R[(e_1\ \texttt{>>=}\ e_2)\ \texttt{>>=}\ D] & \longrightarrow & \texttt{sto}\ \theta\ R[e_1\ \texttt{>>=}\ \lambda x.((e_2\ x)\ \texttt{>>=}\ D)] \\
\texttt{sto}\ \theta\ R[s] & \longrightarrow & \texttt{sto}\ \theta\ R[s\ \texttt{>>=}\ \lambda x.\texttt{returnST}\ x]
\end{array}
$$

Return Reductions: (Orient Return Axioms from left to right.)

Figure 5. Standard Reductions

an expression e to $(e\ \texttt{>>=}\ \lambda x.\texttt{returnST}\ x)$. Second, during evaluation, we only perform reductions that are demanded by an evaluation context.

Figure 5 presents the reductions of the language. The main restrictions with respect to the axioms are that the use of the structural axioms has been restricted to cases where it is actually useful to make progress in a computation. Also the primitive store operations are not performed unless their results are demanded via a chain of dependencies D.

Given the complexity of our evaluation contexts, reductions, and faulty expressions, how do we know, for example, that we didn't forget one kind of faulty expression. The following proposition which is used to prove type soundness later, verifies that the above definitions are consistent and complete.

Proposition 1 *Every term e is either a syntactic value or can be uniquely partitioned into the form $E[T]$ where T is either:*

- *a variable not bound in E,*
- *a faulty term (see Definition 7), or*
- *a redex (see Figure 4).*

Proof. The proof is by induction on the structure of e and proceeds by cases. All cases are straightforward except the case $e = (\texttt{sto}\ \theta\ e')$ which requires an additional induction as follows:

(i) First we show by induction on the number of occurrences of `>>=` in e' (and using the main inductive hypothesis too) that e' must be in one of the following forms:

  – $R[E[T]]$ or $R[e\ \texttt{>>=}\ E[T]]$ or $R[E[T]\ \texttt{>>=}\ D]$ where T is faulty, a redex, or a variable bound in neither R nor E,

  – $R[v]$, where v is not of the form `>>=`,

  – $R[e\ \texttt{>>=}\ v]$ or $R[v\ \texttt{>>=}\ D]$

(ii) Second we show that the main claim applies to the expression $(\texttt{sto}\ \theta\ e')$ where e' is given by one of the forms in (i).

Having split the proof as above, both subproofs are now straightforward. □

Corollary 1 *Every closed term e is either a syntactic value, or the form $E[T]$ where T is either faulty or a redex.*

6 Type Soundness

The type soundness proof closely follows the subject reduction proofs by Wright and Felleisen [29], providing extra evidence that their techniques are widely applicable. There are two cases where the proofs make clear the rôle of the typings we provide, and in particular, the way in which the type rule for `runST` provides safe encapsulation. These cases will be done in some detail.

Once the operational semantics and type system have been defined, the general form of the syntactic type soundness proof is as follows:

(i) Show that reduction in the operational semantics preserves well-typing. This is called subject reduction.

(ii) Show that faulty expressions are not typable.

If programs are closed and well-typed, then we can put together the previous results as follows: By (i), evaluation of the program will only produce well-typed terms. By Corollary 1, every such term is either faulty, or a syntactic value, or contains a standard redex. The first case is impossible by (ii). Thus either the program reduces to a value of the correct type, or it diverges. We prove the above points (i) and (ii) in the remainder of the section.

6.1 Subject Reduction

The subject-reduction lemma states that a well-typed term remains well typed under reduction. Hence, if ever a `readVar` or `writeVar` is performed, for example, the expression extracted from the state will not introduce a type error when it is substituted into the receiving term. In other words, if a variable claims to hold an `Int` $\rightarrow$ `Int` function, then indeed it does (and not a `Bool` $\rightarrow$ `Bool` function as in the introduction).

The reason that this works out correctly is that no locations can ever be assigned a type scheme such as $\{p_3 : \forall\alpha.\texttt{MutVar}\ \alpha^\circ\ \alpha\}$. If such a typing were possible, then we could easily duplicate the counterexample in the introduction.

As usual, the proof of subject reduction relies on other standard lemmas, most notably the substitution lemma.

Lemma 1 (Substitution) *If $\Gamma[x \mapsto \forall\alpha_i.\tau'] \vdash e : \tau$ and $x \notin dom(\Gamma)$ and $\Gamma \vdash e' : \tau'$ and $\{\alpha_i\}_i \cap FV(\Gamma) = \emptyset$ then $\Gamma \vdash e[e'/x] : \tau$*

In our context this lemma is a generalization of Wright and Felleisen's Lemma 4.4 in two ways: we require substitution of arbitrary expressions rather than of syntactic values only; and we need to show the `sto` form causes no problems. The first of these generalizations is handled by a trivial extension of the proof of Lemma 4.4, and the second similarly from the proof of the corresponding lemma dealing with the $\rho\theta.e$ form (Lemma 5.3).

Once the substitution lemma is shown, subject reduction follows fairly easily.

Lemma 2 (Subject Reduction) *If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$*

Proof. The proof proceeds by case analysis on the reductions $e \longrightarrow e'$. Most of the cases are standard, and are a minor generalization of the proof found in Wright and Felleisen, so we will not rehearse them here. The only interesting cases are for the various instances of `sto`. We will exhibit two instances to show the general form.

Case:
$\texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{readVar}\ p_k\ \texttt{>>=}\ D) \longrightarrow$
$\texttt{sto}\ \{p_i \mapsto e_i\}_i\ (D\ e_k)$ where there exists an i such that $p_i = p_k$.

By assumption we know that:

$$\Gamma \vdash \texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{readVar}\ p_k\ \texttt{>>=}\ D) : \tau,$$

but in order to be able to deduce this we must have been able to show all of the following:

- for all j, $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e_j : \tau_j$,
- $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash p_k : \texttt{MutVar}\ \alpha^\circ\ \tau'$,
- $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash D : \tau' \to \texttt{ST}\ \alpha^\circ\ \tau$

where $\alpha^\circ \notin FV(\tau, \Gamma)$ (the latter two judgments follow after an application of the rule for `>>=` and for `readVar`). It is clear from the second of these that τ' and τ_k are equal.

In order to typecheck the right hand side we need to be able to show that:

- for all j, $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e_j : \tau_j$,
- $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e_k : \tau'$,
- $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash D : \tau' \to \texttt{ST}\ \alpha^\circ\ \tau$

where, again, $\alpha^\circ \notin FV(\tau, \Gamma)$. Given that τ' is equal to τ_k, all these follow from the above.

Case:
$\texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{returnST}\ (\lambda y.e)) \longrightarrow$
$\lambda y.\texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{returnST}\ e)$

Again, by assumption we know that:

$$\Gamma \vdash \texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{returnST}\ (\lambda y.e)) : \tau \to \tau'$$

(here we have jumped to the conclusion that the result type must be a function type—it just simplifies the presentation). In order to be able to deduce this judgment we must have been able to show:

- for all j, $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e_j : \tau_j$,
- $\Gamma \cup \{y : \tau\} \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e : \tau'$

where $\alpha^\circ \notin FV(\tau \to \tau', \Gamma)$.

In order to typecheck:

$$\Gamma \vdash \lambda y.\texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{returnST}\ e) : \tau \to \tau'$$

we must show that:

$$\Gamma \cup \{y : \tau\} \vdash \texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{returnST}\ e) : \tau'$$

which, in turn, requires all of the following:

- for all j, $\Gamma \cup \{y : \tau\} \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e_j : \tau_j$,
- $\Gamma \cup \{y : \tau\} \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e : \tau'$

where, this time, $\alpha^\circ \notin FV(\tau', \Gamma \cup \{y : \tau\})$. In fact, the side condition is the only thing that requires any thought, and that follows immediately from the fact that $FV(\tau \to \tau', \Gamma) = FV(\tau', \Gamma \cup \{y : \tau\})$ since y does not occur in Γ. This final case is one place which motivates the choice of the side condition on α° in the type rule for `sto`. It clearly would not be enough simply to restrict α° from appearing in the free type variables of Γ without mentioning the result type τ.

The other cases all follow the same form, so concluding the proof. □

6.2 Faulty Expressions

Lemma 3 *If an expression e is faulty, then it is not typable.*

Proof. Each case in the Definition of faulty expressions (Definition 7) is treated separately. We show how expressions with interfering state threads are not typable:

Case: $\texttt{sto}\ \{p_i \mapsto e_i\}_i\ (\texttt{readVar}\ p\ \texttt{>>=}\ D)$ where $p \notin \{p_i\}_i$. To typecheck the expression in a context Γ, we must show the following:

- for all j, $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash e_j : \tau_j$, and
- $\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash \texttt{readVar}\ p\ \texttt{>>=}\ D : \texttt{ST}\ \alpha^\circ\ \tau$

where $\alpha^\circ \notin FV(\tau, \Gamma)$. To satisfy the second requirement, we must show that there exists a τ' such that:

$$\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash \texttt{readVar}\ p : \texttt{ST}\ \alpha^\circ\ \tau'$$

which in turn requires that:

$$\Gamma \cup \{p_i : \texttt{MutVar}\ \alpha^\circ\ \tau_i\}_i \vdash p : \texttt{MutVar}\ \alpha^\circ\ \tau'$$

By assumption, $p \notin \{p_i\}_i$, hence we require:

$$\Gamma \vdash p : \texttt{MutVar}\ \alpha^\circ\ \tau'$$

which implies that Γ must contain an entry $p : \texttt{MutVar}\ \alpha^\circ\ \tau'$. But the side condition on `sto` states that α° is not a free type variable in Γ: a contradiction. In other words, typechecking fails if there is any possibility of a "segmentation fault" across state threads. □

7 Other State Operations

The State in Haskell paper [12] presented two other operations on the state. The first `eqVar` tests for equality of locations; it has the type:

$$\texttt{eqVar} :: \texttt{MutVar}\ \alpha^{\circ}\ \tau \rightarrow \texttt{MutVar}\ \alpha^{\circ}\ \tau \rightarrow \texttt{Bool}$$

It introduces no difficulties to the foregoing material. The second, `fixST`, does.

The purpose of `fixST` is to allow recursive bindings *within* the state monad. The usual recursion gained from `let` allows us to define recursive state transformers (`while`-loops and the like), but `fixST` provides and entirely new facility. Using the `do`-notation we might like to write:

```
data IntNVar s = Pair Int (MutVar s IntNVar)

makeLoop = do v <- newVar (Pair 7 w)
              w <- newVar (Pair 2 v)
              returnST w
```

in which the v and w are both in scope for each of the `newVar` operations. Upon executing these operations, the store would construct a cycle, and the location w would be returned. Note that even though the definition is recursive, the two store operations are each performed once only.

For recursive definitions like this, one would expect to use a fixed point operator, and this case is no exception. We want an operator:

$$\texttt{fixST} : (\alpha \rightarrow \texttt{ST}\ \alpha^{\circ}\ \alpha) \rightarrow \texttt{ST}\ \alpha^{\circ}\ \alpha$$

From a denotational perspective this is fine. We could define the meaning of `fixST` as:

$$\begin{array}{rcl} \mathcal{E}[\![\texttt{fixST}]\!]\rho\ f\ \theta & = & \bigsqcup_{i\geq 0} g^{i}(\bot,\bot) \\ \text{where } g(p) & = & f\ (\pi_1\ p)\ \theta \end{array}$$

or more loosely, as the expansion:

```
fixST e = \s -> let (x,s') = e x s
                in (x,s')
```

Using `fixST` and the usual `do`-notation, we could define our recursive store above by:

```
makeLoop = fixST
  (\w -> do v <- newVar (Pair 7 w)
            w' <- newVar (Pair 2 v)
            returnST w')
```

There are however two problems with all this. First, the inclusion of `fixST` breaks some axioms that would otherwise be sound. However, that is a price we have to pay if we desire the functionality of `fixST`. Second, and more seriously, the call-by-name axiomatization of `fixST` is problematic. We will describe each of these in more detail.

7.1 Sequencing Axioms

In the absence of `fixST`, it is reasonable to expect that a read and a write that refer to different variables can be performed in any order as they will not affect each other. Because of aliasing this is often difficult to determine, but we might expect the following to hold:

$$\begin{array}{rl} & \texttt{newVar}\ e\ \texttt{>>=}\ \lambda x.\texttt{writeVar}\ y\ e'\ \texttt{>>}\ e'' \\ = & \texttt{writeVar}\ y\ e'\ \texttt{>>}\ \texttt{newVar}\ e\ \texttt{>>=}\ \lambda x.e'' \quad (x \neq y) \end{array}$$

as x and y "clearly" refer to different variables. Unfortunately such an axiom is unsound in the presence of `fixST` as it equates the following two terms:

```
fixST (\y->newVar 0 >>= \x->
           writeVar y 1 >>
           returnST x)

fixST (\y->writeVar y 1 >>
           newVar 0 >>= \x->
           returnST x)
```

According to the semantics, the denotation of the first term is non-bottom and the denotation of the second term is bottom. To understand the problem, we expand and inline the state combinators:

```
\s0->let (y,s1) = let (x,s2) = newVar 0 s0
                      (_,s3) = writeVar y 1 s2
                  in (x,s3)
     in (y,s1)

\s0->let (y,s1) = let (_,s2) = writeVar y 1 s0
                      (x,s3) = newVar 0 s2
                  in (x,s3)
     in (y,s1)
```

When applied to a store, the evaluation of the first term demands `(x,s3)` which demands `(x,s2)`. Thus the first computational step is to create a location `x` with initial value `0`. This binds `y` to the new location which makes the write operation well-behaved. In contrast the evaluation of the second term demands `(x,s3)` which demands `s2` (remember that `newVar` is strict in its store argument) which demands `y` which demands `x` which demands `s2`. In other words the evaluation of the second term diverges.

To address this problem we are simply careful not to include axioms that change the order of state operations, even when such changes are apparently safe. These axioms are not needed for evaluation anyway.

7.2 Call-by-Name and `fixST`

In general the axiomatic semantics of fixed point computations is expressed by unwinding the recursion. For example, the usual semantics for the fixed point combinator on values `fix` is:

$$\texttt{fix}\ e = e\ (\texttt{fix}\ e)$$

Using this idea, it is a simple exercise to derive the following semantic equivalence for `fixST`:

$$\begin{array}{rl} & \texttt{sto}\ \theta\ (\texttt{fixST}\ e'\ \texttt{>>=}\ e) \\ = & \texttt{sto}\ \theta\ (e'\ (\texttt{sto}\ \theta\ (\texttt{fixST}\ e'))\ \texttt{>>=}\ e) \quad (*) \end{array}$$

To understand the intuition, remember that $(\texttt{sto}\ \theta\ e)$ evaluates the computation e in the state thread θ. If this evaluation terminates, it yields a final *value* and a final *state*. The final state is ignored and only the final value is returned. Thus the equivalence illustrates that only values (but not stores) are propagated across the unwindings.

So what's the problem? The problem is that the right hand of the equivalence does not typecheck! Consider a location in e'; both the outer and inner state threads may attempt to access that location. This is exactly the kind of situation that the typing of `runST` is designed to avoid!

The discussion points to a fundamental problem with fixST. The construct fixST expresses the computation of a recursive value that takes one input store and returns one final store. Any unwinding of the recursive computation must do some non-standard manipulation of the store, *e.g.*, duplicating a store, or ignoring a store. As we have seen duplication of the input store is likely to produce untypable terms and it is impossible to ignore a store using our combinators. In other words, it appears that in a call-by-name world the combinator fixST needs to be restricted in order to make sense. Here we restrict its type to prevent any recursive values that use the store:

$$\frac{\Gamma \vdash e : \tau \rightarrow \mathtt{ST}\ \alpha^\circ\ \tau}{\Gamma \vdash \mathtt{fixST}\ e : \mathtt{ST}\ \alpha^\circ\ \tau}\ \alpha^\circ \notin FV(\tau)$$

With this restriction our axiom $(*)$ is sound and typable.

In versions of Haskell incorporating fixST, there is no such type restriction. We believe that the full rule will only make sense in an explicit call-by-need setting in which reductions do not duplicate state threads.

8 Implementing Monadic State in GHC

The Glasgow Haskell compiler (GHC) implements monadic state by expanding the monadic combinators to pure Haskell. This is fine so long as either we refrain from performing arbitrary call-by-name transformations on the code, or we give up on destructive update. Part of the motivation for this current work was the desire to be able to retain both.

In more detail, the common implementation strategy [12] for Haskell's extension with built-in monads is to:

1. translate the source programs by expressing and inlining returnST, >>=, and runST in the intermediate language of the compiler,

2. apply full compiler optimizations to the resulting intermediate programs, and

3. instruct the code generator not to generate any code to pass the state around and to generate *destructive* versions of newVar, readVar, and writeVar that operate on a global store.

Following this strategy, consider the following source program:

$$\begin{array}{ll} \mathtt{runST}\ (& \mathtt{newVar}\ 0\ \mathtt{>>=}\ \lambda p. \\ & \mathtt{writeVar}\ p\ 5\ \mathtt{>>} \\ & \mathtt{readVar}\ p\ \mathtt{>>=}\ \lambda v. \\ & \mathtt{returnST}\ v) \end{array}$$

whose evaluation according to the denotational semantics produces 5. After inlining the monadic combinators (runST, >>=, and returnST), and doing some simplifications, we get a program in the compiler's intermediate language:

```
fst (let (p,s1) = newVar 0 s0
         (_,s2) = writeVar p 5 s1
         (v,s3) = readVar p s2
     in (v,s3))
```

where s0 is the initial store. If the compiler only performs call-by-need optimizations that only duplicates syntactic values, the evaluation of this intermediate program produces the correct answer 5 even if the operations newVar, writeVar, and readVar ignore the store argument and perform side-effects on a global store. However, using the call-by-name reasoning principles that are valid in Haskell, we can transform this program as follows:

```
fst (let d1 = newVar 0 s0
         d2 = writeVar (fst d1) 5 (snd d1)
         (v,s3) = readVar (fst d1) (snd d2)
     in (v,s3))
=
fst (let d1 = newVar 0 s0
         (v,s3) = readVar (fst d1)
                          (snd (writeVar (fst d1)
                                         5
                                         (snd d1)))
     in (v,s3))
=
fst (let (v,s3) = read (fst (newVar 0 s0))
                       (snd (writeVar
                                (fst (newVar 0 s0))
                                5
                                (snd (newVar 0 s0))))
     in (v,s3))
```

If newVar were a pure function, then all the occurrences of (newVar 0 s0) would evaluate to the same location, and the program would evaluate to the expected answer 5. However, an implementation of the operations newVar, writeVar, and readVar that ignores the store argument and performs side-effects on a global store will not produce the answer 5. To understand why, note that the expression (newVar 0 s0) has been duplicated several times; each evaluation of this expression will create a *fresh* location. It follows that the location in which 5 is written is not the same location from which we attempt to read.

The counterexample reveals that call-by-name and call-by-need evaluations of the intermediate program do not coincide. In other words, the compiler's intermediate language is not purely functional and hence must be optimized with care. Not only can β steps in the compiler cause severe performance problems, for example by duplicating expensive computations [2], but more drastically, they are unsound. Fortunately, even before the monadic extensions, most Haskell compilers were careful not to duplicate work and hence refrained from using β steps for performance reasons. Consequently, the addition of assignments to the back end did not cause any problems for such compilers.

9 Related Work

The λ_{var} system [17] is very similar in spirit to state in Haskell, and hence to the work presented here. A pure construct was introduced that played the role of runST in that it encapsulated imperative computations, guaranteeing their external purely functional behavior. Two methods were presented by which this can be achieved. The first was to demand an explicit expansion of the whole of the spine of the monadic computation so that a run-time check could ensure that the variables referenced were indeed local. Of course this would be prohibitively expensive in practice, and it seems impossible to generalize to lazy state.

As an alternative, a type system was proposed which statically ensured that the state threads were pure [4]. Like early versions of ML, the type system had two sorts of type variables (applicative and imperative). In addition, the typing judgment for `pure` demanded that only applicative types appeared in the type environment and in the result type—much more restrictive than the Haskell solution. Unfortunately, the type system is now known to be incorrect. Reductions may change the set of free variables in a term, so the purity condition, which only restricts the types of free variables, can be circumvented. As a consequence, subject reduction fails. The problem was corrected by adapting the Haskell solution [20].

The work on region inference is also remarkably similar [27]. Our `sto` construct is essentially creating a new region and initializing it. However, in contrast to the region language, an expression in our language cannot access variables in several regions.

The type-based encapsulation works well in Haskell because the explicit use of the state monad (and others) provided a ready home for the extra type variable. Could such a thing be done in ML? One method might be through something like effects annotations [25] or, indeed, through a region inference system, but there are many details to be worked out.

Finally, the parametric models of local variables have strong semantic similarities to the work here [18]. A denotational semantics has to generate new local variables every time a new block is entered. By using parametricity, these variables can be hidden from the outside world. In our setting, where variables are first class values, we need to have a similar feature *within* the language, hence the type of `runST`.

10 Conclusion and Future Work

In terms of its relation to future developments in understanding and controlling effects, the most exciting aspect of this paper is the clarification of the mechanism for encapsulation. This mechanism is so powerful that it permits the type system to guarantee that references (and hence effects) cannot be perceived outside of the encapsulation barrier. This means that a computation could use state internally to achieve efficiency, yet show a guaranteed pure face to the outside world, without having to do any expensive run-time checks.

10.1 Typechecked Segmentation

Given the spread of *run-time* mechanisms used for checking locality of references, from operating system segmentation checks to mechanisms for encapsulating effects in functional languages [10, 21, 22] it is perhaps surprising to discover that the type system is quite strong enough to do it statically. Of course, the fact that type systems can figure out the lifetimes of references has been known for some time [6]. What distinguishes our solution based on `runST` is that it requires such minor changes to the language.

To give some ideas of the accuracy achieved by this mechanism it is worth noting that it is quite feasible to have one state thread manipulate locations belonging to another quite separate thread. As long as no attempt is made to dereference these other locations, the type system does not unify the state-type parameters of the locations with the state-type of the thread. Under these conditions, the host thread could build and traverse a graph containing the foreign locations, perhaps duplicate or discard the locations, or build them into data structures, eventually to be returned, presumably, to the owning thread for it to dereference at will. Through all this the type system is able to track that the threads do not interfere with each other, and that they are indeed separate state threads.

10.2 Nested Scopes

The principle behind `runST` can be generalized to provide nested scope. We could introduce two constants:

$$\begin{array}{rcl} \texttt{blockST} & :: & (\forall\beta.\texttt{ST}\ (\alpha,\beta)\ \tau) \rightarrow \texttt{ST}\ \alpha\ \tau \\ \texttt{importVar} & :: & \texttt{MutVar}\ \alpha\ \tau \rightarrow \texttt{MutVar}\ (\alpha,\beta)\ \tau \end{array}$$

(actually, like `runST`, we would introduce `blockST` as a language construct with a typing judgment that simulated the nested polymorphism in its type). Using `importVar` we can explicitly allow variables from an enclosing scope to be manipulated by the inner scope. For example,

```
f = do a <- newVar 0
       b <- newVar True
       blockST (g (importVar a))
       v <- readVar a
       returnST v

g x = do c <- newVar "hello"
         writeVar x 1
         returnST ()
```

The type for `blockST` guarantees that the variable `c` is only used in the inner scope. It is not exported to the outer scope in any way. This provides a firm notion of *local pointer*, one that cannot be accessed outside the block.

In practice, we might like even finer control than this. Extending the system to provide only read access in inner scopes (and not write access) is easy to achieve (`MutVar`s need to take two state variables: one to say which thread can do reads, the other to say which can do writes), but the more challenging control of, say, dividing an array into two distinct parts to be worked on concurrently seems much harder to achieve.

10.3 Type-encapsulated Exceptions

State is a natural first application for this technique, but it is bound to be applicable to others like exceptions and continuations. The trick to success here is finding a formulation of the basic operations which is both natural and convenient while succumbing to the extra type variable technique.

As an example, we present a formulation of exceptions which allows us to use the same type-encapsulation technique. We need two new abstract type constructors: the monad of exception-raising computations $\texttt{ET}\ \alpha\ \tau$, and the type $\texttt{Exn}\ \alpha$ of exceptions raised in thread α; together with the following operations:

$$\begin{array}{rcl} \texttt{runET} & :: & \tau \rightarrow (\forall\alpha.\texttt{ET}\ \alpha\ \tau) \rightarrow \tau \\ \texttt{newExn} & :: & \texttt{ET}\ \alpha\ (\texttt{Exn}\ \alpha) \\ \texttt{raiseExn} & :: & \texttt{Exn}\ \alpha \rightarrow \texttt{ET}\ \alpha\ \tau \\ \texttt{handleExn} & :: & \texttt{ET}\ \alpha\ \tau \rightarrow (\texttt{Exn}\ \alpha, \texttt{ET}\ \alpha\ \tau) \rightarrow \texttt{ET}\ \alpha\ \tau \end{array}$$

The first argument to `runET` is a default value used to replace any uncaught exception. The operation `newExn` dynamically creates a new exception, which can be handled by any handler that is executed within the same monadic thread.

10.4 Call-by-need Semantics

Within the narrower scope of state in Haskell, this paper suggests that there is something significant to be gained in moving to an explicit call-by-need semantics of monadic state, in that `fixST` would lose its side condition. It would be in this setting also that we should expect to be able to use an axiomatic semantics to establish formally the correctness of destructive implementations of monadic state.

Acknowledgments

We would like to thank Zena Ariola for many discussions about the axiomatic and operational semantics. The reviewers provided many comments that improved the presentation.

References

[1] Ariola, Z. M., and Felleisen, M. The call-by-need lambda calculus. To appear in the *Journal of Functional Programming*, 1996.

[2] Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages* (1995), pp. 233–246.

[3] Boehm, H.-J. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 637–655.

[4] Chen, K., and Odersky, M. A type system for a lambda calculus with assignment. In *Theoretical Aspects of Computer Software* (1994), Springer Verlag, LNCS 789.

[5] Crank, E., and Felleisen, M. Parameter-passing and the lambda calculus. In *ACM Symposium on Principles of Programming Languages* (1991), pp. 233–244.

[6] Damas, L. M. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[7] Felleisen, M., and Friedman, D. A calculus for assignments in higher-order languages. In *ACM Symposium on Principles of Programming Languages* (1987), pp. 314–325.

[8] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 102* (1992), 235–271. Technical Report 89-100, Rice University.

[9] Hoare, C., Hayes, I., Jifeng, H., Morgan, C., Roscoe, A., Sanders, J., Sorensen, I., Spivey, J., and Sufrin, B. Laws of programming. *Communications of the ACM 30*, 8 (1987), 672–686.

[10] Launchbury, J. Lazy imperative programming. Technical Report, Yale University, 1993. ACM SIGPLAN Workshop on State in Programming Languages.

[11] Launchbury, J., and Peyton Jones, S. L. Lazy functional state threads. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 24–35.

[12] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8* (1995), 193–341.

[13] Mason, I., and Talcott, C. L. Equivalence in functional languages with effects. *Journal of Functional Programming 1*, 3 (July 1991), 287–327.

[14] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978).

[15] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML, Revised 1996*. Forthcoming, 1996.

[16] Odersky, M. A syntactic theory of local names. Technical Report YALEU/DCS/RR-965, Yale University, 1993.

[17] Odersky, M., Rabin, D., and Hudak, P. Call by name, assignment, and the lambda calculus. In *ACM Symposium on Principles of Programming Languages* (Jan. 1993), pp. 43–56.

[18] O'Hearn, P. W., and Tennent, R. D. Relational parametricity and local variables. In *ACM Symposium on Principles of Programming Languages* (1993).

[19] Peterson, John, et al. Report on the programming language Haskell (version 1.3). Technical Report YALEU/DCS/RR-1106, Yale University, 1996.

[20] Rabin, D. *Calculi for Functional Programming Languages with Assignments*. PhD thesis, Yale University, 1996. Technical Report YALEU/DCS/RR-1107.

[21] Riecke, J. G. Delimiting the scope of effects. In *Conference on Functional Programming and Computer Architecture* (1993), pp. 146–155.

[22] Riecke, J. G., and Viswanathan, R. Isolating side effects in sequential languages. In *ACM Symposium on Principles of Programming Languages* (1995), pp. 1–12.

[23] Sabry, A., and Field, J. Reasoning about explicit and implicit representations of state. Technical Report YALEU/DCS/RR-968, Yale University, 1993. ACM SIGPLAN Workshop on State in Programming Languages, pages 17-30.

[24] Swarup, V., Reddy, U., and Ireland, E. Assignments for applicative languages. In *Conference on Functional Programming and Computer Architecture* (1991), pp. 192–214.

[25] Talpin, J., and Jouvelot, P. The type and effect discipline. In *IEEE Symposium on Logic in Computer Science* (June 1992), pp. 162–173.

[26] Tofte, M. Type inference for polymorphic references. *Information and Computation 89*, 1 (November 1990), 1–34.

[27] Tofte, M., and Talpin, J. Implementing the call-by-value calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages* (1994), pp. 188–201.

[28] Wadler, P. Comprehending monads. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 61–78.

[29] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. Technical Report 91-160, Rice University, April 1991. Final version in *Information and Computation* **115** (1), 1994, 38–94.