J. Functional Programming **1** (1): 1–000, January 1993

Monadic Encapsulation of Effects: A Revised Approach (Extended Version)

E. Moggi †
DISI, Univ. of Genova, v. Dodecaneso 35
16146 Genova, Italy

Amr Sabry ‡
Computer Science Department, Indiana University
Bloomington, IN 47405, USA

Abstract

Launchbury and Peyton Jones came up with an ingenious idea for embedding regions of imperative programming in a pure functional language like Haskell. The key idea was based on a simple modification of Hindley-Milner's type system. Our first contribution is to propose a more natural encapsulation construct exploiting higher-order kinds, which achieves the same encapsulation effect, but avoids the *ad hoc* type parameter of the original proposal. The second contribution is a type safety result for encapsulation of strict state using both the original encapsulation construct and the newly introduced one. We establish this result in a more expressive context than the original proposal, namely in the context of the higher-order lambda-calculus. The third contribution is a type safety result for encapsulation of *lazy* state in the higher-order lambda-calculus. This result resolves an outstanding open problem on which previous proof attempts failed. In all cases, we formalize the intended implementations as simple big-step operational semantics on untyped terms, which capture interesting implementation details not captured by the reduction semantics proposed previously.

1 Introduction

Launchbury and Peyton Jones (1995) came up with an ingenious idea for encapsulating *regions* of imperative programming in a pure functional language. More specifically, they introduced a simple modification of Hindley-Milner's type system, and proved (using logical relations) that if a program is well-typed (in a restricted system, where all locations contain expressions of a fixed *base* type), then different state threads do not interfere. Subsequently Launchbury and Sabry (1997) gave a formal account of type safety (for the whole system) by attempting to prove

† Research partially supported by MURST and ESPRIT WG APPSEM.
‡ Worked started at the University of Oregon. Supported by the National Science Foundation under Grant No. CCR-9733088.

subject reduction for a reduction semantics. Unfortunately, there is a bug in the attempted proof, which "can be traced to the complicated semantics of lazy state threads" (Semmelroth & Sabry, 1999). However, Semmelroth and Sabry (1999) are able to adapt the formal developments by Launchbury and Sabry (1997) to prove type safety for monadic encapsulation of strict state.

Our goals are similar to those stated by Semmelroth and Sabry (1999), while we differ substantially in the methodology and strength of the results. We formalize the intended implementations as big-step operational semantics (which are referred to as dynamic semantics), then we prove type safety for three systems. The first system uses strict state and a newly introduced encapsulation construct based on higher-order kinds. The second system uses strict state and the original encapsulation construct *runST*: it is almost identical to the system considered by Semmelroth and Sabry (1999); the only difference being the rather minor issue of using Call-By-Name (CBN) evaluation for pure terms rather than Call-By-Value (CBV). The final system uses lazy state which adds considerable complexity since it does not allow the simple deallocation strategy typical of region-based memory management. But despite the more complicated semantics, the general approach and proof techniques used for reasoning about the previous two systems scale nicely and enable us to establish the open problem of type safety for lazy state.

The dynamic semantics we use is substantially simpler than the reduction semantics of Semmelroth and Sabry (1999), and we argue that it formalizes certain implementation details more accurately, such as deallocation of local state and *leakage* of locations referring to a deallocated state. Another advantage of the dynamic semantics we use is that it avoids the limitations one encounters when applying reduction semantics to lazy state (Launchbury & Sabry, 1997). On the other hand, reduction semantics provides a more direct support for sound equational reasoning.

Methodology and techniques. We follow a standard approach for proving type safety. The techniques used are fairly elementary and well-established:

- the dynamic behavior of programs is specified *operationally* with a structural operational semantics (SOS);
- type systems are presented à la Church (Barendregt, 1991; Cardelli, 1996), and we use *erasure* to remove information not needed at run-time;
- type safety is established for an instrumented SOS, which handles also type and region information and performs additional run-time checks, and the pattern of the proof follows (Harper, 1994).

These techniques are quite robust with respect to language extensions such as recursive definitions of terms and types, therefore we mostly ignore such desirable (but technically easy) extensions.

Summary. The article extends the paper with the same title (Moggi & Palumbo, 1999) with two results: a proof of type safety for a system with strict state and the *runST* encapsulation construct, and a proof of type safety for a system with lazy state. The first extension is an adaptation of a result by Semmelroth and Sabry (1999); the second extension is novel and resolves a problem in the paper by Launchbury and Sabry (1997).

The paper is structured as follows.

Section 2 recalls some necessary background about monads and monadic state. It also motivates the problem of monadic encapsulation and relates it to the more general problem of encapsulation of effects.

Section 3 gives a big-step operational semantics (dynamic semantics) for an untyped λ-calculus with a *run*-construct and strict state, describing the *intended implementation*, including what constitutes a run-time error. We have refrained from using a reduction semantics along the lines of several other papers (Wright & Felleisen, 1994; Launchbury & Sabry, 1997; Semmelroth & Sabry, 1999), because it fails to capture certain low-level implementations details (see Remarks 3.3 and 3.4).

Section 4 introduces a type system à la Church, basically a higher-order λ-calculus with constants. The expressiveness of the type system allows us:

- to adopt a more natural encapsulation construct, which relies on higher-order kinds, and avoids the *ad hoc* type parameter for monadic types and operations introduced in previous studies (Launchbury & Peyton Jones, 1995). Our construct can be added as a primitive to Haskell (see Appendix C).
- to establish a stronger type safety result, since more untyped terms are typable (but one must restrict to Haskell, to get a type inference algorithm).

Section 5 introduces an *instrumented semantics* for the pseudo-expressions of the type system à la Church. The instrumented semantics makes explicit the two-dimensional structure of the address space, typical of region-based memory management (Tofte & Talpin, 1997), and enables a more accurate description of *improper program behavior*. We prove type safety for the instrumented semantics, by exploiting region information in a crucial way. Then we relate the instrumented and dynamic semantics independently from well-typedness assumptions (in general the instrumented semantics does not agree with the dynamic semantics, *e.g.* the former does not permit access to the state of a thread with a location generated by another thread, while the latter semantics does). Finally, we derive type safety for the dynamic semantics, namely the *erasure* of a well-typed term cannot cause a run-time error.

Section 6 repeats the previous development for the original encapsulation construct maintaining the simpler strict state semantics. This section confirms that our methodology and proof technique are not tied to the new encapsulation construct but could also be applied to the original *runST* construct, and are thus at least as powerful as the reduction semantics approach used by Semmelroth and Sabry (1999).

Section 7 adapts the development to the more challenging problem of lazy state, showing that the current methodology and proof technique apply even to problems on which several previous approaches based on reduction semantics failed.

Section 8 briefly discusses two language extensions beyond the minimal calculi presented in the early sections.

Finally Section 9 draws some conclusions and discusses related and future work. Proof details are found in the appendices.

Acknowledgments. We thank John Launchbury for enlightening discussions. We also thank Miley Semmelroth and Fabrizio Palumbo for earlier contributions, and Steve Ganz for helpful comments. An anonymous referee, Martin Elsman, and Philip Wadler all provided valuable comments and suggestions.

Notation. We summarize some conventions used throughout.

Notation 1.1 for untyped and higher-order λ-calculi:

- An overline, *e.g.* $\overline{e}$, indicates a comma-separated sequence (of terms), and $|\overline{e}|$ denotes its length.
- We write $e(\overline{e})$ and $\lambda\overline{x}.e$ for iterated application and abstraction respectively (similar notation is also introduced for other binary constructs and binders, *e.g.* $\overline{\tau} \to \tau$, $e[\overline{u}]$, $\Lambda\overline{X}\colon K.e$ and $\forall\overline{X}\colon K.\tau$).
- Terms are treated up to α-conversion, and $e[\overline{x}\colon= \overline{e}]$ stands for parallel substitution with renaming of bound variables.
- Γ stands for a typing context, *i.e.* a sequence of variable declarations $x\colon\tau$ (and $X\colon K$); we write $\overline{x}\colon\tau$ for declaring several variables of the same type. In a well-formed context a variable can be declared at most once, while an arbitrary context can have multiple declarations of the same variable.
- Σ stands for a *signature*, *i.e.* a sequence of constant declarations $c\colon\tau$ (and $C\colon K$). Well-formed signatures will not contain multiple declarations of the same constant; we informally enforce this by requiring that any constant that is to be added to a signature is "fresh."
  A constant is like a variable that cannot be bound. With some abuse of notation (used only for constructor constants C in the instrumented semantics) one can extend operations involving variables to constants, such as substitution $e[C\colon= u]$ and binding $\Lambda C.e$ of a constant C.

Notation 1.2 related to BNF:

- We allow *production schemes* in BNF. Let $\#o$ be the arity of operation o. A scheme, *e.g.* "$o(\overline{e})$ with $|\overline{e}| \leq \#o$", stands for a finite set of productions, *i.e.* "$o(e_1, \ldots, e_n)$" with $0 \leq n \leq \#o$.
- We write $e \notin \mathit{BNF}$ to say that a certain expression e is not in the set of expressions defined by the BNF BNF. We use this notation mainly in side-conditions. For instance, the rule
  $$\frac{e_1 \Longrightarrow v}{e_1\ e_2 \Longrightarrow \mathit{err}}\ v \notin \lambda x.e \mid \mathit{run} \mid o(\overline{e}) \text{ with } |\overline{e}| < \#o$$
  says that "$e_1\ e_2$ evaluates to err", provided "e_1 evaluates to a value v" and v is not among the values defined by the BNF "$\lambda x.e \mid \mathit{run} \mid o(\overline{e})$ with $|\overline{e}| < \#o$".

2 Monadic Encapsulation: Introduction and Examples

We recall some necessary background about monads and monadic state, and then motivate the problem of monadic encapsulation and relate it to the more general problem of encapsulation of effects.

2.1 Programming with Monads

In most programming languages, evaluation may have implicit side-effects that are not predicted by the type of the expression. For example, if `r` is a global variable holding a reference to an `int`, and `Error` is a global exception name, then the Ocaml expression:

```
3 + !r + raise Error
```

has type `int`, which does not reflect the fact that the function reads the global reference and raises a global exception. In fact evaluating the expression does not return an `int` at all but rather raises the exception `Error`.

From their first introduction to the world of programming languages (Moggi, 1989; Moggi, 1991), monads were used to distinguish between *values*, whose evaluation is pure, and *computations*, whose evaluation may have side-effects. The semantic separation quickly led to a stratified programming style in which a pure functional sublanguage is used to express the manipulation of values, and a monadic sublanguage is used to express the manipulation of computations (Wadler, 1992; Wadler, 1990). The interaction between the two sublanguages is mediated by the type system, which keeps track of the computational effects and their propagation.

Hence, to write the above example in monadic style, one must expose the computational effects. This can be done in almost any modern language, but Haskell provides elegant mechanisms to do so. First, one defines a type that explicitly records the fact that computations of that type depend on global locations and may raise exceptions. For simplicity the following type assumes only one fixed global location holding an `Int`. The dependency on a global location means that computations are really functions that take the value of that location as an argument; the ability to raise exceptions means that computations may either return a value or fail, which is modeled with the `Maybe` type:

```
type Loc = Int
data GE a = GE (Loc -> Maybe a)
```

Given appropriate definitions to the operations `deref` and `raise`, our example at the beginning of the section becomes:

```
do x1 <- return 3
   x2 <- deref
   x3 <- raise "Error"
   return (x1+x2+x3)
```

The monadic expression looks more like an imperative program: sequencing and termination of monadic evaluation are made explicit through `do` and `return`. The expression no longer has type `Int` but rather has the type `GE Int` which clearly exposes the dependency on the global location and the ability to raise exceptions, and more accurately predicts the dynamic behavior of the expression.

2.2 Monadic State

To integrate assignments into Haskell, Launchbury and Peyton Jones (1995) propose to extend the Haskell runtime with a state monad. As one would expect, an efficient implementation of the monad is not possible in the source (functional) language, and hence must be provided as a primitive in the implementation. The monad comes equipped with the type constructor `ST`, and several operations for manipulating references, of which we focus on the following three: `newSTRef`, `readSTRef`, and `writeSTRef`. Using these operations, many imperative algorithms can be implemented naturally and without a loss of efficiency in Haskell. For our purposes, we are mostly concerned with the types of the operations, and their interaction with the operation `runST` discussed in the next section.

2.3 Encapsulation

One can argue that exposing the computational effects using monads has software engineering as well as semantic benefits. However, without the ability to *encapsulate* the computational effects, the monadic approach forces *every* effect to be propagated to the top-level, and even worse, interferes with the modular decomposition of programs. Hence it is desirable and even necessary to associate a construct *run* with every monad to encapsulate the computational effects.

The problem of encapsulating effects is not unique to the monadic approach: it has been studied in various contexts and is known to be quite subtle. The subtleties are most visible in the case of the state monad `ST` where the encapsulation construct is called `runST`. To gain some intuition about the problem, consider the following Haskell term:

```
runST (runST (do x <- newSTRef 0
                 return (do _ <- return x
                            return 2)))
```

Operationally, the evaluation of the expression proceeds as follows. The first `runST` creates an outer region in which its subexpression is evaluated. This subexpression immediately establishes an inner region in which a reference to 0 is allocated and bound to the name `x`, and returns a computation to be evaluated in the outer region. Since all computational effects within the inner `runST` are supposed to be encapsulated, the inner region is reclaimed at this point, so the computation:

```
do _ <- return x
   return 2
```

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

The example shows that the straightforward typing of `runST` as `ST a -> a` is unsound, since it would fail to reject terms that actually tried to use the dangling pointer.

In the history of programming languages, several approaches have been introduced to keep track of the lifetimes of references as required above, starting with Reynolds's syntactic control of interference (1978), the imperative lambda calculus (Swarup *et al.*, 1991), lambda var (Odersky *et al.*, 1993; Chen & Odersky, 1994; Rabin, 1996), type-and-effect systems (Tofte, 1990; Talpin & Jouvelot, 1992b), coercions (Riecke, 1993; Riecke & Viswanathan, 1995), and region calculi (Tofte & Talpin, 1997) to name a few. The multitude of proposals and the fact that some variants of these proposals were initially unsound, witness the difficulty of the problem. Instead of attempting to adapt any of these approaches directly to the monadic framework, Launchbury and Peyton Jones (1995) proposed to achieve the same result with a modest extension to the type system. State computations are given an additional type parameter `rho` making the type of computations `ST rho a`. The type indicates that the computation delivers a value of type `a` and that it occurs in a region indexed by the type variable `rho`. The additional type variable is propagated by every computation and stored in the types of references to keep track of the current region. In that framework, `runST` can be given the following type:

```
runST : (forall rho. ST rho a) -> a
```

which intuitively says that the effects of the state computation can be encapsulated if that computation makes no assumptions about the region in which it is evaluated. This idea has been formalized in the context of a strict state monad in which the monadic operations are performed as they are encountered. In the more complex case of the lazy state monad, the correctness of the typing of `runST` was still an open problem.

3 Dynamic Semantics: *run* with Strict State

We extend the pure untyped λ-calculus with a *run*-construct. Intuitively, when an interpreter for the λ-calculus has to evaluate *run* e, it calls a *monadic interpreter* which evaluates e applied to an *internal implementation* of the monadic operations, and then evaluates the term returned by the monadic interpreter. The term e in *run* e should be considered *abstract code*, since it abstracts from the implementation of the monadic operations. One can envisage several monadic interpreters for the same *abstract code*, which would differ in the implementation of the monadic operations.

To define the dynamic semantics for such a language, we introduce auxiliary semantic domains and extend the syntax for terms with additional constants. The relevant syntactic categories are:

- Names $m, n \in \mathsf{N}$, *e.g.* natural numbers, for *locations* ℓ_m.
- Monadic operations $o \in \mathsf{Op} \stackrel{\text{def}}{=} \{ret, do, new, get, set\}$ with arities defined as follows:

monadic operation	o	ret	do	new	get	set
arity	$\#o$	1	2	1	1	2

  Given an expression e that abstracts from the implementation of monadic operations, we write $e(\mathsf{Op})$ for applying e to the sequence of monadic operations in Op in the order listed above.
- Constants $c \in \mathsf{Const}{::}= run \mid o \mid \ell_m$.
- Terms $e \in \mathsf{E}{::}= c \mid x \mid \lambda x.e \mid e_1\ e_2$; we write E_0 for the set of closed terms.
- Values $v \in \mathsf{Val}{::}= \lambda x.e \mid run \mid \ell_m \mid o(\overline{e})$ with $|\overline{e}| \leq \#o$; we write Val_0 for the set of closed values.
- Stores $\mu \in \mathsf{S} \stackrel{\text{def}}{=} \mathsf{N} \stackrel{fin}{\to} \mathsf{E}_0$, *i.e.* partial maps from N to E_0 with finite domain; we write Loc_μ for the set $\{\ell_m | m \in dom(\mu)\}$ of locations in μ.
- Descriptions $d \in \mathsf{D}{::}= v \mid (\mu, e) \mid err$, *i.e.* possible outcomes of evaluation.

Remark 3.1 (*About constants*) *run* is the only constant allowed in user-defined programs, while monadic operations o and locations ℓ_m are instrumental to the dynamic semantics.

The dynamic semantics is given by two mutually recursive interpreters for **closed terms**, which may also raise run-time errors. There are two evaluation judgments:

- $e \Longrightarrow v \mid err$ says that evaluation of $e \in \mathsf{E}_0$ by the pure interpreter returns $v \in \mathsf{Val}_0$ (or raises an error);
- $\mu, e \Longrightarrow \mu', e' \mid err$ says that evaluation of $e \in \mathsf{E}_0$ in local store μ by the monadic interpreter returns $e' \in \mathsf{E}_0$ and a final store μ' (or raises an error).

Figure 1 gives the evaluation rules for the dynamic semantics, which satisfies the following basic property:

Proposition 3.2
$\mu, e \Longrightarrow \mu', e'$ implies $dom(\mu) \subseteq dom(\mu')$.

Remark 3.3 (*About evaluation*) On pure λ-terms pure evaluation coincides with CBN evaluation. Pure evaluation treats locations ℓ_m as values (like *nil* for the empty list) and monadic operations o as term-constructors (like *cons*). Monadic evaluation proceeds much like evaluation for an imperative language, but sequencing and termination of monadic evaluation are made explicit through *do* and *ret*. The sequencing of monadic operations is strict in the sense that monadic operations are immediately performed in the order they are encountered whether their effects are needed or not. Moreover, monadic evaluation calls pure evaluation whenever it needs the value of a term. The dynamic semantics is non-deterministic, since we do not fix a deterministic strategy for choosing an $m \notin dom(\mu)$. Finally, evaluation is rather permissive:

- locations referring to a deallocated state can be returned as values, *e.g.* $run\ (\lambda\overline{x}.x_{new}\ 0) \Longrightarrow \ell_m$ where x_{new} is the variable in $\overline{x}$ which gets bound to *new* and m can be any name;

Pure Evaluation

$$v \Longrightarrow v \qquad \frac{e_1 \Longrightarrow \lambda x.e \quad e[x:=e_2] \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

$$\frac{e_1 \Longrightarrow o(\overline{e})}{e_1\ e_2 \Longrightarrow o(\overline{e}, e_2)}\ |\overline{e}| < \#o \qquad \frac{e_1 \Longrightarrow run \quad \emptyset, e_2(\mathsf{Op}) \Longrightarrow \mu, e \quad e \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

Monadic Evaluation

$$\frac{e \Longrightarrow ret(e')}{\mu, e \Longrightarrow \mu, e'} \qquad \frac{e \Longrightarrow do(e_0, e_1) \quad \mu_0, e_0 \Longrightarrow \mu_1, e_0' \quad \mu_1, e_1\ e_0' \Longrightarrow \mu_2, e'}{\mu_0, e \Longrightarrow \mu_2, e'}$$

$$\frac{e \Longrightarrow new(e_0)}{\mu, e \Longrightarrow \mu\{m = e_0\}, \ell_m}\ m \notin dom(\mu)$$

$$\frac{e \Longrightarrow get(e_0) \quad e_0 \Longrightarrow \ell_m}{\mu, e \Longrightarrow \mu, e'}\ e' = \mu(m) \qquad \frac{e \Longrightarrow set(e_0, e_1) \quad e_0 \Longrightarrow \ell_m}{\mu, e \Longrightarrow \mu\{m = e_1\}, \ell_m}\ m \in dom(\mu)$$

Pure and Monadic Run-Time Errors

The rules for error propagation follow the ML convention, *i.e.* those for error generation are:

$$\frac{e_1 \Longrightarrow v}{e_1\ e_2 \Longrightarrow err}\ v \notin \lambda x.e \mid run \mid o(\overline{e}) \text{ with } |\overline{e}| < \#o$$

$$\frac{e \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin o(\overline{e}) \text{ with } |\overline{e}| = \#o$$

$$\frac{e \Longrightarrow get(e_0) \quad e_0 \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin \mathsf{Loc}_\mu \qquad \frac{e \Longrightarrow set(e_0, e_1) \quad e_0 \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin \mathsf{Loc}_\mu$$

Fig. 1. Evaluation Rules for Dynamic Semantics (strict state, run)

- *new* can be implemented by a local *name server*, which does not require communication with other threads, *e.g.* $run\ (\lambda\overline{x}.x_{new}\ \ell_m) \Longrightarrow \ell_n$ where n can be any name (including m);
- there is no check on whether a location generated by a thread is used to access the state of another, *e.g.* $run\ (\lambda\overline{y}.y_{do}\ (y_{new}\ 1)(\lambda\_.y_{get}\ (run\ (\lambda\overline{x}.x_{new}\ 0))))$ may evaluate to 1 or *err*. The value 1 is returned when the name servers for the two threads choose the same name, while the run-time error occurs if they choose different names.

Despite these examples, the dynamic semantics behaves properly on well-typed programs (as we will prove).

Remark 3.4 (*Untyped run vs. runST*) Previous studies (Launchbury & Peyton Jones, 1995; Launchbury & Sabry, 1997; Semmelroth & Sabry, 1999) adopt a *runST*-construct with a slightly different dynamic semantics (See Section 6):

$$\frac{e_1 \Longrightarrow runST \quad \emptyset, e_2 \Longrightarrow \mu, e \quad e \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

In an untyped language where Op can appear in user programs, the two constructs are inter-definable as follows: $run\ e \equiv runST\ (e(\mathsf{Op}))$ and $runST\ e \equiv run\ (\lambda\overline{x}.e)$ with $\overline{x}$ not free in e, but they are no longer inter-definable in the typed languages introduced in the rest of the paper.

There are important trade-offs between our dynamic semantics and the reduction semantics used previously (Launchbury & Sabry, 1997; Semmelroth & Sabry, 1999). The latter introduce an auxiliary construct $\mathsf{sto}(\mu, e)$ (or $\mathsf{sto}(\overline{\mu}, e)$), which roughly speaking corresponds to the configuration (μ, e) for our monadic interpreter. However, in $\mathsf{sto}(\mu, e)$ the locations in μ are considered bound variables (while for us they are constants), therefore one has that:

- reduction $e \longrightarrow e'$ has to be defined on open terms (and thus it is convenient to identify terms modulo α-conversion), while our dynamic semantics is given on closed terms;
- the reduction $\mathsf{sto}(\mu, ret(e)) \longrightarrow e$ makes no sense when $\mathsf{sto}(\mu, e)$ is a binder (unless no locations in μ occur in e), so one has to postpone deallocation of the local store (in a lazy state semantics there are other reasons why one must postpone deallocation);
- the reduction $\mathsf{sto}(\mu, do\ (new(e_0))\ e_1) \longrightarrow \mathsf{sto}(\mu\{m = e_0\}, e_1\ \ell_m)$ is correct only if ($m \notin dom(\mu)$ and) ℓ_m is not free in e_0, e_1, and μ, so the name server has to look at the whole term.

One can adapt the reduction semantics to provide a faithful account of store deallocation, but other implementation details (*e.g.* name generation) are at a lower level of abstraction. On the other hand, the reduction semantics is *directly related* to sound equational reasoning.

4 Higher Order Lambda-Calculus à la Church

We formalize the type system as a higher-order λ-calculus à la Church (Barendregt, 1991; Geuvers, 1993).

4.1 Syntax and Formation Rules

For convenience, we distinguish between constants (declared in signatures) and variables (declared in contexts). The type system uses the following syntactic categories:

- Constructor constants $C \in \mathsf{CONST}$ and constructor variables $X \in \mathsf{VAR}$, term constants $c \in \mathsf{Const}$ and term variables $x \in \mathsf{Var}$; these sets are assumed to be infinite and mutually disjoint.
- Kinds $K \in \mathsf{K}{::=} * \mid K_1 \to K_2$; $*$ is the kind of all types.
- Constructors $u, \tau \in \mathsf{U}{::=} C \mid X \mid \tau_1 \to \tau_2 \mid \forall X{:}K.\tau \mid \Lambda X{:}K.u \mid u_1[u_2]$; we write τ for a constructor that is expected to have kind $*$.
- Terms $e \in \mathsf{E}{::=} c \mid x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda X{:}K.e \mid e[u]$
- Signatures $\Sigma \in \mathsf{Sig}{::=} \emptyset \mid \Sigma, C{:}K \mid \Sigma, c{:}\tau$; we write $\Sigma \preceq \Sigma'$ when Σ is a prefix of Σ'.

Signatures and Contexts : $\Sigma \vdash$ and $\Sigma; \Gamma \vdash$

$$\emptyset\text{-}\Sigma \; \frac{}{\emptyset \vdash} \qquad C\text{-}\Sigma \; \frac{\Sigma \vdash}{\Sigma, C\colon K \vdash} \; C \text{ fresh in } \Sigma \qquad c\text{-}\Sigma \; \frac{\Sigma; \emptyset \vdash \tau\colon *}{\Sigma, c\colon \tau \vdash} \; c \text{ fresh in } \Sigma$$

$$\emptyset\text{-}\Gamma \; \frac{\Sigma \vdash}{\Sigma; \emptyset \vdash} \qquad X\text{-}\Gamma \; \frac{\Sigma; \Gamma \vdash}{\Sigma; \Gamma, X\colon K \vdash} \; X \text{ fresh in } \Gamma \qquad x\text{-}\Gamma \; \frac{\Sigma; \Gamma \vdash \tau\colon *}{\Sigma; \Gamma, x\colon \tau \vdash} \; x \text{ fresh in } \Gamma$$

Constructors : $\Sigma; \Gamma \vdash u\colon K$

$$C \; \frac{\Sigma; \Gamma \vdash}{\Sigma; \Gamma \vdash C\colon K} \; C\colon K \in \Sigma \qquad X \; \frac{\Sigma; \Gamma \vdash}{\Sigma; \Gamma \vdash X\colon K} \; X\colon K \in \Gamma$$

$$\Lambda \; \frac{\Sigma; \Gamma, X\colon K_1 \vdash u\colon K_2}{\Sigma; \Gamma \vdash \Lambda X\colon K_1.u\colon (K_1 \to K_2)} \qquad \text{app} \; \frac{\Sigma; \Gamma \vdash u_1\colon K_1 \to K_2 \quad \Sigma; \Gamma \vdash u_2\colon K_1}{\Sigma; \Gamma \vdash u_1[u_2]\colon K_2}$$

$$\forall \; \frac{\Sigma; \Gamma, X\colon K \vdash \tau\colon *}{\Sigma; \Gamma \vdash (\forall X\colon K.\tau)\colon *} \qquad \to \; \frac{\Sigma; \Gamma \vdash \tau_1\colon * \quad \Sigma; \Gamma \vdash \tau_2\colon *}{\Sigma; \Gamma \vdash (\tau_1 \to \tau_2)\colon *}$$

Terms : $\Sigma; \Gamma \vdash e\colon \tau$

$$c \; \frac{\Sigma; \Gamma \vdash}{\Sigma; \Gamma \vdash c\colon \tau} \; c\colon \tau \in \Sigma \quad x \; \frac{\Sigma; \Gamma \vdash}{\Sigma; \Gamma \vdash x\colon \tau} \; x\colon \tau \in \Gamma \quad \text{conv} \; \frac{\begin{array}{c}\Sigma; \Gamma \vdash e\colon \tau_1 \\ \Sigma; \Gamma \vdash \tau_2\colon *\end{array}}{\Sigma; \Gamma \vdash e\colon \tau_2} \; \tau_1 =^u_{\beta\eta} \tau_2$$

$$\to\text{I} \; \frac{\Sigma; \Gamma, x\colon \tau_1 \vdash e\colon \tau_2}{\Sigma; \Gamma \vdash \lambda x\colon \tau_1.e\colon (\tau_1 \to \tau_2)} \qquad \to\text{E} \; \frac{\Sigma; \Gamma \vdash e_1\colon (\tau_1 \to \tau_2) \quad \Sigma; \Gamma \vdash e_2\colon \tau_1}{\Sigma; \Gamma \vdash e_1 \; e_2\colon \tau_2}$$

$$\forall\text{I} \; \frac{\Sigma; \Gamma, X\colon K \vdash e\colon \tau}{\Sigma; \Gamma \vdash \Lambda X\colon K.e\colon (\forall X\colon K.\tau)} \qquad \forall\text{E} \; \frac{\Sigma; \Gamma \vdash e\colon (\forall X\colon K.\tau) \quad \Sigma; \Gamma \vdash u\colon K}{\Sigma; \Gamma \vdash e[u]\colon \tau[X\colon= u]}$$

Fig. 2. Formation Rules for Type System

- Contexts $\Delta, \Gamma \in \mathsf{Ctx}\colon\colon= \emptyset \mid \Gamma, X\colon K \mid \Gamma, x\colon \tau$.

Notation 4.1 There are several notions of reduction one may consider:

- $(\Lambda X\colon K.u')[u] \longrightarrow^u_\beta u'[X\colon= u]$ and $\Lambda X\colon K.u[X] \longrightarrow^u_\eta u$ when $X \notin \mathrm{FV}(u)$
- $(\Lambda X\colon K.e)[u] \longrightarrow^\forall_\beta e[X\colon= u]$ and $\Lambda X\colon K.e[X] \longrightarrow^\forall_\eta e$ when $X \notin \mathrm{FV}(e)$
- $(\lambda x\colon \tau.e') \; e \longrightarrow^e_\beta e'[x\colon= e]$ and $\lambda x\colon \tau.e \; x \longrightarrow^e_\eta e$ when $x \notin \mathrm{FV}(e)$

The only notion of reduction needed for defining the type system is $\longrightarrow^u_{\beta\eta}$, *i.e.* the union of $\longrightarrow^u_\beta$ and $\longrightarrow^u_\eta$. With some abuse of notation, we use the same notation to refer to notions of reductions and their *compatible closure*. Moreover, we denote with $=^u_{\beta\eta}$ the reflexive, symmetric and transitive closure of the reduction $\longrightarrow^u_{\beta\eta}$ (and similarly for other notions of reduction).

Figure 2 gives the rules of the type system for deriving judgments of the form:

- $\Sigma \vdash$, *i.e.* Σ is a well-formed signature
- $\Sigma; \Gamma \vdash$, *i.e.* Γ is a well-formed context
- $\Sigma; \Gamma \vdash u\colon K$, *i.e.* u is a well-formed constructor of kind K
- $\Sigma; \Gamma \vdash e\colon \tau$, *i.e.* e is a well-formed term of type τ.

Δ	$\emptyset$	$X{:}K',\ \Delta'$	$x{:}\tau',\ \Delta'$
ρ	$\emptyset$	$u',\ \rho'$	$e',\ \rho'$

Derived notation for:

kinds	$\Delta \to K$	K	$K' \to \Delta' \to K$	$\Delta' \to K$
constructors	$\Lambda\Delta.u$	u	$\Lambda X{:}K'.\Lambda\Delta'.u$	$\Lambda\Delta'.u$
	$u(\rho)$	u	$u[u'](\rho')$	$u(\rho')$
	$\forall\Delta.\tau$	τ	$\forall X{:}K'.\forall\Delta'.\tau$	$\tau' \to \forall\Delta'.\tau$
terms	$\Lambda\Delta.e$	e	$\Lambda X{:}K'.\Lambda\Delta'.e$	$\lambda x{:}\tau'.\Lambda\Delta'.e$
	$e(\rho)$	e	$(e[u'])(\rho')$	$(e\ e')(\rho')$
signatures and contexts	$\Pi\Gamma.\Delta$	$\emptyset$	$X{:}\Gamma \to K',\ \Pi\Gamma.\Delta'[X{:=}X(\lvert\Gamma\rvert)]$	$x{:}\forall\Gamma.\tau',\ \Pi\Gamma.\Delta'$
sequences	$\lvert\Delta\rvert$	$\emptyset$	$X,\ \lvert\Delta'\rvert$	$x,\ \lvert\Delta'\rvert$
	$\lambda\Gamma.\rho$	$\emptyset$	$\Lambda\Gamma.u',\ \lambda\Gamma.\rho'$	$\Lambda\Gamma.e',\ \lambda\Gamma.\rho'$
	$\rho(\theta)$	$\emptyset$	$u'(\theta),\ \rho'(\theta)$	$e'(\theta),\ \rho'(\theta)$

The two top lines give the three cases of the inductive definitions of Δ and ρ, while the others introduce notation defined by induction on the structure of Δ or ρ. For instance, the first line in the second table defines $\Delta \to K$ (first entry) by cases on the inductive definition of Δ, *i.e.*

$$\emptyset \to K \stackrel{\text{def}}{\equiv} K, (X{:}K',\ \Delta') \to K \stackrel{\text{def}}{\equiv} K' \to \Delta' \to K \text{ and } (x{:}\tau',\ \Delta') \to K \stackrel{\text{def}}{\equiv} \Delta' \to K.$$

Fig. 3. Derived Notation

Appendix A summarizes some basic facts about the type system needed for later developments in the paper.

4.2 Types for encapsulation

This section describes the type for *run*, which is the only constant allowed in user-defined programs, and relates it to the type of the original *runST* encapsulation construct (Launchbury & Peyton Jones, 1995) and to existential types (Mitchell & Plotkin, 1988). We argue that the type of *run* is intuitive: it simply maps monadic code to values. The type of *run* is however a new point in the design space of type-based encapsulation mechanisms; it differs from abstract data types, existential types, and the *runST* proposal.

For conciseness, we use the derived notation in Figure 3 defined by induction on the structure of a context Δ or a sequence ρ, where sequences are given by the BNF $\rho, \theta \in \mathsf{Seq}{::=}\ \emptyset \mid u,\ \rho \mid e,\ \rho$.

Type of run. The signature Σ_{run} for the constant *run* is:

$$run{:}\forall X{:}*.(\forall\Gamma_M.X_M[X]) \to X \ \text{, where}$$

$$\begin{array}{lll}\Gamma_M & \equiv & X_M, X_R{:}\,* \to *,\\ & & x_{ret}{:}\,\forall X{:}\,*.X \to X_M[X],\\ & & x_{do}{:}\,\forall X, Y{:}\,*.X_M[X] \to (X \to X_M[Y]) \to X_M[Y],\\ & & x_{new}{:}\,\forall X{:}\,*.X \to X_M[X_R[X]],\\ & & x_{get}{:}\,\forall X{:}\,*.X_R[X] \to X_M[X],\\ & & x_{set}{:}\,\forall X{:}\,*.X_R[X] \to X \to X_M[X_R[X]]\end{array}$$

A more appealing way of writing the type for *run* is $\forall X{:}\,*.(M[X]) \to X$, where the type constructor M is defined as $M \equiv \Lambda X{:}\,*.\forall\Gamma_M.X_M[X] \quad : * \to *$. Intuitively $M[X]$ is the type of *monadic code* (in higher-order abstract syntax).

One can almost define an *initial algebra* for the *specification* Γ_M. In second-order λ-calculus one can *represent* initial algebras for algebraic specifications. For instance, given the specification $\Gamma_N \equiv X_N{:}\,*,\ x_{zero}{:}\,X_N,\ x_{succ}{:}\,X_N \to X_N$ of the natural numbers, one can define the type $N \equiv \forall\Gamma_N.X_N$ (of Church's numerals), which has the structure of a *weakly* initial algebra (Reynolds & Plotkin, 1993). The specification Γ_M is not algebraic, but one can mimic the definition of the initial algebra given by Reynolds and Plotkin (1993), except for the operation *new* (and *set*), since the type $\forall X{:}\,*.X \to M[R[X]]$ of *new* has a nesting of M and R.

$$\begin{array}{rcl}M \equiv & \Lambda X{:}\,*.\forall\Gamma_M.X_M[X] & : * \to *\\ R \equiv & \Lambda X{:}\,*.\forall\Gamma_M.X_R[X] & : * \to *\\ ret \equiv & \Lambda X{:}\,*.\lambda x{:}\,X.\Lambda\Gamma_M.x_{ret}[X](x) & : \forall X{:}\,*.X \to M[X]\\ new \equiv & ???? & : \forall X{:}\,*.X \to M[R[X]]\\ get \equiv & \Lambda X{:}\,*.\lambda x{:}\,R[X].\Lambda\Gamma_M.x_{get}[X]\ (x(|\Gamma_M|)) & : \forall X{:}\,*.R[X] \to M[X]\end{array}$$

Comparison with runST. It is easy to recast the original proposal of encapsulation (Launchbury & Peyton Jones, 1995) in the higher-order λ-calculus, and thus compare its expressiveness with that of our *run*. The type system of Launchbury and Peyton Jones (1995) introduces several constants. If we write Σ_M for Γ_M viewed as a signature (we write M in place of X_M, etc.), then these constants are those declared in:

$$\Sigma'_M,\ runST{:}\,\forall X{:}\,*.(\forall\alpha{:}\,*.M[\alpha, X]) \to X \quad \text{where } \Sigma'_M \overset{\text{def}}{\equiv} \Pi\alpha{:}\,*.\Sigma_M$$

(every constant in Σ'_M takes an extra type parameter with respect to the corresponding constant in Σ_M). In the higher-order λ-calculus one can define our *run* in terms of these constants:

$$run \overset{\text{def}}{\equiv} \Lambda X{:}\,*.\ \lambda x{:}\,(\forall\Gamma_M.X_M[X]).\ runST[X]\ (\Lambda\alpha{:}\,*.x\ (|\Sigma'_M|\ (\alpha)))$$

In other words, $run[X]\ x$ first specializes the monadic code x with the constants in Σ'_M applied to a generic type parameter α, *i.e.*

$$\Sigma'_M;\ X{:}\,*,\ x{:}\,\forall\Gamma_M.X_M[X],\ \alpha{:}\,* \vdash x\ (|\Sigma'_M|\ (\alpha)){:}\,M[\alpha, X]\ ,$$

then applies *runST* to the abstraction of the specialized code with respect to α. As shown in Appendix C, this construction can be easily implemented in Haskell (using the non-standard extensions for rank-2 polymorphism).

Remark 4.2 (*Typed run vs. runST*) We conjecture that $runST$ (and the other constants in Σ'_M) cannot be defined in terms of run. We advocate run in place of $runST$ because it avoids the *ad hoc* type parameter α, and thus it complies with standard monadic programming style. Moreover, we have given an intuitive reading for the type of run in terms of the definable type constructor for *monadic code*.

Comparison with existential types. With the notation of Figure 3 one can define the *existential type* $\exists\Delta$ as $\forall X{:}*.(\forall\Delta.X) \to X$. Then the definition $\forall X{:}*.(\forall\Gamma_M.X) \to X$ of the existential type $\exists\Gamma_M$ is similar to the type $\forall X{:}*.(\forall\Gamma_M.X_M[X]) \to X$ of run.

5 Instrumented Semantics: *run* with Strict State

The instrumented semantics is a refinement of the dynamic semantics of Section 3 which uses terms of the type system à la Church described in Section 4 instead of untyped λ-terms. The instrumented semantics serves two technical purposes: to give a more accurate description of *improper program behavior*, and to prove type safety for the dynamic semantics. In order to transfer the type safety result from the instrumented to the dynamic semantics, one has to establish a *compatibility* result linking dynamic and instrumented semantics.

To define the instrumented semantics, we introduce auxiliary semantic domains and syntactic categories. Most of them are analogues of those introduced for the dynamic semantics, and are indicated by a superscript *:

- Names $r \in \mathsf{N}$ for *regions*. Region names are used to index locations $\ell_{r,m}$, the monadic type-constructors O_r, and the monadic operations o_r.
- Names $m, n \in \mathsf{N}$ for *locations* $\ell_{r,m}$ within a region.
- Names $a \in \mathsf{N}$ for constructor constants C_a created by evaluation of polymorphic terms (see Remark 5.1).
- Monadic type-constructors $O \in \mathsf{OP} \stackrel{\text{def}}{=} \{M, R\}$.
- Monadic operations $o \in \mathsf{Op} \stackrel{\text{def}}{=} \{ret, do, new, get, set\}$ with the following arities:

monadic operation	o	ret	do	new	get	set
type-arity	$\Box o$	1	2	1	1	1
term-arity	$\#o$	1	2	1	1	2

- Type- and term-constants, kinds, constructors and terms:

$$
\begin{array}{rcl}
C & ::= & C_a \mid O_r \text{ ; we write } \mathsf{OP}_r \text{ for } \{O_r | O \in \mathsf{OP}\} \\
c \in \mathsf{Const}^\star & ::= & run \mid o_r \mid \ell_{r,m} \text{ ; we write } \mathsf{Op}_r \text{ for } \{o_r | o \in \mathsf{Op}\} \\
K & ::= & * \mid K_1 \to K_2 \\
u & ::= & C \mid X \mid u_1 \to u_2 \mid \forall X{:}K.u \mid \Lambda X{:}K.u \mid u_1[u_2] \\
e \in \mathsf{E}^\star & ::= & c \mid x \mid \lambda x{:}u.e \mid e_1\ e_2 \mid \Lambda X{:}K.e \mid e[u]
\end{array}
$$

- Values: $v \in \mathsf{Val}^\star \ ::= \ \lambda x{:}u.e \mid run \mid run[u] \mid \ell_{r,m} \mid o_r[\overline{u}] \text{ with } |\overline{u}| < \Box o$
  $\mid \ o_r[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \text{ and } |\overline{e}| \le \#o$
- Stores $\mu \in \mathsf{S}^\star \stackrel{\text{def}}{=} \mathsf{N} \stackrel{fin}{\to} \mathsf{E}^\star_0$; we write $\mathsf{Loc}_{r,\mu}$ for the set $\{\ell_{r,m} | m \in dom(\mu)\}$.

- Dynamic signatures $\Sigma \in \mathsf{Sig}::= \emptyset \mid \Sigma, C\colon K \mid \Sigma, c\colon u$ and
  Static signatures $\Delta::= \emptyset \mid \Delta, C_a\colon K$.
  Dynamic signatures keep track of the monadic type-constructors, operations, and locations associated with each region. Static signatures keep track of the polymorphism of terms. (See Remark 5.1.)
- Descriptions $d \in \mathsf{D}^\star::= (\Delta|\Sigma; v) \mid (\Sigma; \mu, e) \mid err$.

Remark 5.1 (*Polymorphism*) The values in $\mathsf{Val}^\star$ do not include polymorphic terms $\Lambda X\colon K.e$, and hence the instrumented semantics has to account for evaluation under Λ. More specifically, to evaluate $\Lambda X\colon K.e$ we replace X with a fresh type constant C_a, and evaluate $e[X:= C_a]$. These type constants C_a and their kinds form the static signatures Δ.

One can obtain a term in E from a term in $\mathsf{E}^\star$ by *erasing* types and regions.

Definition 5.2
Given $e \in \mathsf{E}^\star$ its **erasure** $|e| \in \mathsf{E}$ is defined by induction as

$$|run| = run \quad |o_r| = o \quad |\ell_{r,m}| = \ell_m \quad |x| = x$$
$$|\lambda x\colon u.e| = \lambda x.|e| \quad |e_1\ e_2| = |e_1|\ |e_2| \quad |\Lambda X\colon K.e| = |e[u]| = |e|$$

Erasure is extended to stores $\mu \in \mathsf{S}^\star$ and descriptions $d \in \mathsf{D}^\star$ as follows

$$|\mu|(m) = |\mu(m)| \qquad |(\Delta|\Sigma; v)| = |v| \qquad |(\Sigma; \mu, e)| = (|\mu|, |e|) \qquad |err| = err$$

Proposition 5.3
Erasure satisfies the following properties:

- $|e[X:= u]| = |e|$ and $|e[x:= e']| = |e|[x:= |e'|]$
- $v \in \mathsf{Val}^\star$ implies $|v| \in \mathsf{Val}$
- $e \longrightarrow^{u\forall}_{\beta\eta} e'$ implies $|e| \equiv |e'|$.

The instrumented semantics (like the dynamic one) is given in terms of two mutually recursive interpreters, which evaluate **closed terms** and may raise run-time errors:

- $\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v \mid err$ says that evaluation of $e \in \mathsf{E}^\star_0$ by the pure interpreter returns $v \in \mathsf{Val}^\star_0$ and extends the dynamic signature from Σ to Σ' (or raises an error); moreover the polymorphism of e *is made explicit* by extending the static signature Δ to Δ';
- $\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} \Sigma'; \mu', e' \mid err$ says that evaluation of $e \in \mathsf{E}^\star_0$ in local store μ by the monadic interpreter for region r returns $e' \in \mathsf{E}^\star_0$, changes the store to μ' and extends the dynamic signature from Σ to Σ' (or raises an error).

Figures 4 and 5 give the evaluation rules for the instrumented semantics. Each rule is either the counterpart of a rule for the dynamic semantics of Figure 1, or is for evaluating terms of the form $e[u]$ and $\Lambda X\colon K.e$. The dynamic signature Σ keeps track of the monadic constants and their regions. The rule for evaluating $\Lambda X\colon K.e$ is the one forcing the introduction of static signatures Δ and Δ'. The pure interpreter

Pure Evaluation

$$\Delta|\Sigma; v \Longrightarrow \emptyset|\Sigma; v \qquad \frac{\Delta|\Sigma_0; e_1 \Longrightarrow \emptyset|\Sigma_1; \lambda x\colon u.e \quad \Delta|\Sigma_1; e[x\colon= e_2] \Longrightarrow \Delta'|\Sigma_2; v}{\Delta|\Sigma_0; e_1\ e_2 \Longrightarrow \Delta'|\Sigma_2; v}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; run}{\Delta|\Sigma; e[u] \Longrightarrow \emptyset|\Sigma'; run[u]} \qquad \frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}]}{\Delta|\Sigma; e[u] \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}, u]}\ |\overline{u}| < \Box o$$

$$\frac{\Delta|\Sigma; e_1 \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}](\overline{e})}{\Delta|\Sigma; e_1\ e_2 \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}](\overline{e}, e_2)}\ |\overline{u}| = \Box o \wedge |\overline{e}| < \# o$$

$$\frac{\Delta|\Sigma; e \Longrightarrow C_a\colon K, \Delta'|\Sigma'; v}{\Delta|\Sigma; e[u] \Longrightarrow \Delta'|(\Sigma'; v)[C_a\colon= u]}$$

$$\frac{\Delta, C_a\colon K|\Sigma; e[X\colon= C_a] \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; \Lambda X\colon K.e \Longrightarrow C_a\colon K, \Delta'|\Sigma'; v}\ a \text{ fresh}$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0; e_1 \Longrightarrow \emptyset|\Sigma_1; run[u] \\ \Delta|\Sigma_1 + \Sigma_r^{op}; \emptyset, e_2[\mathsf{OP}_r](\mathsf{Op}_r) \stackrel{r}{\Longrightarrow} \Sigma_2; \mu, e \\ \Delta|\Sigma_2; e \Longrightarrow \Delta'|\Sigma_3; v\end{array}}{\Delta|\Sigma_0; e_1\ e_2 \Longrightarrow \Delta'|\Sigma_3; v}\ r \text{ fresh}$$

where $\Sigma_r^{op} \stackrel{\text{def}}{\equiv} M_r,\ R_r\colon * \to *\ ,$
$ret_r\colon \forall X\colon *.X \to M_r[X]\ ,$
$do_r\colon \forall X, Y\colon *.M_r[X], (X \to M_r[Y]) \to M_r[Y]\ ,$
$new_r\colon \forall X\colon *.X \to M_r[R_r[X]]\ ,$
$get_r\colon \forall X\colon *.R_r[X] \to M_r[X]\ ,$
$set_r\colon \forall X\colon *.R_r[X], X \to M_r[R_r[X]]$

Monadic Evaluation

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; ret_r[u](e')}{\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma'; \mu, e'} \qquad \frac{\begin{array}{l}\Delta|\Sigma_0; e \Longrightarrow \emptyset|\Sigma_1; do_r[u_0, u_1](e_0, e_1) \\ \Delta|\Sigma_1; \mu_0, e_0 \stackrel{r}{\Longrightarrow} \Sigma_2; \mu_1, e_0' \\ \Delta|\Sigma_2; \mu_1, e_1\ e_0' \stackrel{r}{\Longrightarrow} \Sigma_3; \mu_2, e'\end{array}}{\Delta|\Sigma_0; \mu_0, e \stackrel{r}{\Longrightarrow} \Sigma_3; \mu_2, e'}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; new_r[u](e_0)}{\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma', \ell_{r,m}\colon R_r u; \mu\{m = e_0\}, \ell_{r,m}}\ m \notin dom(\mu)$$

$$\frac{\Delta|\Sigma_0; e \Longrightarrow \emptyset|\Sigma_1; get_r[u](e_0) \quad \Delta|\Sigma_1; e_0 \Longrightarrow \emptyset|\Sigma_2; \ell_{r,m}}{\Delta|\Sigma_0; \mu, e \stackrel{r}{\Longrightarrow} \Sigma_2; \mu, e'}\ e' = \mu(m)$$

$$\frac{\Delta|\Sigma_0; e \Longrightarrow \emptyset|\Sigma_1; set_r[u](e_0, e_1) \quad \Delta|\Sigma_1; e_0 \Longrightarrow \emptyset|\Sigma_2; \ell_{r,m}}{\Delta|\Sigma_0; \mu, e \stackrel{r}{\Longrightarrow} \Sigma_2; \mu\{m = e_1\}, \ell_{r,m}}\ m \in dom(\mu)$$

Fig. 4. Evaluation Rules for Instrumented Semantics I (strict state, run)

may evaluate under Λ but the monadic interpreter never evaluates under Λ which avoids known problems with the combination of polymorphism and effects (Harper & Lillibridge, 1993).

The following are basic properties of the instrumented semantics.

Proposition 5.4

Pure and Monadic Run-Time Errors

The rules for error propagation follow the ML convention, those for error generation are:

$$\frac{\Delta|\Sigma; e_1 \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; e_1\ e_2 \Longrightarrow err} \quad \begin{array}{l}\Delta' \not\equiv \emptyset \text{ or} \\ v \notin \lambda x\!: u.e \mid run[u] \mid o_r[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| < \#o\end{array}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; v}{\Delta|\Sigma; e[u] \Longrightarrow err}\ v \notin run \mid o_r[\overline{u}] \text{ with } |\overline{u}| < \Box o$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} err}\ \Delta' \not\equiv \emptyset \vee v \notin o_r[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| = \#o$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; get_r[u](e_0) \quad \Delta|\Sigma'; e_0 \Longrightarrow \Delta'|\Sigma''; v}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} err}\ \Delta' \not\equiv \emptyset \vee v \notin \mathsf{Loc}_{r,\mu}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; set_r[u](e_0, e_1) \quad \Delta|\Sigma'; e_0 \Longrightarrow \Delta'|\Sigma''; v}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} err}\ \Delta' \not\equiv \emptyset \vee v \notin \mathsf{Loc}_{r,\mu}$$

Fig. 5. Evaluation Rules for Instrumented Semantics II (strict state, run)

- $\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v$ implies $\Sigma \preceq \Sigma'$
- $\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} \Sigma'; \mu', e'$ implies $\Sigma \preceq \Sigma'$ and $dom(\mu) \subseteq dom(\mu')$
- $\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v$ implies no C_a in Δ' occurs in $\Delta|\Sigma$.

The last clause is needed in the rule for evaluating polymorphic abstractions to conclude that Σ is identical to $\Sigma[C_a\!:= u]$.

5.1 Type safety and compatibility

We show that well-formed programs cannot go wrong, which amounts to proving subject reduction for the instrumented semantics.

Notation 5.5 We introduce auxiliary definitions useful in stating type safety.

- $\Delta, \Sigma \models J \overset{\text{def}}{\Longleftrightarrow} \Sigma_{run}, \Delta, \Sigma \vdash J$ and for all $C\!: K$ and $c\!: \tau$ declared in Σ:
  - — $C \equiv M_r$ implies $K \equiv * \to *$
  - — $C \equiv R_r$ implies $K \equiv * \to *$
  - — $c \equiv ret_r$ implies $\tau \equiv \forall X\!:*.X \to M_r[X]$
  - — $c \equiv do_r$ implies $\tau \equiv \forall X, Y\!:*.M_r[X], (X \to M_r[Y]) \to M_r[Y]$
  - — $c \equiv new_r$ implies $\tau \equiv \forall X\!:*.X \to M_r[R_r[X]]$
  - — $c \equiv get_r$ implies $\tau \equiv \forall X\!:*.R_r[X] \to M_r[X]$
  - — $c \equiv set_r$ implies $\tau \equiv \forall X\!:*.R_r[X], X \to M_r[R_r[X]]$
  - — $c \equiv \ell_{r,m}$ implies $\tau \equiv R_r[\tau']$ for some τ'
- Reg_Σ is the set of regions in Σ, *i.e.*
  $r \in \mathsf{Reg}_\Sigma \overset{\text{def}}{\Longleftrightarrow}$ at least one O_r is declared in Σ.
- $\mathsf{Loc}_{r,\Sigma}$ is the set of locations of region r in Σ, *i.e.*
  $\ell_{r,m} \in \mathsf{Loc}_{r,\Sigma} \overset{\text{def}}{\Longleftrightarrow} \ell_{r,m}$ is declared in Σ.

- $\Sigma \hookrightarrow \Sigma' \stackrel{\mathrm{def}}{\iff} \Sigma \preceq \Sigma'$ and $\forall n \in \mathsf{Reg}_\Sigma.\mathsf{Loc}_{n,\Sigma} = \mathsf{Loc}_{n,\Sigma'}$, *i.e.* Σ' extends Σ but the set of locations in pre-existing regions does not change.
- $\Sigma \stackrel{r}{\hookrightarrow} \Sigma' \stackrel{\mathrm{def}}{\iff} \Sigma \preceq \Sigma'$ and $\forall n \in \mathsf{Reg}_\Sigma - \{r\}.\mathsf{Loc}_{n,\Sigma} = \mathsf{Loc}_{n,\Sigma'}$, *i.e.* Σ' extends Σ but the set of locations in pre-existing regions, except region r, does not change.
- $\Delta, \Sigma \models_r \mu \stackrel{\mathrm{def}}{\iff} \Delta, \Sigma \models$ and $\mathsf{Loc}_{r,\mu} = \mathsf{Loc}_{r,\Sigma}$ and for all names $m \in \mathsf{N}$ $\ell_{r,m}\colon R_r[\tau]$ in Σ and $e \equiv \mu(m)$ imply $\Delta, \Sigma \models e\colon\tau$

To establish type safety one has to prove a much stronger result, which involves in an essential way regions, namely pure evaluation does not add new locations to pre-existing regions, while monadic evaluation can add new locations to the *active region* r, but not to other pre-existing regions. However, both evaluations may add new locations to newly generated regions.

Theorem 5.6 (Type Safety for Instrumented Semantics)

1. If $\Delta|\Sigma; e \Longrightarrow d$ and $\Delta, \Sigma \models e\colon\tau$, then exist Δ', Σ', v and τ' *s.t.* $d \equiv (\Delta'|\Sigma'; v)$, $\tau =^u_{\beta\eta} \forall\Delta'.\tau'$, $\Sigma \hookrightarrow \Sigma'$ and $\Delta, \Delta', \Sigma' \models v\colon\tau'$.
   The type $\forall\Delta'.\tau'$ is defined by induction on Δ' (see Figure 3), and by blurring the distinction between constants and variables (see Notation 1.1).
2. If $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} d$, $\Delta, \Sigma \models_r \mu$ and $\Delta, \Sigma \models e\colon M_r[\tau]$, then exist Σ', μ' and e' *s.t.* $d \equiv (\Sigma'; \mu', e')$, $\Sigma \stackrel{r}{\hookrightarrow} \Sigma'$, $\Delta, \Sigma' \models_r \mu'$ and $\Delta, \Sigma' \models e'\colon\tau$.

Proof

By induction on the derivation of an evaluation judgment, and by applying the generation lemma to the typing assumption. The details for some cases are given in Appendix B. $\square$

One expects the following relation (mediated by erasure) between the instrumented and dynamic semantics: if e may evaluate to v, then $|e|$ evaluates to $|v|$.

Theorem 5.7 (Erasure)

- $\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma; v$ implies $|e| \Longrightarrow |v|$
- $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma'; \mu', e'$ implies $|\mu|, |e| \stackrel{r}{\Longrightarrow} |\mu'|, |e'|$

When e may raise an error, one cannot say anything about $|e|$. To prove type safety for the dynamic semantics, the above result is useless. Instead we need implications in the opposite direction covering also the case of run-time error (in the dynamic semantics), more precisely: if $|e|$ may raise an error so does e, if $|e|$ may evaluate to v', then e may evaluate to a *compatible* value or raise an error. This compatibility result together with type safety for the instrumented semantics will immediately imply type safety for the dynamic semantics via Corollary 5.9.

Theorem 5.8 (Compatibility)

For every Δ, Σ, e, μ, r and d' the following implications hold:

- $|e| \Longrightarrow d'$ implies exists d *s.t.* $\Delta|\Sigma; e \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv \mathit{err}$)
- $|\mu|, |e| \Longrightarrow d'$ implies exists d *s.t.* $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} d$ and ($d' \equiv |d|$ or $d \equiv \mathit{err}$)

Proof

The implications are proved by lexicographic induction on the derivation of an evaluation judgment $|e| \Longrightarrow d'$ and $|\mu|, |e| \Longrightarrow d'$ for the dynamic semantics, and the *size* of e and (μ, e). The details for some cases are given in Appendix B. $\square$

Corollary 5.9 (Type Safety for Dynamic Semantics)
If $\Sigma_{run}; \emptyset \vdash e\colon\tau$ and $|e| \Longrightarrow d'$, then $d' \not\equiv err$.

Proof

The typing assumption on e is equivalent to $\emptyset \models e\colon\tau$. By compatibility we know there exists a d *s.t.* $\emptyset|\emptyset; e \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$). By type safety for the instrumented semantics we know that $d \not\equiv err$, therefore $d' \equiv |d| \not\equiv err$. $\square$

6 *runST* with Strict State

We explain how to adapt the development of Sections 3 to 5 to establish type safety for a language with a strict state variant of the original state monad consisting of *runST*, OP, and Op.

6.1 Dynamic Semantics

The only change in the syntactic categories for the untyped language is in the names of constants, the evaluation judgments are unchanged, and the evaluation rules (see Figure 6) are unchanged, except the rule for *run*, which has been replaced by the rule for *runST*.

- Names $m, n \in \mathsf{N}$ unchanged.
- Monadic operations $o \in \mathsf{Op} \stackrel{\text{def}}{=} \{retST, doST, newST, getST, setST\}$ with the following arities:

monadic operation	o	*retST*	*doST*	*newST*	*getST*	*setST*
arity	$\#o$	1	2	1	1	2

- Constants $c \in \mathsf{Const}{::=}\ runST \mid o \mid \ell_m$.
- Terms $e \in \mathsf{E}{::=}\ c \mid x \mid \lambda x.e \mid e_1\ e_2$ unchanged.
- Values $v \in \mathsf{Val}{::=}\ \lambda x.e \mid runST \mid \ell_m \mid o(\overline{e})$ with $|\overline{e}| \leq \#o$.
- Stores $\mu \in \mathsf{S}$ unchanged.
- Descriptions $d \in \mathsf{D}$ unchanged.

Remark 6.1 (*On constants*) In this language *runST* and the monadic operations o are allowed in user-defined programs, but locations ℓ_m are instrumental to the dynamic semantics.

6.2 Types for encapsulation

This section gives the signature Σ_{runST} for the constants allowed in user-defined programs, *i.e.* *runST* and the monadic type-constructors and operations. The kinds

Pure Evaluation

$$\frac{e_1 \Longrightarrow runST \quad \emptyset, e_2 \Longrightarrow \mu, e \quad e \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

Monadic Evaluation

$$\frac{e \Longrightarrow retST(e')}{\mu, e \Longrightarrow \mu, e'} \qquad \frac{e \Longrightarrow doST(e_0, e_1) \quad \mu_0, e_0 \Longrightarrow \mu_1, e_0' \quad \mu_1, e_1\ e_0' \Longrightarrow \mu_2, e'}{\mu_0, e \Longrightarrow \mu_2, e'}$$

$$\frac{e \Longrightarrow newST(e_0)}{\mu, e \Longrightarrow \mu\{m = e_0\}, \ell_m}\ m \notin dom(\mu)$$

$$\frac{e \Longrightarrow getST(e_0) \quad e_0 \Longrightarrow \ell_m}{\mu, e \Longrightarrow \mu, e'}\ e' = \mu(m)$$

$$\frac{e \Longrightarrow setST(e_0, e_1) \quad e_0 \Longrightarrow \ell_m}{\mu, e \Longrightarrow \mu\{m = e_1\}, \ell_m}\ m \in dom(\mu)$$

Pure and Monadic Run-Time Errors

The rules for error propagation follow the ML convention; those for error generation are:

$$\frac{e_1 \Longrightarrow v}{e_1\ e_2 \Longrightarrow err}\ v \notin \lambda x.e \mid runST \mid o(\overline{e}) \text{ with } |\overline{e}| < \#o$$

$$\frac{e \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin o(\overline{e}) \text{ with } |\overline{e}| = \#o$$

$$\frac{e \Longrightarrow getST(e_0) \quad e_0 \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin \mathsf{Loc}_\mu \qquad \frac{e \Longrightarrow setST(e_0, e_1) \quad e_0 \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin \mathsf{Loc}_\mu$$

Fig. 6. Evaluation Rules for Dynamic Semantics (strict state, runST)

and types of constants are those given by Launchbury and Peyton Jones (1995).

$$\begin{aligned}
&\mathrm{ST}, \mathrm{Ref}{:}\, * \to * \to *\ ,\\
&retST{:}\,\forall \alpha, X{:}\,*.X \to \mathrm{ST}[\alpha, X]\ ,\\
&doST{:}\,\forall \alpha, X, Y{:}\,*.\mathrm{ST}[\alpha, X] \to (X \to \mathrm{ST}[\alpha, Y]) \to \mathrm{ST}[\alpha, Y]\ ,\\
&newST{:}\,\forall \alpha, X{:}\,*.X \to \mathrm{ST}[\alpha, \mathrm{Ref}[\alpha, X]]\ ,\\
&getST{:}\,\forall \alpha, X{:}\,*.\mathrm{Ref}[\alpha, X] \to \mathrm{ST}[\alpha, X]\ ,\\
&setST{:}\,\forall \alpha, X{:}\,*.\mathrm{Ref}[\alpha, X] \to X \to \mathrm{ST}[\alpha, \mathrm{Ref}[\alpha, X]]\ ,\\
&runST{:}\,\forall X{:}\,*.(\forall \alpha{:}\,*.\mathrm{ST}[\alpha, X]) \to X
\end{aligned}$$

6.3 Instrumented semantics

In comparison to the language of Section 5 the main change involves the constants, in particular: monadic type-constructors and operations no longer have region annotations, instead they take an extra type parameter, since region information is now encoded by type-constants V_r; $runST$ expects code abstracted over a region (encoded as a type), instead of code abstracted with respect to the monadic operations.

- Names $r \in \mathsf{N}$ for the *type encoding* V_r of a region. Region names are used to index locations $\ell_{r,m}$.

- Names $m, n \in \mathsf{N}$ for locations $\ell_{r,m}$ within a region.
- Names $a \in \mathsf{N}$ for constructor constants C_a created by evaluation of polymorphic terms.
- Monadic type-constructors $O \in \mathsf{OP} \stackrel{\text{def}}{=} \{\mathrm{ST}, \mathrm{Ref}\}$ for describing the types of monadic computations and references.
- Monadic operations $o \in \mathsf{Op} \stackrel{\text{def}}{=} \{\mathit{retST}, \mathit{doST}, \mathit{newST}, \mathit{getST}, \mathit{setST}\}$ with type-arities and term-arities defined as follows:

monadic operation	o	retST	doST	newST	getST	setST
type-arity	$\Box o$	2	3	2	2	2
term-arity	$\#o$	1	2	1	1	2

- Type-constants, term-constants, kinds, constructors and terms:

$$
\begin{array}{rcl}
C \in \mathsf{CONST} & ::= & V_r \mid C_a \mid O \\
c \in \mathsf{Const} & ::= & \mathit{runST} \mid o \mid \ell_{r,m} \\
K \in \mathsf{Kind} & ::= & * \mid K_1 \to K_2 \\
u \in \mathsf{U} & ::= & C \mid X \mid u_1 \to u_2 \mid \forall X\!:K.u \mid \Lambda X\!:K.u \mid u_1[u_2] \\
e \in \mathsf{E} & ::= & c \mid x \mid \lambda x\!:u.e \mid e_1\ e_2 \mid \Lambda X\!:K.e \mid e[u]
\end{array}
$$

  are unchanged, except for constants C and c.
- Values:

$$
\begin{array}{rcl}
v \in \mathsf{Val}^\star & ::= & \lambda x\!:u.e \mid \mathit{runST} \mid \mathit{runST}[u] \mid \ell_{r,m} \mid o[\overline{u}] \text{ with } |\overline{u}| < \Box o \\
& \mid & o[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \text{ and } |\overline{e}| \leq \#o
\end{array}
$$

- Stores $\mu \in \mathsf{S}^\star$ are unchanged.
- Dynamic signatures $\Sigma \in \mathsf{Sig}$ and Static signatures Δ are unchanged. But dynamic signatures keep track of the type encoding of a region instead of the types of the monadic operations associated with the region.
- Descriptions $d \in \mathsf{D}^\star$ unchanged.

Erasure $|\_|$ is defined by analogy with Definition 5.2 and satisfies the same properties.

The evaluation judgments for the instrumented semantics are unchanged.

Figures 7 and 8 give the evaluation rules for the instrumented semantics. The rules for pure evaluation are identical to the rules in Figure 4 except for runST. The rules for monadic evaluation are similar to the rules in Figure 4: the differences are that a monadic operation, like ret_r, which was implicitly parameterized by a region now takes an explicit type argument identifying the region, $\mathit{retST}[u_r]$. Since region information is now encoded in types, the check that the region names match in each operation, is now in the side-condition $V_r =^u_{\beta\eta} u_r$.

6.4 Type safety and compatibility

The statements of the technical results are unchanged, but the auxiliary definitions used for stating type safety have to be redefined.

Pure Evaluation

$$\Delta|\Sigma; v \Longrightarrow \emptyset|\Sigma; v \qquad \frac{\Delta|\Sigma_0; e_1 \Longrightarrow \emptyset|\Sigma_1; \lambda x\colon u.e \quad \Delta|\Sigma_1; e[x := e_2] \Longrightarrow \Delta'|\Sigma_2; v}{\Delta|\Sigma_0; e_1\ e_2 \Longrightarrow \Delta'|\Sigma_2; v}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; runST}{\Delta|\Sigma; e[u] \Longrightarrow \emptyset|\Sigma'; runST[u]} \qquad \frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}]}{\Delta|\Sigma; e[u] \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}, u]}\ |\overline{u}| < \Box o$$

$$\frac{\Delta|\Sigma; e_1 \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}](\overline{e})}{\Delta|\Sigma; e_1\ e_2 \Longrightarrow \emptyset|\Sigma'; o_r[\overline{u}](\overline{e}, e_2)}\ |\overline{u}| = \Box o \wedge |\overline{e}| < \# o$$

$$\frac{\Delta|\Sigma; e \Longrightarrow C_a\colon K, \Delta'|\Sigma'; v}{\Delta|\Sigma; e[u] \Longrightarrow \Delta'|(\Sigma'; v)[C_a := u]}$$

$$\frac{\Delta, C_a\colon K|\Sigma; e[X := C_a] \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; \Lambda X\colon K.e \Longrightarrow C_a\colon K, \Delta'|\Sigma'; v}\ a \text{ fresh}$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0; e_1 \Longrightarrow \emptyset|\Sigma_1; runST[u]\\ \Delta|\Sigma_1,\ V_r\colon *; \emptyset, e_2[V_r] \stackrel{r}{\Longrightarrow} \Sigma_2; \mu, e\\ \Delta|\Sigma_2; e \Longrightarrow \Delta'|\Sigma_3; v\end{array}}{\Delta|\Sigma_0; e_1\ e_2 \Longrightarrow \Delta'|\Sigma_3; v}\ r \text{ fresh}$$

Monadic Evaluation

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; retST\ [u_r, u](e')}{\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma'; \mu, e'}\ V_r =^u_{\beta\eta} u_r$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0; e \Longrightarrow \emptyset|\Sigma_1; doST\ [u_r, u_0, u_1](e_0, e_1)\\ \Delta|\Sigma_1; \mu_0, e_0 \stackrel{r}{\Longrightarrow} \Sigma_2; \mu_1, e_0'\\ \Delta|\Sigma_2; \mu_1, e_1\ e_0' \stackrel{r}{\Longrightarrow} \Sigma_3; \mu_2, e'\end{array}}{\Delta|\Sigma_0; \mu_0, e \stackrel{r}{\Longrightarrow} \Sigma_3; \mu_2, e'}\ V_r =^u_{\beta\eta} u_r$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; newST\ [u_r, u](e_0)}{\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma', \ell_{r,m}\colon \mathrm{Ref}[V_r, u]; \mu\{m = e_0\}, \ell_{r,m}}\ m \notin dom(\mu) \wedge V_r =^u_{\beta\eta} u_r$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0; e \Longrightarrow \emptyset|\Sigma_1; getST\ [u_r, u](e_0)\\ \Delta|\Sigma_1; e_0 \Longrightarrow \emptyset|\Sigma_2; \ell_{r,m}\end{array}}{\Delta|\Sigma_0; \mu, e \stackrel{r}{\Longrightarrow} \Sigma_2; \mu, e'}\ e' = \mu(m) \wedge V_r =^u_{\beta\eta} u_r$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0; e \Longrightarrow \emptyset|\Sigma_1; setST\ [u_r, u](e_0, e_1)\\ \Delta|\Sigma_1; e_0 \Longrightarrow \emptyset|\Sigma_2; \ell_{r,m}\end{array}}{\Delta|\Sigma_0; \mu, e \stackrel{r}{\Longrightarrow} \Sigma_2; \mu\{m = e_1\}, \ell_{r,m}}\ m \in dom(\mu) \wedge V_r =^u_{\beta\eta} u_r$$

Fig. 7. Evaluation Rules for Instrumented Semantics I (strict state, runST)

Notation 6.2 The auxiliary definitions of Notation 5.5 are modified as follows:

- $\Delta, \Sigma \models J \stackrel{\text{def}}{\iff} \Sigma_{runST}, \Delta, \Sigma \vdash J$ and for all $C\colon K$ and $c\colon \tau$ declared in Σ:
  — $C \equiv V_r$ implies $K \equiv *$
  — $c \equiv \ell_{r,m}$ implies $\tau \equiv \mathrm{Ref}[V_r, \tau']$ for some τ'
- Reg_Σ is the set of regions in Σ, *i.e.*
  $r \in \mathsf{Reg}_\Sigma \stackrel{\text{def}}{\iff} V_r$ is declared in Σ.

Pure and Monadic Run-Time Errors

The rules for error propagation follow the ML convention, those for error generation are:

$$\frac{\Delta|\Sigma; e_1 \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; e_1 e_2 \Longrightarrow err} \quad \begin{array}{l}\Delta' \not\equiv \emptyset \text{ or} \\ v \notin \lambda x{:}u.e \mid runST[u] \mid o[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| < \# o\end{array}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; v}{\Delta|\Sigma; e[u] \Longrightarrow err} \; v \notin runST \mid o[\overline{u}] \text{ with } |\overline{u}| < \Box o$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} err} \quad \begin{array}{l}\Delta' \not\equiv \emptyset \text{ or} \\ v \notin o[u_r, \overline{u}](\overline{e}) \text{ with } u_r =^u_{\beta\eta} V_r \wedge 1 + |\overline{u}| = \Box o \wedge |\overline{e}| = \# o\end{array}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; getST\ [u_r, u](e_0) \quad \Delta|\Sigma'; e_0 \Longrightarrow \Delta'|\Sigma''; v}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} err} \; \Delta' \not\equiv \emptyset \vee v \notin \mathsf{Loc}_{r,\mu}$$

$$\frac{\Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; setST\ [u_r, u](e_0, e_1) \quad \Delta|\Sigma'; e_0 \Longrightarrow \Delta'|\Sigma''; v}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} err} \; \Delta' \not\equiv \emptyset \vee v \notin \mathsf{Loc}_{r,\mu}$$

Fig. 8. Evaluation Rules for Instrumented Semantics II (strict state, runST)

- The definitions of $\mathsf{Loc}_{r,\Sigma}$, $\Sigma \hookrightarrow \Sigma'$, $\Sigma \overset{r}{\hookrightarrow} \Sigma'$ and $\Delta, \Sigma \models_r \mu$ are unchanged.

The statement of type safety for the instrumented semantics requires a minor adjustment regarding the type of e in the second clause.

Theorem 6.3 (Type Safety for Instrumented Semantics)

1. If $\Delta|\Sigma; e \Longrightarrow d$ and $\Delta, \Sigma \models e{:}\tau$, then exist Δ', Σ', v and τ' *s.t.* $d \equiv (\Delta'|\Sigma'; v)$, $\tau =^u_{\beta\eta} \forall\Delta'.\tau'$, $\Sigma \hookrightarrow \Sigma'$ and $\Delta, \Delta', \Sigma' \models v{:}\tau'$.
2. If $\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} d$, $\Delta, \Sigma \models_r \mu$ and $\Delta, \Sigma \models e{:}\mathrm{ST}[V_r, \tau]$, then exist Σ', μ' and e' *s.t.* $d \equiv (\Sigma'; \mu', e')$, $\Sigma \overset{r}{\hookrightarrow} \Sigma'$, $\Delta, \Sigma' \models_r \mu'$ and $\Delta, \Sigma' \models e'{:}\tau$.

The statement of compatibility is unchanged. The statement of type safety for the dynamic semantics reflects the change in the syntax, and more specifically in the constants allowed in user-defined programs.

Corollary 6.4 (Type Safety for Dynamic Semantics)
If $\Sigma_{runST}; \emptyset \vdash e{:}\tau$ and $|e| \Longrightarrow d'$, then $d' \not\equiv err$.

7 *run* with Lazy State

In this variant of the language pure evaluation is CBN and monadic evaluation is lazy. This variant gives rise to lazy monadic state (Launchbury & Peyton Jones, 1994) and is significantly more complicated.

7.1 Dynamic Semantics

Because of laziness of the monadic constants, commands are no longer performed when they are first encountered. Instead, monadic evaluation immediately returns

Pure Evaluation

$$\xi, v \Longrightarrow \xi, v \qquad \frac{\xi, e_1 \Longrightarrow \xi', \lambda x.e \quad \xi', e[x{:}= e_2] \Longrightarrow d}{\xi, e_1\ e_2 \Longrightarrow d}$$

$$\frac{\xi \overset{r,p}{\Longrightarrow} \xi', e' \quad \xi', e' \Longrightarrow d}{\xi, s_{r,p} \Longrightarrow d} \qquad \frac{\xi, e_1 \Longrightarrow \xi', o(\overline{e})}{\xi, e_1\ e_2 \Longrightarrow \xi', o(\overline{e}, e_2)}\ |\overline{e}| < \#o$$

$$\frac{\xi, e_1 \Longrightarrow \xi', run \quad \xi'\{r = (\emptyset, \emptyset)\}, e_2(\mathsf{Op}) \overset{r,nil}{\Longrightarrow} \xi'', p \quad \xi'', s_{r,p} \Longrightarrow d}{\xi, e_1\ e_2 \Longrightarrow d}\ r \notin dom(\xi')$$

Generation of Suspensions

$$\xi, e \overset{r,k}{\Longrightarrow} \xi(r).\rho\{p = (e, \mathsf{c}, k)\}, p \qquad p \notin dom(\xi(r).\rho)$$

Fig. 9. Evaluation Rules for Dynamic Semantics I (lazy state, run)

a *suspension* which is treated as a first-class entity. The suspension may or may not be forced as a result of the demand-driven evaluation mechanism. This scenario is complicated by the fact that suspensions cannot be forced independently of each other if the semantics is to maintain the (required) appearance of sequential execution of the effects. In particular forcing a command that looks up the value of a location should not be done without ensuring that all previous commands that might set the location have also been forced. To realize this, suspensions are maintained in lists of dependencies.

This is however not enough! Suspensions can be forced in two different ways that have different semantics. Consider a chain of dependencies where suspension s_3 depends on suspension s_2 which in turn depends on suspension s_1. Further consider the case where s_1 is the suspended monadic command $set(\ell_m, 0)$, s_2 is the suspended monadic command $ret(5)$, and s_3 is the suspended monadic command $get(\ell_m)$. If the value of s_2 is demanded during execution we can force s_2 and immediately return the value 5 *without* forcing s_1 since the latter suspension cannot possibly affect the returned value. However, if the value of s_3 is demanded, then we must force s_2 in a more strict fashion than before, which also forces s_1.

Hence for the dynamic semantics, a new category of suspensions is introduced. The semantics still maintains a set of regions, one for each *run*-expression it encounters. As before, locations have names ℓ_m which implicitly refer to the current region, and hence the dynamic semantics is susceptible to the same pathological examples from Remark 3.3 in which locations from different regions are confused. Suspensions can be forced at any time, even within another region, and hence have global names that includes their originating region.

More formally the syntax of the language, in comparison to that of Section 3, is defined as follows:

- Names $r \in \mathsf{N}$ for regions. Region names are used to index suspensions $s_{r,p}$.
- Names $m, n \in \mathsf{N}$ for locations ℓ_m: the region in which the location is allocated is implicit.

Forcing of Suspensions

$$\xi \overset{r,p}{\Longrightarrow} \xi, e \quad \xi(r).\rho(p) = (e, \mathsf{v}, k) \qquad \frac{\xi, e \Longrightarrow \xi', ret(e')}{\xi \overset{r,p}{\Longrightarrow} \xi'(r).\rho\{p = (e', \mathsf{v}, k)\}, e'} \; \xi(r).\rho(p) = (e, \mathsf{c}, k)$$

$$\frac{\begin{array}{l} \xi, e \Longrightarrow \xi', do(e_0, e_1) \\ \xi', e_0 \overset{r,k}{\Longrightarrow} \xi'', q \\ \xi''(r).\rho\{p = (e_1 \; s_{r,q}, \mathsf{c}, q)\} \overset{r,p}{\Longrightarrow} d \end{array}}{\xi \overset{r,p}{\Longrightarrow} d} \; \xi(r).\rho(p) = (e, \mathsf{c}, k)$$

$$\frac{\xi, e \Longrightarrow \xi', new(e_0) \quad \xi' \overset{r,k}{\Longrightarrow}_s \xi''}{\xi \overset{r,p}{\Longrightarrow} \xi''(r).\mu\{m = e_0\}.\rho\{p = (\ell_m, \mathsf{v}, nil)\}, \ell_m} \; \xi(r).\rho(p) = (e, \mathsf{c}, k) \wedge m \notin dom(\xi''(r).\mu)$$

$$\frac{\begin{array}{l} \xi, e \Longrightarrow \xi', get(e_0) \\ \xi' \overset{r,k}{\Longrightarrow}_s \xi'' \\ \xi'', e_0 \Longrightarrow \xi''', \ell_m \end{array}}{\xi \overset{r,p}{\Longrightarrow} \xi'''(r).\rho\{p = (e', \mathsf{v}, nil)\}, e'} \; \xi(r).\rho(p) = (e, \mathsf{c}, k) \wedge e' = \xi'''(r).\mu(m)$$

$$\frac{\xi, e \Longrightarrow \xi', set(e_0, e_1) \quad \xi' \overset{r,k}{\Longrightarrow}_s \xi'' \quad \xi'', e_0 \Longrightarrow \xi''', \ell_m}{\xi \overset{r,p}{\Longrightarrow} \xi'''(r).\mu\{m = e_1\}.\rho\{p = (\ell_m, \mathsf{v}, nil)\}, \ell_m} \; \xi(r).\rho(p) = (e, \mathsf{c}, k)$$

Strict Forcing of Suspensions

$$\xi \overset{r,nil}{\Longrightarrow}_s \xi \qquad \frac{\xi \overset{r,p}{\Longrightarrow} \xi', \_ \quad \xi' \overset{r,k}{\Longrightarrow}_s \xi''}{\xi \overset{r,p}{\Longrightarrow}_s \xi''(r).\rho\{p = (e, \mathsf{v}, nil)\}} \; \xi'(r).\rho(p) = (e, \mathsf{v}, k)$$

Fig. 10. Evaluation Rules for Dynamic Semantics II (lazy state, run)

- Names $p, q \in \mathsf{N}$ for *suspensions* $s_{r,p}$ within a region.
- Monadic operations $o \in \mathsf{Op} \overset{\text{def}}{=} \{ret, do, new, get, set\}$ and their arities are unchanged.
- Constants $c \in \mathsf{Const} ::= run \mid o \mid \ell_m \mid s_{r,p}$ now include also suspensions.
- Terms $e \in \mathsf{E} ::= c \mid x \mid \lambda x.e \mid e_1 \; e_2$ are unchanged.
- Values $v \in \mathsf{Val} ::= \lambda x.e \mid run \mid \ell_m \mid o(\overline{e})$ with $|\overline{e}| \leq \#o$ are unchanged; note that suspensions are *not* values.
- Stores $\mu \in \mathsf{S} \overset{\text{def}}{=} \mathsf{N} \overset{fin}{\rightarrow} \mathsf{E}_0$ are unchanged.
- Suspension lists $\rho \in \mathsf{P} \overset{\text{def}}{=} \mathsf{N} \overset{fin}{\rightarrow} (\mathsf{E}_0 \times \{\mathsf{v}, \mathsf{c}\} \times (nil + \mathsf{N}))$ of suspended computations. The tag v or c distinguishes between suspensions whose effects have been performed and those that have not. The tag from the set $(nil + \mathsf{N})$ specifies a possible dependency on another suspension: *nil* means no dependency, and a name p means a dependency on suspension $s_{r,p}$ in the current region r. Note that the effect of a suspension may have been performed (if it is a trivial effect like *ret*) without having necessarily forced the effects of the suspensions on which it depends.
- Regions $\xi \in \mathsf{ST} \overset{\text{def}}{=} \mathsf{N} \overset{fin}{\rightarrow} (\mathsf{S} \times \mathsf{P})$ consist of a store and a suspension list; we write $\xi(r).\mu$ and $\xi(r).\rho$ for the two components of the region indexed by r.
- Descriptions $d \in \mathsf{D} ::= (\xi, e) \mid (\xi, p) \mid \xi \mid err$.

Run-Time Errors

The rules for error propagation follow the ML convention, those for error generation are:

$$\frac{\xi, e_1 \Longrightarrow \xi', v}{\xi, e_1\ e_2 \Longrightarrow err}\ v \notin \lambda x.e \mid run \mid o(\overline{e}) \text{ with } |\overline{e}| < \#o$$

$$\xi, e \overset{r,k}{\Longrightarrow} err \quad r \notin dom(\xi)$$

$$\xi \overset{r,p}{\Longrightarrow} err \quad r \notin dom(\xi) \vee p \notin dom(\xi(r).\rho)$$

$$\frac{\xi, e \Longrightarrow \xi', v}{\xi \overset{r,p}{\Longrightarrow} err}\ \xi(r).\rho(p) = (e, \mathsf{c}, k) \wedge v \notin o(\overline{e}) \text{ with } |\overline{e}| = \#o$$

$$\frac{\xi, e \Longrightarrow \xi', get(e_0) \quad \xi' \overset{r,k}{\Longrightarrow}_s \xi'' \quad \xi'', e_0 \Longrightarrow \xi''', v}{\xi \overset{r,p}{\Longrightarrow} err}\ \xi(r).\rho(p) = (e, \mathsf{c}, k) \wedge v \notin \mathsf{Loc}_{\xi'''(r).\mu}$$

$$\frac{\begin{array}{l}\xi, e \Longrightarrow \xi', set(e_0, e_1)\\ \xi' \overset{r,k}{\Longrightarrow}_s \xi''\\ \xi'', e_0 \Longrightarrow \xi''', v\end{array}}{\xi \overset{r,p}{\Longrightarrow} err}\ \xi(r).\rho(p) = (e, \mathsf{c}, k) \wedge v \notin \mathsf{Loc}_{\xi'''(r).\mu}$$

$$\frac{\xi \overset{r,p}{\Longrightarrow} \xi', \_}{\xi \overset{r,p}{\Longrightarrow}_s err}\ r \notin dom(\xi') \vee p \notin dom(\xi'(r).\rho) \vee \xi'(r).\rho(p) \neq (e, \mathsf{v}, k)$$

Fig. 11. Evaluation Rules for Dynamic Semantics III (lazy state, run)

The pure and monadic interpreters are more tightly coupled than in the strict case; in fact ξ gets threaded. There are three evaluation judgments: the pure evaluator is much like before, but the monadic evaluator has been split in three parts: one part generates suspensions, and the others force those suspensions to various degrees depending on the kind of demand.

- $\xi, e \Longrightarrow \xi', v \mid err$ for pure CBN evaluation;
- $\xi, e \overset{r,k}{\Longrightarrow} \xi', p \mid err$ for lazy monadic evaluation which simply generates suspensions: r is the index of the current region and $k \in \mathsf{N} + \{nil\}$ is the index of the suspension (within region r) on which the current command e may depend; the result is the name of a suspension for the current command.
- $\xi \overset{r,p}{\Longrightarrow} \xi', e \mid err$ for forcing evaluation (and updating) of the suspension p within region r. If evaluation does not require the store, then some of the suspensions on which suspension p depends may not be forced. The dependencies among suspensions are maintained in case a later suspension requires the store.
- $\xi \overset{r,k}{\Longrightarrow}_s \xi' \mid err$, where k is either nil or a the name of a suspension p. This forces strict evaluation (and updating) a suspension p within region r, *i.e.* the suspension and anything on which it depends is forced. This mode is entered whenever a command requires access to the store.

Figures 9, 10, and 11 give the evaluation rules for the dynamic semantics. An example that illustrates much of the subtleties of the semantics is the following:

```
run (\ ret do new get set ->
  let omega = omega      -- shortand for any diverging computation
  in do omega (\ _ ->
     do (ret (\ x -> get x)) (\ f ->
     do (new 5) (\ a ->
     f a))))
```

The term consists of a *run*-expression whose body consists of a sequence of monadic operations. The first operation refers to an incalculable monadic operation `omega`; the second operation is also unusual in the sense that it returns a function that when invoked, returns another monadic operation. The evaluation of this example, immediately builds a suspension which contains all the monadic operations, and then forces the suspension lazily since there is no demand for the store from the pure evaluator. Forcing the value of the suspension, creates a linked list of suspensions for the first three operations, and then attempts to evaluate the call `(f a)`. The reference to `f` forces the corresponding suspension which immediately returns the function `(\x -> get x)` without forcing the suspension for `omega` on which it depends. This is clearly the right behavior since only the value of `f` is needed. However the application of `f` requires the execution of `(get a)` which requires that all the monadic operations that might assign to `a` be forced. During this second forcing the suspension for `f` is re-visited and this time, forcing it also forces the suspension for `omega`, which makes the whole program diverge.

7.2 Instrumented Semantics

To define the instrumented semantics, we extend the syntax of Section 7.1 with type information in a way that mirrors the transition from Section 3 to Section 5. In comparison to the language of Section 5, the syntax is defined as follows:

- Names $r \in \mathsf{N}$ for regions. Region names are used to index locations $\ell_{r,m}$, suspensions $s_{r,p}$, monadic type-constructors O_r, and monadic operations o_r.
- Names $m, n \in \mathsf{N}$ for locations $\ell_{r,m}$ within a region.
- Names $a \in \mathsf{N}$ for constructor constants C_a created by evaluation of polymorphic terms.
- Names $p, q \in \mathsf{N}$ for suspensions $s_{r,p}$ within a region.
- Monadic type-constructors $O \in \mathsf{OP} \stackrel{\text{def}}{=} \{M, R\}$ are unchanged.
- Monadic operations $o \in \mathsf{Op} \stackrel{\text{def}}{=} \{\mathit{ret}, \mathit{do}, \mathit{new}, \mathit{get}, \mathit{set}\}$ and their arities are unchanged.
- Type- and term-constants, kinds, constructors and terms:

$$
\begin{array}{rcl}
C & ::= & C_a \mid O_r \\
c \in \mathsf{Const}^\star & ::= & \mathit{run} \mid o_r \mid \ell_{r,m} \mid s_{r,p} \\
K & ::= & * \mid K_1 \to K_2 \\
u & ::= & C \mid X \mid u_1 \to u_2 \mid \forall X{:}\,K.u \mid \Lambda X{:}\,K.u \mid u_1[u_2] \\
e \in \mathsf{E}^\star & ::= & c \mid x \mid \lambda x{:}\,u.e \mid e_1\ e_2 \mid \Lambda X{:}\,K.e \mid e[u]
\end{array}
$$

Pure Evaluation

$$\Delta|\Sigma;\xi,v \Longrightarrow \emptyset|\Sigma;\xi,v \qquad \frac{\Delta|\Sigma_0;\xi_0,e_1 \Longrightarrow \emptyset|\Sigma_1;\xi_1,\lambda x\colon u.e \quad \Delta|\Sigma_1;\xi_1,e[x\colon= e_2] \Longrightarrow \Delta'|\Sigma_2;\xi_2,v}{\Delta|\Sigma_0;\xi_0,e_1\ e_2 \Longrightarrow \Delta'|\Sigma_2;\xi_2,v}$$

$$\frac{\Delta|\Sigma;\xi,e \Longrightarrow \emptyset|\Sigma';\xi',run}{\Delta|\Sigma;\xi,e[u] \Longrightarrow \emptyset|\Sigma';\xi',run[u]} \qquad \frac{\Delta|\Sigma;\xi,e \Longrightarrow \emptyset|\Sigma';\xi',o_r[\overline{u}]}{\Delta|\Sigma;\xi,e[u] \Longrightarrow \emptyset|\Sigma';\xi',o_r[\overline{u},u]}\ |\overline{u}| < \Box o$$

$$\frac{\Delta|\Sigma;\xi,e_1 \Longrightarrow \emptyset|\Sigma';\xi',o_r[\overline{u}](\overline{e})}{\Delta|\Sigma;\xi,e_1\ e_2 \Longrightarrow \emptyset|\Sigma';\xi',o_r[\overline{u}](\overline{e},e_2)}\ |\overline{u}| = \Box o \wedge |\overline{e}| < \#o$$

$$\frac{\Delta|\Sigma;\xi,e \Longrightarrow C_a\colon K,\Delta'|\Sigma';\xi',v}{\Delta|\Sigma;\xi,e[u] \Longrightarrow \Delta'|(\Sigma';\xi',v)[C_a\colon= u]}$$

$$\frac{\Delta,C_a\colon K|\Sigma;\xi,e[X\colon= C_a] \Longrightarrow \Delta'|\Sigma';\xi',v}{\Delta|\Sigma;\xi,\Lambda X\colon K.e \Longrightarrow C_a\colon K,\Delta'|\Sigma';\xi',v}\ a \text{ fresh}$$

$$\frac{\Delta|\Sigma_0;\xi_0 \stackrel{r,p}{\Longrightarrow} \Sigma_1;\xi_1,e \quad \Delta|\Sigma_1;\xi_1,e \Longrightarrow \Delta'|\Sigma_2;\xi_2,v}{\Delta|\Sigma_0;\xi_0,s_{r,p} \Longrightarrow \Delta'|\Sigma_2;\xi_2,v}$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0;\xi_0,e_1 \Longrightarrow \emptyset|\Sigma_1;\xi_1,run[u] \\ \Delta|\Sigma_1+\Sigma_r^{op};\xi_1\{r=(\emptyset,\emptyset)\},e_2[\mathsf{OP}_r](\mathsf{Op}_r) \stackrel{r,nil,u}{\Longrightarrow} \Sigma_2;\xi_2,p \\ \Delta|\Sigma_2;\xi_2,s_{r,p} \Longrightarrow \Delta'|\Sigma_3;\xi_3,v\end{array}}{\Delta|\Sigma_0;\xi_0,e_1\ e_2 \Longrightarrow \Delta'|\Sigma_3;\xi_3,v}\ r \text{ fresh, } \Sigma_r^{op} \text{ as in Figure 4}$$

Generation of Suspensions

$$\Delta|\Sigma;\xi,e \stackrel{r,k,u}{\Longrightarrow} \Sigma,s_{r,p}\colon u;\xi(r).\rho\{p=(e,\mathsf{c},k)\},p \qquad p \notin dom(\xi(r).\rho)$$

Fig. 12. Evaluation Rules for Instrumented Semantics I (lazy state, run)

are unchanged, excepts $\mathsf{Const}^\star$ which now include also suspensions.

- Values: $v \in \mathsf{Val}^\star$ are unchanged.
- Stores $\mu \in \mathsf{S}^\star \stackrel{\text{def}}{=} \mathsf{N} \stackrel{fin}{\to} \mathsf{E}_0^\star$ are unchanged.
- Suspension lists $\rho \in \mathsf{P}^\star \stackrel{\text{def}}{=} \mathsf{N} \stackrel{fin}{\to} (\mathsf{E}_0^\star \times \{\mathsf{v},\mathsf{c}\} \times (nil + \mathsf{N}))$ and regions $\xi \in \mathsf{ST}^\star \stackrel{\text{def}}{=} \mathsf{N} \stackrel{fin}{\to} (\mathsf{S}^\star \times \mathsf{P}^\star)$ mimic those for the dynamic semantics; we write $\mathsf{Loc}_{r,\xi}$ for the set $\{\ell_{r,m}|m \in dom(\xi(r).\mu)\}$ and $\mathsf{Susp}_{r,\xi}$ for the set $\{s_{r,p}|p \in dom(\xi(r).\rho)\}$.
- Dynamic signatures: $\Sigma \in \mathsf{Sig}\colon\colon= \emptyset \mid \Sigma,C\colon K \mid \Sigma,c\colon u$ and Static signatures $\Delta\colon\colon= \emptyset \mid \Delta,C_a\colon K$ are unchanged.
- Descriptions: $d \in \mathsf{D}^\star\colon\colon= (\Delta|\Sigma;\xi;v) \mid (\Sigma;\xi,p) \mid (\Sigma;\xi,e) \mid (\Sigma;\xi) \mid err$.

Erasure $|\_|$ is defined by analogy with Definition 5.2, in particular $|s_{r,p}| = s_{r,p}$, and it satisfies the same properties.

The instrumented semantics, like the dynamic semantics, is given in terms of four mutually recursive interpreters. The evaluation judgments are:

- $\Delta|\Sigma;\xi,e \Longrightarrow \Delta'|\Sigma';\xi',v \mid err$ for pure CBN evaluation;
- $\Delta|\Sigma;\xi,e \stackrel{r,k,u}{\Longrightarrow} \Sigma';\xi',p \mid err$ for lazy monadic evaluation which generates suspensions; the evaluation judgment takes an additional parameter u which is

Forcing of Suspensions

$$\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} \Sigma;\xi,e \qquad \xi(r).\rho(p) = (e,\mathsf{v},k)$$

$$\frac{\Delta|\Sigma;\xi,e \Longrightarrow \emptyset|\Sigma';\xi',ret_r[u](e')}{\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} \Sigma';\xi'(r).\rho\{p=(e',\mathsf{v},k)\},e'}\ \xi(r).\rho(p) = (e,\mathsf{c},k)$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0;\xi,e \Longrightarrow \emptyset|\Sigma_1;\xi',do_r[u,u_1](e_0,e_1)\\ \Delta|\Sigma_1;\xi',e_0 \overset{r,k,u}{\Longrightarrow} \Sigma_2;\xi'',q\\ \Delta|\Sigma_2;\xi''(r).\rho\{p=(e_1\ s_{r,q},\mathsf{c},q)\} \overset{r,p}{\Longrightarrow} d\end{array}}{\Delta|\Sigma_0;\xi \overset{r,p}{\Longrightarrow} d}\ \xi(r).\rho(p) = (e,\mathsf{c},k)$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0;\xi,e \Longrightarrow \emptyset|\Sigma_1;\xi',new_r[u](e_0)\\ \Delta|\Sigma_1;\xi' \overset{r,k}{\Longrightarrow}_s \Sigma_2;\xi''\\ \xi''' = \xi''(r).\mu\{m=e_0\}.\rho\{p=(\ell_{r,m},\mathsf{v},nil)\}\end{array}}{\Delta|\Sigma_0;\xi \overset{r,p}{\Longrightarrow} \Sigma_2,\ell_{r,m}{:}\,R_r u;\xi''',\ell_{r,m}}\ \begin{array}{l}\xi(r).\rho(p) = (e,\mathsf{c},k)\ \wedge\\ m \notin dom(\xi''(r).\mu)\end{array}$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0;\xi,e \Longrightarrow \emptyset|\Sigma_1;\xi',get_r[u](e_0)\\ \Delta|\Sigma_1;\xi' \overset{r,k}{\Longrightarrow}_s \Sigma_2;\xi''\\ \Delta|\Sigma_2;\xi'',e_0 \Longrightarrow \emptyset|\Sigma_3;\xi''',\ell_{r,m}\end{array}}{\Delta|\Sigma_0;\xi \overset{r,p}{\Longrightarrow} \Sigma_3;\xi'''(r).\rho\{p=(e',\mathsf{v},nil)\},e'}\ \xi(r).\rho(p) = (e,\mathsf{c},k) \wedge e' = \xi'''(r).\mu(m)$$

$$\frac{\begin{array}{l}\Delta|\Sigma_0;\xi,e \Longrightarrow \emptyset|\Sigma_1;\xi',set_r[u](e_0,e_1)\\ \Delta|\Sigma_1;\xi' \overset{r,k}{\Longrightarrow}_s \Sigma_2;\xi''\\ \Delta|\Sigma_2;\xi'',e_0 \Longrightarrow \emptyset|\Sigma_3;\xi''',\ell_{r,m}\end{array}}{\Delta|\Sigma_0;\xi \overset{r,p}{\Longrightarrow} \Sigma_3;\xi'''(r).\mu\{m=e_1\}.\rho\{p=(\ell_{r,m},\mathsf{v},nil)\},\ell_{r,m}}\ \xi(r).\rho(p) = (e,\mathsf{c},k)$$

Strict Forcing of Suspensions

$$\Delta|\Sigma;\xi \overset{r,nil}{\Longrightarrow}_s \Sigma;\xi \qquad \frac{\begin{array}{l}\Delta|\Sigma_0;\xi \overset{r,p}{\Longrightarrow} \Sigma_1;\xi',\_\\ \Delta|\Sigma_1;\xi' \overset{r,k}{\Longrightarrow}_s \Sigma_2;\xi''\end{array}}{\Delta|\Sigma_0;\xi \overset{r,p}{\Longrightarrow}_s \Sigma_2;\xi''(r).\rho\{p=(e,\mathsf{v},nil)\}}\ \xi'(r).\rho(p) = (e,\mathsf{v},k)$$

Fig. 13. Evaluation Rules for Instrumented Semantics II (lazy state, run)

the type of the suspension to generate. The type is used to augment the dynamic signature.

- $\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} \Sigma';\xi',e' \mid err$ for forcing evaluation of suspension p within region r.
- $\Delta|\Sigma;\xi \overset{r,k}{\Longrightarrow}_s \Sigma';\xi' \mid err$ where k is either nil or the name of a suspension p. This forces strict evaluation of the suspension $s_{r,p}$ and everything on which it depends.

Note that the result of lazy monadic evaluation or evaluation of a suspension is never *polymorphic*. Figures 12, 13, and 14 give the rules for the instrumented semantics.

7.3 Type safety and compatibility

We show that well-formed programs cannot go wrong, which amounts to proving subject reduction for the instrumented semantics.

Run-Time Errors

The rules for error propagation follow the ML convention, those for error generation are:

$$\frac{\Delta|\Sigma;\xi,e_1 \Longrightarrow \Delta'|\Sigma';\xi',v}{\Delta|\Sigma;\xi,e_1\ e_2 \Longrightarrow err}\ \begin{array}{l}\Delta' \not\equiv \emptyset \text{ or}\\ v \notin \lambda x{:}\,u.e \mid run[u] \mid o_r[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| < \#o\end{array}$$

$$\frac{\Delta|\Sigma;\xi,e \Longrightarrow \emptyset|\Sigma';\xi',v}{\Delta|\Sigma;\xi,e[u] \Longrightarrow err}\ v \notin run \mid o_r[\overline{u}] \text{ with } |\overline{u}| < \Box o$$

$$\Delta|\Sigma;\xi,e \overset{r,k,u}{\Longrightarrow} err \text{ if } r \notin dom(\xi)$$

$$\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} err \text{ if } r \notin dom(\xi) \vee p \notin dom(\xi(r).\rho)$$

$$\frac{\Delta|\Sigma;\xi,e \Longrightarrow \Delta'|\Sigma';\xi',v}{\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} err}\ \begin{array}{l}\xi(r).\rho(p) = (e,\mathsf{c},k) \text{ and}\\ (\Delta' \not\equiv \emptyset \text{ or } v \notin o_r[\overline{u}](\overline{e}) \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| = \#o)\end{array}$$

$$\frac{\begin{array}{l}\Delta|\Sigma;\xi,e \Longrightarrow \emptyset|\Sigma';\xi',get_r[u](e_0)\\ \Delta|\Sigma';\xi' \overset{r,k}{\Longrightarrow}_s \Sigma'';\xi''\\ \Delta|\Sigma'';\xi'',e_0 \Longrightarrow \Delta'|\Sigma''';\xi''',v\end{array}}{\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} err}\ \xi(r).\rho(p) = (e,\mathsf{c},k) \wedge (\Delta' \not\equiv \emptyset \vee v \notin \mathsf{Loc}_{r,\xi'''})$$

$$\frac{\begin{array}{l}\Delta|\Sigma;\xi,e \Longrightarrow \emptyset|\Sigma';\xi',set_r[u](e_0,e_1)\\ \Delta|\Sigma';\xi' \overset{r,k}{\Longrightarrow}_s \Sigma'';\xi''\\ \Delta|\Sigma'';\xi'',e_0 \Longrightarrow \Delta'|\Sigma''';\xi''',v\end{array}}{\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} err}\ \xi(r).\rho(p) = (e,\mathsf{c},k) \wedge (\Delta' \not\equiv \emptyset \vee v \notin \mathsf{Loc}_{r,\xi'''})$$

$$\frac{\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} \Sigma';\xi',\_}{\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow}_s err}\ r \notin dom(\xi') \vee p \notin dom(\xi'(r).\rho) \vee \xi'(r).\rho(p) \neq (e,\mathsf{v},k)$$

Fig. 14. Evaluation Rules for Instrumented Semantics III (lazy state, run)

Notation 7.1 The auxiliary definitions of Notation 5.5 are modified as follows:

The definition of $\Delta,\Sigma \models J$ has an extra case for $c{:}\,\tau$, *i.e.* $c \equiv s_{r,p}$, but then τ can be any type.

- The definitions of Reg_Σ and $\mathsf{Loc}_{r,\Sigma}$ are unchanged.
- $\mathsf{Susp}_{r,\Sigma}$ is the set of suspensions of region r in Σ, *i.e.* $m \in \mathsf{Susp}_{r,\Sigma} \overset{\text{def}}{\iff} s_{r,p}$ is declared in Σ.
- $\Delta,\Sigma \models \xi \overset{\text{def}}{\iff} \Delta,\Sigma \models$ and $\mathsf{Reg}_\Sigma = dom(\xi)$ and for all names $r,m \in \mathsf{N}$:
  - — $\mathsf{Loc}_{r,\Sigma} = \mathsf{Loc}_{r,\xi}$ and $\mathsf{Susp}_{r,\Sigma} = \mathsf{Susp}_{r,\xi}$
  - — $\ell_{r,m}{:}\,R_r[\tau]$ in Σ and $e \equiv \xi(r).\mu(m)$ imply $\Delta,\Sigma \models e{:}\,\tau$
  - — $s_{r,p}{:}\,\tau$ in Σ and $\xi(r).\rho(m) \equiv (e,\mathsf{c},nil)$ imply $\Delta,\Sigma \models e{:}\,M_r[\tau]$
  - — $s_{r,p}{:}\,\tau$ in Σ and $\xi(r).\rho(m) \equiv (e,\mathsf{v},nil)$ imply $\Delta,\Sigma \models e{:}\,\tau$
  - — $s_{r,p}{:}\,\tau$ in Σ and $\xi(r).\rho(m) \equiv (e,\mathsf{c},q)$ imply $\Delta,\Sigma \models e{:}\,M_r[\tau]$ and $s_{r,q} \in \mathsf{Susp}_{r,\Sigma}$
  - — $s_{r,p}{:}\,\tau$ in Σ and $\xi(r).\rho(m) \equiv (e,\mathsf{v},q)$ imply $\Delta,\Sigma \models e{:}\,\tau$ and $s_{r,q} \in \mathsf{Susp}_{r,\Sigma}$

In the statement of type safety for the instrumented semantics with lazy state the signatures Σ and Σ' before and after evaluation are related only by prefixing $\Sigma \preceq \Sigma'$.

Indeed, the more constrained relations $\Sigma \hookrightarrow \Sigma'$ and $\Sigma \overset{r}{\hookrightarrow} \Sigma'$ (which hold for the instrumented semantics with strict state) are false.

Theorem 7.2 (Type Safety for Instrumented Semantics)

1. If $\Delta|\Sigma;\xi, e \Longrightarrow d$, $\Delta, \Sigma \models \xi$ and $\Delta, \Sigma \models e{:}\,\tau$,
   then exist Δ', Σ', ξ', v and τ' *s.t.*
   $d \equiv (\Delta'|\Sigma';\xi', v)$, $\tau =^{u}_{\beta\eta} \forall\Delta'.\tau'$, $\Sigma \preceq \Sigma'$, $\Delta, \Delta', \Sigma' \models \xi'$ and $\Delta, \Delta', \Sigma' \models v{:}\,\tau'$
2. If $\Delta|\Sigma;\xi, e \overset{r,k,u}{\Longrightarrow} d$, $\Delta, \Sigma \models \xi$ and $\Delta, \Sigma \models e{:}\,M_r[u]$, and ($k = nil$ or $s_{r,k} \in \mathsf{Susp}_{r,\Sigma}$),
   then exist Σ', ξ' and p *s.t.*
   $d \equiv (\Sigma';\xi', p)$, $\Sigma \preceq \Sigma'$, $\Delta, \Sigma' \models \xi'$ and $\Delta, \Sigma' \models s_{r,p}{:}\,u$
3. If $\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} d$, $\Delta, \Sigma \models \xi$, $\Delta, \Sigma \models s_{r,p}{:}\,\tau$,
   then exist Σ', ξ', e' and k *s.t.*
   $d \equiv (\Sigma';\xi', e')$, $\Sigma \preceq \Sigma'$, $\Delta, \Sigma' \models \xi'$ and $\xi'(r).\rho(p) = (e', \mathsf{v}, k)$
4. If $\Delta|\Sigma;\xi \overset{r,k}{\Longrightarrow}_s d$, $\Delta, \Sigma \models \xi$, $\Delta, \Sigma \models s_{r,p}{:}\,\tau$, and ($k = nil$ or $s_{r,k} \in \mathsf{Susp}_{r,\Sigma}$),
   then exist Σ' and ξ' *s.t.*
   $d \equiv (\Sigma';\xi')$, $\Sigma \preceq \Sigma'$, $\Delta, \Sigma' \models \xi'$

Compatibility has to be reformulated to account for the more complex evaluation judgments, but the basic intuition is unchanged, *i.e.* if an error may occur in the dynamic semantics, then it may also occur in the instrumented semantics.

Theorem 7.3 (Compatibility)
For every Δ, Σ, e, ξ, r, p, k, u, and d' the following implications hold:

- $|\xi|, |e| \Longrightarrow d'$ implies exists d *s.t.* $\Delta|\Sigma;\xi, e \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$)
- $|\xi|, |e| \overset{r,k}{\Longrightarrow} d'$ implies exists d *s.t.* $\Delta|\Sigma;\xi, e \overset{r,k,u}{\Longrightarrow} d$ and ($d' \equiv |d|$ or $d \equiv err$)
- $|\xi| \overset{r,p}{\Longrightarrow} d'$ implies exists d *s.t.* $\Delta|\Sigma;\xi \overset{r,p}{\Longrightarrow} d$ and ($d' \equiv |d|$ or $d \equiv err$)
- $|\xi| \overset{r,k}{\Longrightarrow}_s d'$ implies exists d *s.t.* $\Delta|\Sigma;\xi \overset{r,k}{\Longrightarrow}_s d$ and ($d' \equiv |d|$ or $d \equiv err$)

Type safety for the dynamic semantics is about the same set of user-defined programs considered in Corollary 5.9, but the dynamic semantics has changed.

Corollary 7.4 (Type Safety for Dynamic Semantics)
If $\Sigma_{run};\emptyset \vdash e{:}\,\tau$ and $\emptyset, |e| \Longrightarrow d'$, then $d' \not\equiv err$.

8 Extensions

We discuss how to extend the dynamic semantics of Section 3 with a fix-point operation *fix* and a test for equality of locations *eq*. We do not consider the extensions in the context of instrumented semantics or lazy state. We view these extensions as interesting (though unproblematic) for the following reasons:

- *fix* is *independent* of the *run*-construct (like many extensions one can envisage);

$$\frac{e_1 \Longrightarrow run \quad \emptyset, e_2(\mathsf{Op}, eq) \Longrightarrow \mu, e \quad e \Longrightarrow v}{e_1\ e_2 \Longrightarrow v} \qquad \frac{e_1 \Longrightarrow \mathit{fix} \quad e_2\ (\mathit{fix}\ e_2) \Longrightarrow v}{e_1\ e_2 \Longrightarrow v} \qquad \frac{e \Longrightarrow eq}{e\ e_0 \Longrightarrow eq\ e_0}$$

$$\frac{e \Longrightarrow eq\ e_0 \quad e_0 \Longrightarrow \ell_m \quad e_1 \Longrightarrow \ell_m}{e\ e_1 \Longrightarrow \lambda x, y.x} \qquad \frac{e \Longrightarrow eq\ e_0 \quad e_0 \Longrightarrow \ell_m \quad e_1 \Longrightarrow \ell_n}{e\ e_1 \Longrightarrow \lambda x, y.y}\ m \neq n \qquad \frac{e \Longrightarrow eq\ e_0 \quad e_i \Longrightarrow v_i}{e\ e_1 \Longrightarrow err}\ v_i \notin \mathsf{Loc}$$

Fig. 15. Additional Evaluation Rules in the Presence of *fix* and *eq*

- *eq* is a peculiar operation, in fact its type involves the type constructor for locations (and, like the monadic operations, it is passed as a parameter to abstract code), but its result type does not involve the type constructor for computational types (and so it is evaluated by the pure interpreter).

The changes to the syntactic categories are as follows:

- Constants $c \in \mathsf{Const}: := run \mid \mathit{fix} \mid eq \mid o \mid \ell_m$, *i.e.* we have added two new constants; we write Loc for the set $\{\ell_m | m \in \mathsf{N}\}$ of locations.
- Terms $e \in \mathsf{E}: := c \mid x \mid \lambda x.e \mid e_1\ e_2$
- Values $v \in \mathsf{Val}: := \lambda x.e \mid run \mid \mathit{fix} \mid eq \mid eq\ e \mid \ell_m \mid o(\overline{e})$ with $|\overline{e}| \leq \#o$.

Figure 15 gives the additional evaluation rules, and the modified rule for *run*. Only *fix* would be allowed in user-defined programs.

9 Conclusions and Related Work

In this section we discuss what we have done, also in the light of related work, and point out possible future developments.

- **CBV, CBN and lazy semantics**. We have adopted a CBN semantics for the pure interpreter following two previous studies (Launchbury & Peyton Jones, 1995; Launchbury & Sabry, 1997), while another study (Semmelroth & Sabry, 1999) adopts a CBV semantics. Technically speaking, it does not make much of a difference if the pure interpreter adopts CBN or CBV.
  However, CBN is inefficient, so it would be interesting to adopt a lazy semantics for the pure interpreter, and then prove that it would induce the same observational congruence $\approx$ of the pure CBN interpreter.
- **Effect Masking**. Semmelroth and Sabry (1999) show that $runST$ implements a cheap form of effect masking (Lucassen & Gifford, 1988; Talpin & Jouvelot, 1992a). More precisely they give a translation from a type system with effects and regions (EML) to one with $runST$ (MML). The result is related to the relation between effects and monads established by Wadler (1998).
  It seems plausible that the translation given by Semmelroth and Sabry (1999) can be adapted so that the target language (MML) uses our *run*-construct instead of $runST$.

- **Relations to region-based memory management**. While the languages we consider do not have syntactic categories of effects and regions, the instrumented semantics for strict state exhibits certain similarities with region-based memory management (Tofte & Talpin, 1997), namely the two-dimensional structure of the address space, and the store deallocation strategy.

In the paper we have also pointed out a limitation of previous reduction semantics, in which stores are represented as binders. Such semantics prevent the formalization of certain implementation details but are more suitable for equational reasoning. Indeed substantially more work is needed to establish soundness of equational reasoning with respect to our dynamic semantics (even for something as unsurprising as β-equivalence).

A Basic Properties of the Type System

This sections recalls basic facts about the type system for the higher-order λ-calculus. In general, signatures Σ and contexts Γ are sequences of declarations. The notation $\Sigma \preceq \Sigma'$ means that the sequence Σ is a prefix of Σ'. In some cases, we are interested in whether a sequence Σ, viewed as a set, in included in another sequence Σ', also viewed as a set. We write $\Sigma \subseteq \Sigma'$ to describe this situation. (It is always possible to view a well-formed signature or context as a set of declarations, since well-formed signatures and contexts do not allow multiple declarations of the same constant or variable.) A judgment J is either empty, or of the form $u\colon K$, or of the form $e\colon \tau$. The set of declared variables in a context Γ is denoted as $\mathrm{DV}(\Gamma)$.

Proposition A.1

$$< .1 \ \frac{\Sigma, \Sigma' \vdash}{\Sigma \vdash} \qquad < .2 \ \frac{\Sigma, \Sigma'; \Gamma \vdash J}{\Sigma \vdash} \qquad < .3 \ \frac{\Sigma; \Gamma, \Gamma' \vdash J}{\Sigma; \Gamma \vdash}$$

Proposition A.2 (Thinning)

$$T.\Sigma \ \frac{\Sigma; \Gamma \vdash J \quad \Sigma' \vdash}{\Sigma'; \Gamma \vdash J} \ \Sigma \subseteq \Sigma' \qquad T.\Gamma \ \frac{\Sigma; \Gamma, \Delta \vdash J \quad \Sigma; \Gamma' \vdash}{\Sigma; \Gamma', \Delta \vdash J} \ \Gamma \subseteq \Gamma' \wedge \mathrm{DV}(\Gamma') \cap \mathrm{DV}(\Delta) = \emptyset$$

Proposition A.3 (Substitution)

$$S.X \ \frac{\Sigma; \Gamma_1, X\colon K, \Gamma_2 \vdash J \quad \Sigma; \Gamma_1 \vdash u\colon K}{\Sigma; \Gamma_1, \Gamma_2[X\colon= u] \vdash J[X\colon= u]} \qquad S.x \ \frac{\Sigma; \Gamma_1, x\colon u, \Gamma_2 \vdash J \quad \Sigma; \Gamma_1 \vdash e\colon u}{\Sigma; \Gamma_1, \Gamma_2 \vdash J[x\colon= e]}$$

Proposition A.4 (Proper Typing)

$$\frac{\Sigma; \Gamma \vdash e\colon \tau}{\Sigma; \Gamma \vdash \tau\colon *}$$

Proposition A.5 (Generation Lemma for Terms)

The following implications hold:

1. $\Sigma;\Gamma \vdash c\colon\tau$ implies exists τ' *s.t.* $\Sigma;\Gamma \vdash$, $c\colon\tau' \in \Sigma$ and $\tau =^{u}_{\beta\eta} \tau'$
2. $\Sigma;\Gamma \vdash x\colon\tau$ implies exists τ' *s.t.* $\Sigma;\Gamma \vdash$, $x\colon\tau' \in \Gamma$ and $\tau =^{u}_{\beta\eta} \tau'$
3. $\Sigma;\Gamma \vdash \lambda x\colon\tau_1.e\colon\tau$ implies exists τ_2 *s.t.* $\Sigma;\Gamma,x\colon\tau_1 \vdash e\colon\tau_2$ and $\tau =^{u}_{\beta\eta} \tau_1 \to \tau_2$
4. $\Sigma;\Gamma \vdash e_1\ e_2\colon\tau$ implies exist τ_1, τ_2 *s.t.*
   $\Sigma;\Gamma \vdash e_1\colon\tau_1 \to \tau_2$, $\Sigma;\Gamma \vdash e_2\colon\tau_1$ and $\tau =^{u}_{\beta\eta} \tau_2$
5. $\Sigma;\Gamma \vdash \Lambda X\colon K.e\colon\tau$ implies exists τ' *s.t.* $\Sigma;\Gamma,X\colon K \vdash e\colon\tau'$ and $\tau =^{u}_{\beta\eta} \forall X\colon K.\tau'$
6. $\Sigma;\Gamma \vdash e[u]\colon\tau$ implies exist X, K, τ' *s.t.*
   $\Sigma;\Gamma \vdash e\colon\forall X\colon K.\tau'$, $\Sigma;\Gamma \vdash u\colon K$ and $\tau =^{u}_{\beta\eta} \tau'[X\colon= u]$

B Proofs of Type Safety and Compatibility

Proof

Proof of Theorem 5.6. By induction on the derivation of an evaluation judgment, and by applying the generation lemma to the typing assumption. We consider a few cases in detail:

- $\dfrac{(1)\ \Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';o_r[\overline{u}]}{\Delta|\Sigma;e[u] \Longrightarrow \emptyset|\Sigma';o_r[\overline{u},u]}\ |\overline{u}| < \Box o$
  by assumption $\Delta,\Sigma \models e[u]\colon\tau$ and the generation lemma exist X, K, τ_1 *s.t.*
  (2) $\Delta,\Sigma \models e\colon(\forall X\colon K.\tau_1)$ (3) $\Delta,\Sigma \models u\colon K$ (4) $\tau =^{u}_{\beta\eta} \tau_1[X\colon= u]$
  from (2) and the IH for (1) exists τ_2 *s.t.*
  (5) $\Delta,\Sigma' \models o_r[\overline{u}]\colon\tau_2$ (6) $(\forall X\colon K.\tau_1) =^{u}_{\beta\eta} \tau_2$ (7) $\Sigma \hookrightarrow \Sigma'$
  from (5) and (6) by the formation rule (conv) one has
  (8) $\Delta,\Sigma' \models o_r[\overline{u}]\colon(\forall X\colon K.\tau_1)$
  from (3) by thinning one has
  (9) $\Delta,\Sigma' \models u\colon K$
  from (8) and (9) by the formation rule ($\forall$E) follows
  (10) $\Delta,\Sigma' \models o_r[\overline{u},u]\colon\tau_1[X\colon= u]$
  from (4), (10) and (7), taking $\tau' \equiv \tau_1[X\colon= u]$, follows the thesis.
- $\dfrac{(1)\ \Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';v}{\Delta|\Sigma;e[u] \Longrightarrow err}\ v \notin run \mid o_r[\overline{u}]$ with $|\overline{u}| < \Box o$
  by assumption $\Delta,\Sigma \models e[u]\colon\tau$ and the generation lemma exist X, K, τ_1 *s.t.*
  (2) $\Delta,\Sigma \models e\colon(\forall X\colon K.\tau_1)$ (3) $\Delta,\Sigma \models u\colon K$ (4) $\tau =^{u}_{\beta\eta} \tau_1[X\colon= u]$
  from (2) and the IH for (1) exists τ_2 *s.t.*
  (5) $\Delta,\Sigma' \models v\colon\tau_2$ (6) $(\forall X\colon K.\tau_1) =^{u}_{\beta\eta} \tau_2$ (7) $\Sigma \hookrightarrow \Sigma'$
  from (5) and (6) by the formation rule (conv) one has
  (8) $\Delta,\Sigma' \models v\colon(\forall X\colon K.\tau_1)$
  however (8) contradicts the side-condition $v \notin run \mid o_r[\overline{u}]$ with $|\overline{u}| < \Box o$.
  In fact the remaining possibilities for v, *i.e.*
  $\lambda x\colon u.e \mid \ell_{r,m} \mid run[u] \mid o_r[\overline{u}](\overline{e})$ with $|\overline{u}| = \Box o$ and $|\overline{e}| \leq \#o$
  can only have a type of the form $\tau_1 \to \tau_2$, $R_r[\tau]$ or $M_r[\tau]$.
  Therefore the thesis follows, because this case cannot occur, more precisely the side-condition for applying the rule is not satisfied.
- $\dfrac{(1)\ \Delta,C_a\colon K|\Sigma;e[X\colon= C_a] \Longrightarrow \Delta'|\Sigma';v}{\Delta|\Sigma;\Lambda X\colon K.e \Longrightarrow C_a\colon K,\Delta'|\Sigma';v}$ (2) a fresh

by assumption $\Delta, \Sigma \models \Lambda X\colon K.e\colon \tau$ and the generation lemma exists τ_1 *s.t.*
(3) $\Delta, \Sigma; X\colon K \models e\colon \tau_1$ (4) $\tau =^u_{\beta\eta} (\forall X\colon K.\tau_1)$
from (3) and (2) by the substitution lemma
(5) $\Delta, C_a\colon K, \Sigma \models e[X\colon= C_a]\colon \tau_1[X\colon= C_a]$
from (5) and the IH for (1) exists τ_2 *s.t.*
(6) $\tau_1[X\colon= C_a] =^u_{\beta\eta} (\forall \Delta'.\tau_2)$ (7) $\Delta, C_a\colon K, \Delta', \Sigma' \models v\colon \tau_2$ (8) $\Sigma \hookrightarrow \Sigma'$
from (4) and (6) by properties of $=^u_{\beta\eta}$
(9) $\tau =^u_{\beta\eta} (\forall C_a\colon K, \Delta'.\tau_2)$
from (9), (7) and (8), taking $\tau' \equiv \tau_2$, follows the thesis.

- $\dfrac{(1)\ \Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; ret_r[\tau'](e')}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} \Sigma'; \mu, e'}$
by assumption $\Delta, \Sigma \models e\colon M_r[\tau]$ and the IH for (1) exists τ_1 *s.t.*
(2) $M_r[\tau] =^u_{\beta\eta} \tau_1$ (3) $\Delta, \Sigma' \models ret_r[\tau'](e')\colon \tau_1$ (4) $\Sigma \hookrightarrow \Sigma'$
from (2) and (3) by the generation lemma and properties of $=^u_{\beta\eta}$
(5) $\Delta; \Sigma' \models e'\colon \tau'$ (6) $\tau' =^u_{\beta\eta} \tau$
from (3), (4) and the assumption $\Delta, \Sigma \models_r \mu$ follows that
(7) $\Delta, \Sigma' \models_r \mu$
from (5) and (6) by the formation rule (conv) one has
(8) $\Delta, \Sigma' \models e'\colon \tau$
from (4), since $\Sigma \hookrightarrow \Sigma'$ implies $\Sigma \overset{r}{\hookrightarrow} \Sigma'$, follows that
(9) $\Sigma \overset{r}{\hookrightarrow} \Sigma'$
from (9), (7) and (8) follows the thesis.

□

Proof

Proof of Theorem 5.8. The implications are proved by lexicographic induction on the derivation of an evaluation judgment $|e| \Longrightarrow d'$ and $|\mu|, |e| \Longrightarrow d'$ for the dynamic semantics, and the *size* of e and (μ, e). We consider in detail one case, to illustrate why we need the lexicographic induction. Suppose that $|e| \equiv e'_1\ e'_2$ and $\dfrac{e'_1 \Longrightarrow \lambda x.e' \quad e'[x\colon= e'_2] \Longrightarrow d'}{e'_1\ e'_2 \Longrightarrow d'}$ there are three possibilities for e

- $e \equiv e_1\ e_2$, therefore $|e_1| \equiv e'_1$ and $|e_2| \equiv e'_2$.
In this case we apply the IH to Δ, Σ, e_1 and $(\lambda x.e')$. In fact, the derivation of the dynamic evaluation judgment $e'_1 \Longrightarrow \lambda x.e'$ is shorter.
Therefore we have a d *s.t.* $\Delta|\Sigma; e_1 \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv$ *err*).
If $d \equiv (\emptyset|\Sigma_1; \lambda x\colon u.e_0)$ and thus $|e_0| = e'$, then we proceed (and reach the desired conclusion) by applying the IH to Δ, Σ_1, $e_0[x\colon= e_2]$ and d'. In fact, $|e_0[x\colon= e_2]| \equiv e'[x\colon= e'_2]$ and the derivation of the dynamic evaluation judgment $e'[x\colon= e'_2] \Longrightarrow d'$ is shorter.
In all the other cases, namely $d \equiv$ *err* or $d \equiv (\Delta'|\Sigma_1; \lambda x\colon u.e_0)$ with $\Delta' \not\equiv \emptyset$, we get $\Delta|\Sigma; e_1\ e_2 \Longrightarrow$ *err*.
- $e \equiv \Lambda X\colon K.e_0$, therefore $|e_0| \equiv e'_1\ e'_2$.
In this case we apply the IH to $\Delta, C_a\colon K$ (with a fresh), Σ, $e_0[X\colon= C_a]$

and d'. In fact, we have reduced the size of the term (from $\Lambda X{:}\,K.e_0$ to $e_0[X{:=}\ C_a]$), while the dynamic evaluation judgment is unchanged (since $|\Lambda X{:}\,K.e_0| \equiv |e_0[X{:=}\,C_a]|$).
Therefore we have a d *s.t.* $\Delta, C_a{:}\,K|\Sigma; e_0[X{:=}\ C_a] \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$).
If $d \equiv (\Delta'|\Sigma'; v)$ and $|d| = d'$, then $\Delta|\Sigma; \Lambda X{:}\,K.e_0 \Longrightarrow C_a{:}\,K, \Delta'|\Sigma'; v$, and obviously $|C_a{:}\,K, \Delta'|\Sigma'; v| \equiv d'$. Otherwise we get $\Delta|\Sigma; \Lambda X{:}\,K.e_0 \Longrightarrow err$.

- $e \equiv e_0[u]$, therefore $|e_0| \equiv e_1'\ e_2'$.
  In this case we apply the IH to Δ, Σ, e_0 and d'. In fact, we have reduced the size of the term (from $e_0[u]$ to e_0), while the dynamic evaluation judgment is unchanged (since $|e_0[u]| \equiv |e_0|$).
  Therefore we have a d *s.t.* $\Delta|\Sigma; e_0 \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$).
  If $d \equiv (C_a{:}\,K, \Delta'|\Sigma'; v)$ and $|d| = d'$, then $\Delta|\Sigma; e_0[u] \Longrightarrow \Delta'|(\Sigma'; v)[C_a{:=}\,u]$, and obviously $|\Delta'|(\Sigma'; v)[C_a{:=}\,u]| \equiv d'$. Otherwise we get $\Delta|\Sigma; e_0[u] \Longrightarrow err$.

□

C Adding *run* to Haskell

Section 4.2 outlined a way of defining *run* using *runST*. This definition can be made more concrete via an implementation in Haskell (using some non-standard extensions for rank-2 polymorphism in Hugs).

The first step is to encode the necessary types. The type of *run* is:

$$run{:}\,\forall X{:}\,*.(\forall\Gamma_M.X_M[X]) \to X \ \text{, where}$$

$$\begin{aligned}\Gamma_M \equiv\ & X_M, X_R{:}\,* \to * \ ,\\ & x_{ret}{:}\,\forall X{:}\,*.X \to X_M[X] \ ,\\ & x_{do}{:}\,\forall X, Y{:}\,*.X_M[X] \to (X \to X_M[Y]) \to X_M[Y] \ ,\\ & x_{new}{:}\,\forall X{:}\,*.X \to X_M[X_R[X]] \ ,\\ & x_{get}{:}\,\forall X{:}\,*.X_R[X] \to X_M[X] \ ,\\ & x_{set}{:}\,\forall X{:}\,*.X_R[X] \to X \to X_M[X_R[X]]\end{aligned}$$

The operations in Γ_M can be grouped in a record with polymorphic fields as follows:

```
data GammaM m r =
  GammaM { xret :: forall a. a -> m a,
           xdo  :: forall a b. m a -> (a -> m b) -> m b,
           xnew :: forall a. a -> m (r a),
           xget :: forall a. r a -> m a,
           xset :: forall a. r a -> a -> m (r a)
         }
```

It follows that the construct *run* would have the following type in Haskell:

```
run :: forall a. (forall m r. GammaM m r -> m a) -> a
```

Section 4.2 gives the following definition of *run*:

$$run \stackrel{\text{def}}{\equiv} \Lambda X{:}\ast.\ \lambda x{:}(\forall\Gamma_M.X_M[X]).\ runST[X]\ (\Lambda\alpha{:}\ast.x\ (|\Sigma'_M|\ (\alpha)))$$

which can be transliterated as follows:

```
run x =
  let gammaST = GammaM { xret = returnST,
                         xdo = thenStrictST,
                         xnew = newSTRef,
                         xget = readSTRef,
                         xset = \r v -> do writeSTRef r v
                                           return r
                       }
  in runST (x gammaST)
```

Intuitively, given an expression parameterized by generic monadic operations, these operations are first specialized to those of the built-in state monad, and *runST* is applied to the result. For example, one can use such a *run* construct as follows:

```
test = run (\ gammaM ->
               let retM = xret gammaM
                   doM = xdo gammaM
                   newM = xnew gammaM
                   getM = xget gammaM
                   setM = xset gammaM
               in doM (newM 0) (\ x ->
                  doM (setM x 7) (\ _ ->
                  doM (getM x) (\ v ->
                  doM (retM "hello") (\ _ ->
                  retM (v+v))))))
```

which produces 14 using `hugs -98` after importing the module `ST`. Note that operation `retM` is used polymorphically, and hence must be let-bound. In general, one cannot use pattern-matching to extract the monadic operations from the record since this would λ-bind the operations to a monomorphic type.

References

Barendregt, Henk P. (1991). Lambda calculi with types. Abramsky, S., Gabbay, D. M., & Maibaum, T. S. E. (eds), *Handbook of logic in computer science*. Oxford: Oxford University Press.

Cardelli, Luca. (1996). Type systems. Tucker, Allen B. (ed), *Handbook of computer science and engineering*. CRC Press.

Chen, K., & Odersky, M. (1994). A type system for a lambda calculus with assignment. *Theoretical aspects of computer software*. Springer Verlag, LNCS 789.

Geuvers, Herman. (1993). *Logics and type systems*. Ph.D. thesis, Computer Science Institute, Katholieke Universiteit Nijmegen.

Harper, Robert. (1994). A simplified account of polymorphic references. *Information processing letters*, **51**(4), 201–206. See also note (Harper, 1996).

Harper, Robert. (1996). A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information processing letters*, **57**(1), 15–16. See (Harper, 1994).

Harper, Robert, & Lillibridge, Mark. (1993). Explicit polymorphism and CPS conversion. *Pages 206–219 of:* the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York.

Launchbury, John, & Peyton Jones, Simon L. (1994). Lazy functional state threads. *Pages 24–35 of:* the *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York.

Launchbury, John, & Peyton Jones, Simon L. (1995). State in Haskell. *Lisp Symbol. Comput.*, **8**, 193–341.

Launchbury, John, & Sabry, Amr. (1997). Monadic state: Axiomatization and type safety. *Pages 227–238 of:* the *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

Lucassen, J. M., & Gifford, D. K. (1988). Polymorphic effect systems. *Pages 47–57 of:* the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York.

Mitchell, John C., & Plotkin, Gordon D. (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, **10**(3), 470–502. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985, ACM Press, New York.

Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of:* the *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif. Also appeared as: LFCS Report ECS-LFCS-88-86, University of Edinburgh, 1988.

Moggi, Eugenio. (1991). Notions of computation and monads. *Inf. Comput.*, **93**, 55–92.

Moggi, Eugenio, & Palumbo, Fabrizio. (1999). Monadic encapsulation of effects: A revised approach. *Electronic notes in theoretical computer science*, **26**, 119–136. Third International Workshop on Higher Order Operational Techniques in Semantics.

Odersky, Martin, Rabin, Dan, & Hudak, Paul. (1993). Call by name, assignment, and the lambda calculus. *Pages 43–56 of:* the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York.

Rabin, Dan. (1996). *Calculi for functional programming languages with assignments*. Ph.D. thesis, Yale University. Technical Report YALEU/DCS/RR-1107.

Reynolds, John C. (1978). Syntactic control of interference. *Pages 39–46 of:* the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York.

Reynolds, John C., & Plotkin, Gordon. (1993). On functors expressible in the polymorphic typed lambda calculus. *Inf. Comput.*, **105**(1), 1–29.

Riecke, Jon G. (1993). Delimiting the scope of effects. *Pages 146–155 of:* the *Conference on Functional Programming and Computer Architecture*. ACM Press, New York.

Riecke, Jon G., & Viswanathan, R. (1995). Isolating side effects in sequential languages. *Pages 1–12 of:* the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York.

Semmelroth, Miley, & Sabry, Amr. (1999). Monadic encapsulation in ML. *Pages 8–17 of:* the *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

Swarup, V., Reddy, Uday, & Ireland, E. (1991). Assignments for applicative languages. *Pages 192–214 of:* the *Conference on Functional Programming and Computer Architecture*. ACM Press, New York.

Talpin, J.-P., & Jouvelot, P. (1992a). Polymorphic type, region and effect inference. *Journal of functional programming*, **2**(3), 245–271.

Talpin, Jean-Pierre, & Jouvelot, P. (1992b). The type and effect discipline. *Pages 162–173 of:* the *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif.

Tofte, Mads. (1990). Type inference for polymorphic references. *Inf. Comput.*, **89**(1), 1–34.

Tofte, Mads, & Talpin, Jean-Pierre. (1997). Region-based memory management. *Inf. Comput.*, **132**(2), 109–176.

Wadler, Philip. (1990). Comprehending monads. *Pages 61–78 of:* the *ACM Conference on Lisp and Functional Programming*. ACM Press, New York.

Wadler, Philip. (1992). The essence of functional programming (invited talk). *Pages 1–14 of:* the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York.

Wadler, Philip. (1998). The marriage of effects and monads. *Pages 63–74 of:* the *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

Wright, Andrew K., & Felleisen, Matthias. (1994). A syntactic approach to type soundness. *Inf. Comput.*, **115**(1), 38–94.