A Logical View of Effects

Sungwoo Park and Robert Harper
Computer Science Department
Carnegie Mellon University
{gla,rwh}@cs.cmu.edu

Abstract

Despite their invaluable contribution to the programming language community, monads as a foundation for the study of effects have three problems: they make it difficult to combine effects; they enforce sequentialization of computations by the syntax; they prohibit effect-free evaluations from invoking effectful computations. Building on the judgmental formulation and the possible worlds interpretation of modal logic, we propose a logical analysis of effects based upon the view monads are not identified with effects. Our analysis leads to a language called $\lambda^{\rightarrow}_{\bigcirc}$ which distinguishes between control effects and world effects, enforces sequentialization of computations only by the semantics, and logically explains the invocation of computations from evaluations. $\lambda^{\rightarrow}_{\bigcirc}$ also serves as a unified framework for studying Haskell and ML, which have traditionally been studied separately.

1 Introduction

Motivation

Since their introduction to the programming language community, monads [23, 24] have been considered as an elegant means of structuring programs and incorporating effects into purely functional languages. An example of a functional language that makes extensive use of monads is Haskell [30]. At the program level, it provides a type class Monad to support modular programming; at the language design level, it uses monads for a modular semantics of effects and provides a built-in IO monad, which allows programmers to use effects without compromising its properties as a purely functional language. While they are a success as a tool for modular programming, monads as a foundation for the study of effects have the following three problems:

Combining effects. It is well-known that monads do not combine well with each other [14, 12, 20]. This means that although monads provide a modular way to develop semantics for individual effects, they fail to give a modular semantics when all effects are present together. Hence the identification between monads and effects makes it difficult to *combine effects* at the language design level. Haskell avoids this problem by confining all kinds of effects – mutable references, input/output, exception, concurrency, and so on – to the IO monad, but it does not provide a justification for the assumption that individual monads combine into a single monad.

Sequentialization by the syntax. Unlike effect-free/pure *evaluations*, effectful/impure *computations* are sequential by their nature. This, however, does not mean that their syntax must also be in a sequential form. Unfortunately the return and bind constructs of the monadic syntax (*e.g.*, return and >>= in Haskell) force programmers to strictly follow the sequential order of computations. This becomes increasingly inconvenient as the code grows in size.

Entering and escaping from monads. Monads allow computations to freely invoke evaluations, but not vice versa. In essence, we can neither enter monads (and initiate computations) during evaluations nor escape from monads (and return results of computations) back to evaluations, because of effects that computations may produce. In the case of Haskell, this means that we cannot write functions of type IO A -> A, which are particularly useful for benign effects (*e.g.*, accessing read-only files).

In order to overcome this limitation, Haskell provides two constructs: unsafePerformIO [34, 33] and runST [17, 18, 19]. However, unsafePerformIO is unsafe (as its name suggests) because in principle, it can destroy the distinction between evaluations and computations. runST, albeit an ingenious solution, does not have a Hindley-Milner type. It also lacks extensibility because it is specific to mutable references; moreover it initiates computations only with an empty store.

Approach

Our view is that monads are an abstraction that is more general than effects and that monads are a particular way to model effects. The first is based upon the fact that many datatypes that do not involve effects organize themselves into monads. For instance, it is easy to create a list monad by instantiating the type class Monad of Haskell. The second is based upon the observation that monads are not the only way to model effects. For instance, Nanevski [28] shows how to usefully model exceptions with comonads. Thus we are led to conclude that *monads are not identified with effects* and the study of effects is more fundamental than the study of monads. Plotkin and Power [36] present a similar view: "computational effects *determine* monads but are not identified with monads."

Then what is a suitable theory of effects? Following the propositions-as-types interpretation as an underlying principle, we propose an analysis of effects based upon *modal logic*. To apply the propositions-as-types interpretation to modal logic, we use the *judgmental formulation* of Pfenning and Davies [35], which adopts Martin-Löf's methodology of distinguishing judgments from propositions [22]. To relate modal logic to effects, we use the *possible worlds interpretation* [15], which assumes an accessibility relation between worlds and relativizes truth to worlds. Thus we give a *logical* analysis of effects based upon the view that monads are not identified with effects.

Results

Our analysis of effects has the following characteristics:

Segregation of control effects and world effects. We assume that the runtime system consists of a program and a world. A program is subject to a set of reduction rules (*e.g.*, β-reduction rule in the λ-calculus). A world is an object whose behavior is specified by the programming environment (*e.g.*, keyboard buffer). When the program undergoes a change that cannot be explained by a finite number of applications of certain "basic" reduction rules (*e.g.*, capturing and throwing continuations with respect to the "basic" β-reduction rule), we say that a *control effect* occurs. When the program interacts with the world and causes a transition to another world (*e.g.*, reading the keyboard buffer), we say that a *world effect* occurs. In this way, we distinguish between control effects and world effects.

We treat control effects and world effects in an orthogonal way. Control effects are realized by introducing reduction rules that cannot be defined in terms of the basic reduction rules. Note that whether a change in the program is a control effect or not depends on what the basic reduction rules are. For instance, continuations are usually considered as control effects, but only when their reduction rules are not accepted as the basic reduction rules. World effects are realized by specifying a world structure: empty world structure if there are no world effects, keyboard buffer and window for input/output, store for mutable references, and so on.

With the distinction between control effects and world effects, it is easy to combine effects at the language design level. We can combine different world effects by merging corresponding world structures. There is no need to explicitly combine control effects with other effects, since control effects become pervasive once we introduce their reduction rules. This means that we distinguish between effect-free evaluations and effectful computations *only with respect to world effects*.

Sequentialization only by the semantics. The judgmental formulation in our analysis leads to the use of two syntactic categories: *terms* for evaluations and *expressions* for computations. The definition of terms is derived from an ordinary truth judgment via the propositions-as-types interpretation. For the definition of expressions, we apply the propositions-as-types interpretation to *lax logic* [8]. It gives two different definitions for expressions: one with monadic constructs and another with effectful functions. The first results in a language similar to Haskell but with a separate (monadic) syntactic category for computations. The second results in a language similar to (call-by-value) ML but with a separate syntactic category for evaluations. Since these two definitions do not conflict with each other, we incorporate both into a common linguistic framework, where both Haskell-style programming and ML-style programming may coexist. Then effects can be, but need not be, structured as monads; for the same reason, computations can be, but need not be, written in a sequential style.

Logical account for invoking computations. In our analysis, the possible worlds interpretation of modal logic naturally leads to a logically motivated construct, called run, for invoking computations during evaluations. Compared with Haskell's unsafePerformIO and runST constructs, our run construct is similar in purpose but different in details: unlike unsafePerformIO, it is safe in the sense that it preserves the effect-freeness/purity of evaluations; unlike runST, it is not specific to computations for mutable references beginning with an empty store.

We present a language, called $\lambda^{\rightarrow}_{\bigcirc}$, which is based upon the analysis of effects outlined above. $\lambda^{\rightarrow}_{\bigcirc}$ provides a linguistic framework for programming languages with effects. It also serves as a unified framework for studying two languages that have traditionally been studied separately: Haskell and ML. In essence, Haskell consists of terms whereas ML consists of expressions: in

Haskell, every computation is represented as a monad and therefore there are only terms; in ML, every part of a program may produce effects and therefore there are only expressions.

A weakness of $\lambda^{\rightarrow}_{\bigcirc}$ is that it accounts for *internal* world effects, but not *external* world effects. An internal world effect is always caused by the program and is ephemeral in the sense that we can undo the change it makes to the world. An example is to allocate new references, which we can reclaim anytime. An external world effect is caused either by an external agent, affecting the program, or by the program, affecting an external agent. It is perpetual in the sense that we cannot undo the change it makes to the world. An example is to use keyboard input or to send output to a printer (*e.g.*, typing your password to a malicious program or printing it on a public printer). External world effects are difficult to model because they can make a change to the world independently of the program. Moreover the run construct can not be applied to external world effects. Thus we restrict ourselves to internal world effects in developing $\lambda^{\rightarrow}_{\bigcirc}$.

The rest of this paper is organized as follows. In Section 2, we develop our language $\lambda^{\rightarrow}_{\bigcirc}$. Section 3 shows three examples of world effects. In Section 4, we discuss control effects. In Section 5, we develop the run construct. Section 6 discusses related work. Section 7 concludes with future work.

2 Language $\lambda^{\rightarrow}_{\bigcirc}$

At its core, our analysis of effects uses the judgmental formulation and the possible worlds interpretation of modal logic. For a good introduction to the judgmental formulation, we refer the reader to [35]. It also gives an effect-free fragment of $\lambda^{\rightarrow}_{\bigcirc}$ which is a reformulation of Moggi's monadic metalanguage λ_{ml} [23, 24].

The judgmental formulation in our analysis is based upon two categorical judgments:

- *truth judgment* A *true*
- *computability judgment* A *comp*

A *true* means that A is true, and A *comp* means that A *true* holds after potentially producing some world effect. To represent proofs of the two judgments, we use two syntactic categories: *terms* M, N for truth judgments and *expressions* E, F for computability judgments. Thus we have the following correspondence under the propositions-as-types interpretation:

$$\overset{\mathcal{D}}{A\ \mathit{true}} \quad \Leftrightarrow \quad M : A \qquad\qquad \overset{\mathcal{E}}{A\ \mathit{comp}} \quad \Leftrightarrow \quad E \div A$$

That is, we represent a proof $\mathcal{D}$ of A *true* as a term M of type A, written $M : A$, and a proof $\mathcal{E}$ of A *comp* as an expression E of type A, written $E \div A$.

Since it relativizes truth to worlds, the possible worlds interpretation in our analysis requires us to incorporate some worlds into the semantics of $\lambda^{\rightarrow}_{\bigcirc}$. In our case, these worlds are precisely the same worlds that are part of the runtime system. For the type system, we annotate typing judgments with worlds ω where terms or expressions reside:

$$M \mathbin{@} \omega : A \qquad\qquad E \mathbin{@} \omega \div A$$

$M \mathbin{@} \omega : A$ means that M has type A at world ω; $E \mathbin{@} \omega \div A$ means that E has type A at world ω. For the operational semantics, we make a distinction between *evaluations* of terms, which do not involve worlds, and *computations* of expressions, which involve worlds:

$$M \hookrightarrow V \qquad\qquad E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'$$

A term evaluation $M \hookrightarrow V$ does not interact with the world where term M resides; hence the resultant value V resides at the same world. In contrast, an expression computation $E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'$ may interact with world ω where expression E resides, causing a transition to another world ω'; hence the resultant value V may not reside at the same world. Thus term evaluations are always effect-free whereas expression computations are potentially effectful.

In order to facilitate the characterization of the two categorical judgments, we also introduce a hypothetical judgment $\Gamma \vdash J$ where hypotheses Γ are a set of truth judgments and J is a categorical judgment. Under the propositions-as-types interpretation, hypothetical judgments correspond to typing judgments with typing contexts; for notational convenience, we use Γ for typing contexts as well as hypotheses:

$$\Gamma \vdash M \mathbin{@} \omega : A \qquad\qquad \Gamma \vdash E \mathbin{@} \omega \div A$$

A typing context Γ is a set of bindings $x : A$:

$$\text{typing context} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

$x : A$ in Γ means that variable x assumes a term that has type A at a given world *but may not typecheck at other worlds*. Then a term typing judgment $\Gamma \vdash M \mathbin{@} \omega : A$ means that M has type A at world ω if Γ is satisfied at the same world; similarly an expression typing judgment $\Gamma \vdash E \mathbin{@} \omega \div A$ means that E has type A at world ω if Γ is satisfied at the same world.

Below we characterize A *true* and A *comp* with hypothetical judgments and apply the propositions-as-types interpretation. We develop the type system in a natural deduction style, *i.e.*, with an introduction rule and an elimination rule for each connective or modality. Appendix shows the summary of the definition of $\lambda^{\rightarrow}_{\bigcirc}$.

2.1 Constructs for evaluations

We use the following properties of hypothetical judgments to characterize $A\ \mathit{true}$, where J can be any categorical judgment:

1. $\Gamma, A\ \mathit{true} \vdash A\ \mathit{true}$.
2. If $\Gamma \vdash A\ \mathit{true}$ and $\Gamma, A\ \mathit{true} \vdash J$, then $\Gamma \vdash J$.

The first clause expresses that we can use $A\ \mathit{true}$ as a hypothesis. The second clause expresses the substitution principle for $A\ \mathit{true}$.

In terms of term typing judgments, the first clause gives the following rule where we use variable x as a term:

$$\frac{}{\Gamma, x : A \vdash x \mathbin{@} \omega : A}\ \mathsf{Hyp}$$

The second clause with $J = C\ \mathit{true}$ gives the substitution principle for terms:

If $\Gamma \vdash M \mathbin{@} \omega : A$ *and* $\Gamma, x : A \vdash N \mathbin{@} \omega : C$,
then $\Gamma \vdash [M/x]N \mathbin{@} \omega : C$.

$[M/x]N$ denotes a capture-avoiding term substitution.

We apply the propositions-as-types interpretation to $A\ \mathit{true}$ by introducing a connective $\supset$ such that $\Gamma \vdash A \supset C\ \mathit{true}$ expresses $\Gamma, A\ \mathit{true} \vdash C\ \mathit{true}$. It gives the following introduction and elimination rules, where we use a lambda abstraction $\lambda x : A.\, M$ and a lambda application $M_1\ M_2$ as terms:

$$\frac{\Gamma, x : A \vdash M \mathbin{@} \omega : C}{\Gamma \vdash \lambda x{:}A.\, M \mathbin{@} \omega : A \supset C}\ \supset\mathsf{I}$$

$$\frac{\Gamma \vdash M_1 \mathbin{@} \omega : A \supset C \quad \Gamma \vdash M_2 \mathbin{@} \omega : A}{\Gamma \vdash M_1\ M_2 \mathbin{@} \omega : C}\ \supset\mathsf{E}$$

The term reduction rule for $\supset$ and its corresponding proof reduction are:

$$(\lambda x{:}A.\, M)\ N \ \Rightarrow_{\mathsf{term}}\ [N/x]M \qquad (\beta_\supset)$$

$$\frac{\dfrac{\Gamma, x : A \vdash M \mathbin{@} \omega : C}{\Gamma \vdash \lambda x{:}A.\, M \mathbin{@} \omega : A \supset C}\ \supset\mathsf{I} \quad \Gamma \vdash N \mathbin{@} \omega : A}{\Gamma \vdash (\lambda x{:}A.\, M)\ N \mathbin{@} \omega : C}\ \supset\mathsf{E}$$
$$\Rightarrow_{\mathsf{term}}\ \Gamma \vdash [N/x]M \mathbin{@} \omega : C$$

2.2 Constructs for computations

Unlike evaluations, computations are sequential by nature because world effects must be produced in the sequential order specified by the programmer. Therefore $A\ \mathit{comp}$ must be characterized in such a way that it expresses the sequential nature of computations. To this end, we base $A\ \mathit{comp}$ upon lax logic [8], which is indeed the logic underlying monads (see [5]):

1. If $\Gamma \vdash A\ \mathit{true}$, then $\Gamma \vdash A\ \mathit{comp}$.
2. If $\Gamma \vdash A\ \mathit{comp}$ and $\Gamma, A\ \mathit{true} \vdash C\ \mathit{comp}$, then $\Gamma \vdash C\ \mathit{comp}$.

The first clause expresses that if A is true, $A\ \mathit{true}$ holds without producing any world effect. The second clause expresses that if $A\ \mathit{true}$ holds after producing some world effect, we can use $A\ \mathit{true}$ as a hypothesis on the assumption that the world effect is implicitly attached to any subsequent judgment. Hence, once $A\ \mathit{true}$ becomes a hypothesis, we can deduce, for instance, $C\ \mathit{comp}$, but not $C\ \mathit{true}$. The second clause can also be thought of as the substitution principle for $A\ \mathit{comp}$.

In terms of expression typing judgments, the first clause means that a term of type A is also an expression of the same type:

$$\frac{\Gamma \vdash M \mathbin{@} \omega : A}{\Gamma \vdash M \mathbin{@} \omega \div A}\ \mathsf{Term}$$

The substitution principle for expressions is derived from the second clause in the characterization of $A\ \mathit{true}$ with $J = C\ \mathit{comp}$ as well as the second clause above:

If $\Gamma \vdash M \mathbin{@} \omega : A$ *and* $\Gamma, x : A \vdash E \mathbin{@} \omega \div C$,
then $\Gamma \vdash [M/x]E \mathbin{@} \omega \div C$.

If $\Gamma \vdash F \mathbin{@} \omega \div A$ *and* $\Gamma, x : A \vdash E \mathbin{@} \omega \div C$,
then $\Gamma \vdash \langle F/x\rangle E \mathbin{@} \omega \div C$.

Unlike a term substitution $[M/x]E$ which analyzes the structure of E, an *expression substitution* $\langle F/x\rangle E$ analyzes the structure of F instead of E. This is because $\langle F/x\rangle E$ is intended to ensure that both F and E are computed exactly once and in that order. Intuitively we must not replicate F within E (at those places where x occurs), which would result in computing F multiple times; instead we must conceptually replicate E within F (at those places where the computation of F is finished) so that we end up computing both F and E only once. In this sense, an expression substitution $\langle F/x\rangle E$ substitutes not F into E, but E into F. We will give the definition of $\langle F/x\rangle E$ after introducing all expression constructs.

We apply the propositions-as-types interpretation to $A\ \mathit{comp}$ in two ways. First we internalize $A\ \mathit{comp}$ with a modality $\bigcirc$ so that $\Gamma \vdash \bigcirc A\ \mathit{true}$ expresses $\Gamma \vdash A\ \mathit{comp}$. Second we introduce a connective $\rightarrow$ such that $\Gamma \vdash A \rightarrow C\ \mathit{true}$ expresses $\Gamma, A\ \mathit{true} \vdash C\ \mathit{comp}$ (in the same way that we introduce the connective $\supset$).

In the first case, the introduction and elimination rules use a *computation term* $\mathsf{cmp}\ E$ and a *bind expression* $\mathsf{letcmp}\ x \lhd M\ \mathsf{in}\ E$:

$$\frac{\Gamma \vdash E \mathbin{@} \omega \div A}{\Gamma \vdash \mathsf{cmp}\ E \mathbin{@} \omega : \bigcirc A}\ \bigcirc\mathsf{I}$$

$$\frac{\Gamma \vdash M \mathbin{@} \omega : \bigcirc A \quad \Gamma, x : A \vdash E \mathbin{@} \omega \div C}{\Gamma \vdash \mathsf{letcmp}\ x \lhd M\ \mathsf{in}\ E \mathbin{@} \omega \div C}\ \bigcirc\mathsf{E}$$

The expression reduction rule for $\bigcirc$ and its corresponding proof reduction are:

$$\mathsf{letcmp}\ x \lhd \mathsf{cmp}\ E\ \mathsf{in}\ F \quad \Rightarrow_{\mathsf{exp}} \quad \langle E/x\rangle F \qquad (\beta_{\bigcirc})$$

$$\frac{\dfrac{\Gamma \vdash E \mathbin{@} \omega \div A}{\Gamma \vdash \mathsf{cmp}\ E \mathbin{@} \omega : \bigcirc A}\ \bigcirc\mathsf{I} \quad \Gamma, x : A \vdash F \mathbin{@} \omega \div C}{\Gamma \vdash \mathsf{letcmp}\ x \lhd \mathsf{cmp}\ E\ \mathsf{in}\ F \mathbin{@} \omega \div C}\ \bigcirc\mathsf{E}$$
$$\Rightarrow_{\mathsf{exp}} \ \Gamma \vdash \langle E/x\rangle F \mathbin{@} \omega \div C$$

$\mathsf{cmp}\ E$ denotes the computation of E, but it does not actually compute E. In this sense, we say that $\mathsf{cmp}\ E$ *encapsulates* the computation of expression E. $\mathsf{letcmp}\ x \lhd M\ \mathsf{in}\ E$ enables us to sequence two computations (if M evaluates to a computation term). These two constructs can be thought of as monadic constructs, since the modality $\bigcirc$ forms a monad (see [35]).

In the second case, the introduction and elimination rules use an *effectful function* $\hat{\lambda x} : A.\, E$ as a term and an *effectful application* $E_1 \hat{}\, E_2$ as an expression:

$$\frac{\Gamma, x : A \vdash E \mathbin{@} \omega \div C}{\Gamma \vdash \hat{\lambda x} : A.\, E \mathbin{@} \omega : A \to C}\ \to\mathsf{I}$$

$$\frac{\Gamma \vdash E_1 \mathbin{@} \omega \div A \to C \quad \Gamma \vdash E_2 \mathbin{@} \omega \div A}{\Gamma \vdash E_1 \hat{}\, E_2 \mathbin{@} \omega \div C}\ \to\mathsf{E}$$

The expression reduction rule for $\to$ and its corresponding proof reduction are:

$$(\hat{\lambda x} : A.\, F) \hat{}\, E \quad \Rightarrow_{\mathsf{exp}} \quad \langle E/x\rangle F \qquad (\beta_{\to})$$

$$\frac{\dfrac{\dfrac{\Gamma, x : A \vdash F \mathbin{@} \omega \div C}{\Gamma \vdash \hat{\lambda x} : A.\, F \mathbin{@} \omega : A \to C}\ \to\mathsf{I}}{\Gamma \vdash \hat{\lambda x} : A.\, F \mathbin{@} \omega \div A \to C}\ \mathsf{Term} \quad \Gamma \vdash E \mathbin{@} \omega \div A}{\Gamma \vdash (\hat{\lambda x} : A.\, F) \hat{}\, E \mathbin{@} \omega \div C}\ \to\mathsf{E}$$
$$\Rightarrow_{\mathsf{exp}} \ \Gamma \vdash \langle E/x\rangle F \mathbin{@} \omega \div C$$

The modality $\bigcirc$ and the connective $\to$ are both defined without relying on any other connective. Hence their definitions do not conflict with each other, and we use all the above constructs in $\lambda^{\to}_{\bigcirc}$. We also use two additional expression constructs, *instruction* I and *suspension expression* $\{E/x\}F$, which are explained below.

Instructions produce world effects by directly interacting with worlds; without them, there is no way to produce world effects. As an interface to worlds, they are provided by the programming environment. As an example, consider an instruction $\mathsf{new}\ M$ for allocating new references. It causes a change to the store, producing a world effect, and returns a reference. We refer to those objects originating from worlds as *world terms* W (*e.g.*, references). Since they cannot be decomposed into ordinary terms, world terms are assumed to be values and are given special *world term types* $\mathcal{W}$ (*e.g.*, reference type $\mathsf{ref}\ A$ for references).

We assume that the programming environment provides an expression typing rule $\Gamma \vdash I \mathbin{@} \omega \div A$ for each instruction I, and a term typing rule $\Gamma \vdash W \mathbin{@} \omega : \mathcal{W}$ for each world term W. The type of an instruction depends only on the types of its arguments, if any; if it has no argument, its type is fixed. The type of a world term may depend on the world where it resides. For instance, the type of a reference cannot be determined without a store. This is why we need worlds in typing judgments.

A suspension expression $\{E/x\}F$ is similar to an expression substitution $\langle E/x\rangle F$, but suspends the substitution because we do not know yet where the computation is finished within E and hence cannot substitute F into E (not E into F). Consider the following expression reductions:

$$\begin{array}{rcl} \mathsf{letcmp}\ x \lhd \mathsf{cmp}\ I\ \mathsf{in}\ E & \Rightarrow_{\beta} & \langle I/x\rangle E \\ (\hat{\lambda x} : A.\, E) \hat{}\, (F_1 \hat{}\, F_2) & \Rightarrow_{\beta} & \langle F_1 \hat{}\, F_2/x\rangle E \end{array}$$

In the first case, we cannot substitute E into I because we do not know yet the result of computing I. Hence we must suspend the substitution until I interacts with the world and replaces itself by a value:

$$\langle I/x\rangle E \quad = \quad \{I/x\}E$$

In the second case, we do not know yet where the computation is finished within $F_1 \hat{}\, F_2$. Hence we can proceed to compute F_1, but the rest of the computation must be suspended:

$$\langle F_1 \hat{}\, F_2/x\rangle E \quad = \quad \langle F_1/f\rangle\{f \hat{}\, F_2/x\}E \quad \textit{fresh variable } f$$

As a special case, if F_1 is already an effectful function $\hat{\lambda y} : C.\, F$, we can analyze F to determine where the computation of $F_1 \hat{}\, F_2$ is finished. This implies that we can resume the substitution in $\{(\hat{\lambda y} : C.\, F) \hat{}\, F_2/x\}E$:

$$\{(\hat{\lambda y} : C.\, F) \hat{}\, F_2/x\}E \quad \Rightarrow_{\mathsf{exp}} \quad \langle\langle F_2/y\rangle F/x\rangle E \qquad (\beta_{\{\}})$$

The typing rule for suspension expressions is directly obtained from the substitution principle for expressions (*i.e.*, $\{E/x\}F$ is conceptually a suspended $\langle E/x\rangle F$):

$$\frac{\Gamma \vdash E \mathbin{@} \omega \div A \quad \Gamma, x : A \vdash F \mathbin{@} \omega \div C}{\Gamma \vdash \{E/x\}F \mathbin{@} \omega \div C}\ \mathsf{Sus}$$

With the above typing rule, we can show that the rule $\beta_{\{\}}$ has a corresponding proof reduction. Note that suspension expressions are by-products of expression reductions and are not available to programmers.

The remaining three cases of expression substitutions are as follows:

$$\begin{array}{rcl} \langle M/x\rangle F & = & [M/x]F \\ \langle \mathsf{letcmp}\ y \lhd M\ \mathsf{in}\ E/x\rangle F & = & \mathsf{letcmp}\ y \lhd M\ \mathsf{in}\ \langle E/x\rangle F \\ \langle\{E_1/y\}E_2/x\rangle F & = & \{E_1/y\}\langle E_2/x\rangle F \end{array}$$

The above definition of expression substitutions $\langle E/x\rangle F$ obeys the substitution principle for expressions; we can prove this by induction on the structure of E (not F).

2.3 Operational semantics

We assume that the programming environment provides a rule $I \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'$ for each instruction I, where we allow $\omega = \omega'$ and V is not necessarily a world term; we say that instruction I *executes* to value V. We require that every instruction be type-preserving: if $I \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'$ and $\cdot \vdash I \mathbin{@} \omega \div A$, then $\cdot \vdash V \mathbin{@} \omega' : A$.

The operational semantics is based upon term and expression reduction rules ($\beta_{\supset}$, $\beta_{\bigcirc}$, $\beta_{\rightarrow}$, and $\beta_{\{\}}$) in that term evaluations and expression computations represent sequences of term and expression reductions:

$$\frac{V ::= \lambda x{:}A.\, M \mid \mathsf{cmp}\ E \mid \hat{\lambda} x{:}A.\, E \mid W}{V \hookrightarrow V}\ \mathit{Val}$$

$$\frac{M_1 \hookrightarrow \lambda x{:}A.\, M \quad [M_2/x]M \hookrightarrow V}{M_1\ M_2 \hookrightarrow V}\ \mathit{LApp}$$

$$\frac{M \hookrightarrow V}{M \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega}\ \mathit{Term}$$

$$\frac{M \hookrightarrow \mathsf{cmp}\ F \quad \langle F/x\rangle E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'}{\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'}\ \mathit{Letcmp}$$

$$\frac{E_1 \mathbin{@} \omega \rightharpoonup \hat{\lambda} x{:}A.\, F \mathbin{@} \omega' \quad \langle E_2/x\rangle F \mathbin{@} \omega' \rightharpoonup V \mathbin{@} \omega''}{E_1 \hat{}\, E_2 \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega''}\ \mathit{EApp}$$

$$\frac{I \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega' \quad [V/x]E \mathbin{@} \omega' \rightharpoonup V' \mathbin{@} \omega''}{\{I/x\}E \mathbin{@} \omega \rightharpoonup V' \mathbin{@} \omega''}\ \mathit{Sus}^I$$

$$\frac{M \hookrightarrow \hat{\lambda} y{:}A.\, E' \quad \langle\langle E/y\rangle E'/x\rangle F \mathbin{@} \omega \rightharpoonup V' \mathbin{@} \omega'}{\{M \hat{}\, E/x\}F \mathbin{@} \omega \rightharpoonup V' \mathbin{@} \omega'}\ \mathit{Sus}^{\hat{}}$$

The evaluation rule *LApp* shows that we use call-by-name for evaluating lambda applications. We could equally use call-by-value, call-by-need, or even unspecified evaluation order as in Haskell. All these choices are acceptable because term evaluations are effect-free and different evaluation orders do not change their results. The computation rule *EApp* can be thought of as using call-by-value, since $\langle E_2/x\rangle F$ forces the computation of E_2 to precede the computation of F. We exploit the fact that a well-typed closed suspension expression is either $\{I/x\}E$ or $\{M \hat{}\, E/x\}F$, and provide two specialized computation rules.

At this point, we cannot prove the type preservation property. The reason is that the typing rules $\bigcirc\mathsf{E}$, $\rightarrow\mathsf{E}$, and Sus do not accurately reflect the operational behavior of $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$, $E_1 \hat{}\, E_2$, and $\{E/x\}F$, respectively. For instance, while we typecheck E at the same world ω that we typecheck $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$, its computation may take place at a different world ω' (*e.g.*, if $M = \mathsf{cmp}\ I$). If, however, the type of E remains the same at ω', the typing rule $\bigcirc\mathsf{E}$ becomes safe to use. This can be accomplished by requiring that an instruction execution does not affect types of existing terms and expressions. We formalize this requirement with an *accessibility relation* between worlds, thereby completing the possible worlds interpretation in our analysis of effects.

We say that a world ω' is accessible from another world ω if there exists an instruction that causes a transition from ω to ω' when executed. Formally we write $\omega \leq \omega'$ if there exists an instruction I such that $I \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'$ for some value V. We write $\leq^*$ for the reflexive and transitive closure of $\leq$.

The accessibility relation $\leq$ is monotonic if a transition between worlds preserves types of world terms:

Definition 2.1.
$\leq$ is monotonic if $\omega \leq \omega'$ means that for every world term W, $\Gamma \vdash W \mathbin{@} \omega : \mathcal{W}$ implies $\Gamma \vdash W \mathbin{@} \omega' : \mathcal{W}$.

It is easy to show that if $\leq$ is monotonic, a transition between worlds preserves types of all terms and expressions. We require that every instruction maintain the monotonicity of $\leq$.

We can now prove the type preservation property:

Theorem 2.2 (Type preservation).
If $M \hookrightarrow V$ and $\cdot \vdash M \mathbin{@} \omega : A$, then $\cdot \vdash V \mathbin{@} \omega : A$.
If $E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'$ and $\cdot \vdash E \mathbin{@} \omega \div A$, then $\cdot \vdash V \mathbin{@} \omega' \div A$ and $\omega \leq^* \omega'$.

Since expressions may produce world effects, we do not allow expressions to be converted into terms, while we can always lift terms to expressions. Therefore we define a program as a closed expression E that typechecks at a certain initial world $\omega_{initial}$, *i.e.*, $\cdot \vdash E \mathbin{@} \omega_{initial} \div A$. We choose $\omega_{initial}$ according to the world structure being employed. To run a program E, we compute it at $\omega_{initial}$.

2.4 $\lambda^{\rightarrow}_{\bigcirc}$ = Haskell + ML

Since the modality $\bigcirc$ and the connective $\rightarrow$ are independent of each other, $\lambda^{\rightarrow}_{\bigcirc}$ without either $\bigcirc$ or $\rightarrow$ is still a complete language with constructs for computations. The two sublanguages of $\lambda^{\rightarrow}_{\bigcirc}$ have the same expressive power (because $\bigcirc$ and $\rightarrow$ can simulate each other using $\supset$), but they allow us to structure effects and write computations in different ways.

$\lambda^{\rightarrow}_{\bigcirc}$ without $\rightarrow$, written $\lambda_{\bigcirc}$, is similar to Haskell if terms are regarded as its primary syntactic category: terms are effect-free and computations are written in a monadic syntax. The difference is that computations in $\lambda_{\bigcirc}$ use a separate syntactic category, namely expressions, whereas in Haskell, computations are represented as monads and therefore there are only terms

(*e.g.*, $\mathsf{return}\ M$ in Haskell corresponds to a term $\mathsf{cmp}\ M$ in $\lambda_{\bigcirc}$, not to an expression M).

$\lambda^{\rightarrow}_{\bigcirc}$ without $\bigcirc$, written $\lambda^{\rightarrow}$, is similar to ML if expressions are regarded as its primary syntactic category: any expression may produce effects and effectful applications use call-by-value. The difference is that evaluations in $\lambda^{\rightarrow}$ use a separate syntactic category, namely terms, whereas in ML, there is no separate syntactic category for evaluations.

As a combination of $\lambda_{\bigcirc}$ and $\lambda^{\rightarrow}$, therefore, $\lambda^{\rightarrow}_{\bigcirc}$ serves as a unified framework where both Haskell-style programming and ML-style programming may coexist. If you like Haskell, you can begin with terms, structuring effects as monads and writing computations in a sequential style; if you like ML, you can begin with expressions, allowing effects in any expression and writing computations in an imperative style, and also enjoying evaluations when necessary. If you like both, you have the freedom to begin with either syntactic category.

3 Examples of World Effects

In order to implement a specific notion of world effect in $\lambda^{\rightarrow}_{\bigcirc}$, we specify a world structure and provide instructions to interact with worlds. In this section, we discuss three specific notions of world effect.

3.1 Probabilistic computations

We model a probabilistic computation as a computation that returns a value after consuming real numbers drawn independently from a uniform distribution over $(0.0, 1.0]$. A real number r is a world term of type real. A world, the source of probabilistic choices, is represented by an infinite sequence of real numbers drawn independently from a uniform distribution over $(0.0, 1.0]$. We use an instruction random for consuming the first real number of a given world:

$$\text{world}\quad \omega \quad ::= \quad r_1 r_2 \cdots r_i \cdots \quad \textit{where } r_i \in (0.0, 1.0]$$

$$\frac{}{\Gamma \vdash r \mathbin{@} \omega : \mathsf{real}}\ \mathsf{Real} \qquad \frac{}{\Gamma \vdash \mathsf{random} \mathbin{@} \omega \div \mathsf{real}}\ \mathsf{Random}$$

$$\frac{}{\mathsf{random} \mathbin{@} r_1 r_2 r_3 \cdots \rightharpoonup r_1 \mathbin{@} r_2 r_3 \cdots}\ \textit{Random}$$

Since a world does not affect types of world terms, the monotonicity of $\leq$ is preserved. We can use any world as an initial world.

3.2 Sequential file input/output

We model sequential file input/output with a computation that consumes an input character stream *is* and outputs to an output character stream *os*, where a character is a world term of type char. We assume that *is* is read-only and *os* is not used by an external agent so that the computation produces only internal world effects. We use two instructions: $\mathsf{read\_c}$ for reading a character from the input stream and $\mathsf{write\_c}\ M$ for writing a character to the output stream:

$$\text{world}\quad \omega \quad ::= \quad (is, os)$$
$$is \quad ::= \quad c_1 c_2 c_3 \cdots \qquad os \quad ::= \quad \mathsf{nil} \mid c :: os$$

$$\frac{}{\Gamma \vdash c \mathbin{@} \omega : \mathsf{char}}\ \mathsf{Char} \qquad \frac{}{\Gamma \vdash \mathsf{read\_c} \mathbin{@} \omega \div \mathsf{char}}\ \mathsf{Read\_c}$$

$$\frac{\Gamma \vdash M \mathbin{@} \omega : \mathsf{char}}{\Gamma \vdash \mathsf{write\_c}\ M \mathbin{@} \omega \div \mathsf{char}}\ \mathsf{Write\_c}$$

$$\frac{}{\mathsf{read\_c} \mathbin{@} (c_1 c_2 c_3 \cdots, os) \rightharpoonup c_1 \mathbin{@} (c_2 c_3 \cdots, os)}\ \textit{Read\_c}$$

$$\frac{M \hookrightarrow c}{\mathsf{write\_c}\ M \mathbin{@} (is, os) \rightharpoonup c \mathbin{@} (is, c :: os)}\ \textit{Write\_c}$$

Since a world does not affect types of world terms, the monotonicity of $\leq$ is preserved. We use an empty output character stream in an initial world.

3.3 Mutable references

Probabilistic computations and input/output are easy to formulate because worlds do not affect types of world terms. Mutable references, however, require us to introduce world terms whose type depends on worlds, namely references. Now the monotonicity of $\leq$ holds only if values written to a reference l match the type specified by l. We use a reference type $\mathsf{ref}\ A$ for references, and define a store σ as a collection of pairs of a reference and a value:

$$\text{store}\quad \sigma \quad ::= \quad \cdot \mid \sigma[l \mapsto V]$$
$$\begin{array}{lcll} \sigma[l \mapsto V] & = & \sigma, [l \mapsto V] & \textit{if} \quad [l \mapsto V'] \notin \sigma \\ \sigma[l \mapsto V] & = & (\sigma - [l \mapsto V']), [l \mapsto V] & \textit{if} \quad [l \mapsto V'] \in \sigma \end{array}$$

The definition of $\sigma[l \mapsto V]$ implies that references in a store are all distinct. We introduce a reference typing context Γ_{ref} mapping references to reference types:

$$\text{reference typing context}\quad \Gamma_{ref} \quad ::= \quad \cdot \mid \Gamma_{ref}, l : \mathsf{ref}\ A$$

By a notational abuse, we regard reference typing contexts as a special case of ordinary typing contexts. In the presence of a reference typing context, we typecheck references in the same way as variables, but without using $@\ \omega$ in the typing judgment:

$$\frac{l : \mathsf{ref}\ A \in \Gamma_{ref}}{\Gamma, \Gamma_{ref} \vdash l : \mathsf{ref}\ A}\ \mathsf{Ref}_{\mathsf{var}}$$

We say that a store σ is typable if there exists a reference typing context Γ_{ref} such that $[l \mapsto V] \in \sigma$ implies $\Gamma_{ref} \vdash V : A$ and $l : \mathsf{ref}\ A \in \Gamma_{ref}$; the typing rules for $\Gamma, \Gamma_{ref} \vdash M : A$ and $\Gamma, \Gamma_{ref} \vdash E \div A$ are derived from the ordinary typing rules by removing $@\ \omega$. We write $\models \sigma : \Gamma_{ref}$ if store σ is typable under a reference

typing context Γ_{ref}. We define a world as a typable store.

The typing rule for references derives a reference typing context from a given world:

$$\frac{\models \omega : \Gamma_{ref} \quad l : \mathsf{ref}\ A \in \Gamma_{ref}}{\Gamma \vdash l \mathbin{@} \omega : \mathsf{ref}\ A}\ \mathsf{Ref}$$

We use three instructions: $\mathsf{new}\ M$ for initializing a fresh reference, $\mathsf{read}\ M$ for reading the contents of a store, and $\mathsf{write}\ M\ M$ for updating a store. Note that reading the contents of a store is considered to be a world effect.

$$\frac{\Gamma \vdash M \mathbin{@} \omega : A}{\Gamma \vdash \mathsf{new}\ M \mathbin{@} \omega \div \mathsf{ref}\ A}\ \mathsf{New} \qquad \frac{\Gamma \vdash M \mathbin{@} \omega : \mathsf{ref}\ A}{\Gamma \vdash \mathsf{read}\ M \mathbin{@} \omega \div A}\ \mathsf{Read}$$

$$\frac{\Gamma \vdash M \mathbin{@} \omega : \mathsf{ref}\ A \quad \Gamma \vdash N \mathbin{@} \omega : A}{\Gamma \vdash \mathsf{write}\ M\ N \mathbin{@} \omega \div A}\ \mathsf{Write}$$

$$\frac{M \hookrightarrow V \quad \textit{fresh } l \textit{ such that } [l \mapsto V'] \notin \omega}{\mathsf{new}\ M \mathbin{@} \omega \rightharpoonup l \mathbin{@} \omega[l \mapsto V]}\ \textit{New}$$

$$\frac{M \hookrightarrow l \quad [l \mapsto V] \in \omega}{\mathsf{read}\ M \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega}\ \textit{Read}$$

$$\frac{M \hookrightarrow l \quad N \hookrightarrow V \quad [l \mapsto V'] \in \omega}{\mathsf{write}\ M\ N \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega[l \mapsto V]}\ \textit{Write}$$

Proposition 3.1 shows that the three instructions are type-preserving. We use the same idea in its proof to show that the monotonicity of $\leq$ is maintained.

Proposition 3.1.

If $\mathsf{new}\ M \mathbin{@} \omega \rightharpoonup l \mathbin{@} \omega[l \mapsto V]$ *and* $\cdot \vdash \mathsf{new}\ M \mathbin{@} \omega \div \mathsf{ref}\ A$, *then* $\cdot \vdash l \mathbin{@} \omega[l \mapsto V] : \mathsf{ref}\ A$.

If $\mathsf{read}\ M \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega$ *and* $\cdot \vdash \mathsf{read}\ M \mathbin{@} \omega \div A$, *then* $\cdot \vdash V \mathbin{@} \omega : A$.

If $\mathsf{write}\ M\ N \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega[l \mapsto V]$ *and* $\cdot \vdash \mathsf{write}\ M\ N \mathbin{@} \omega \div A$, *then* $\cdot \vdash V \mathbin{@} \omega[l \mapsto V] : A$.

In order to maintain the monotonicity of $\leq$, all references must be persistent because once we deallocate a reference, we cannot determine its type any longer. This means that we cannot use an instruction for deallocating references (*e.g.*, $\mathsf{delete}\ M$). We use an empty store as an initial world.

3.4 Supporting multiple notions of world effect

Since a world structure realizes a specific notion of world effect and instructions provide an interface to worlds, we can support multiple notions of world effect by combining individual world structures and letting each instruction interact with its relevant part of the world. For instance, we can use all the instructions above if a world consists of three sub-worlds: an infinite sequence of real numbers, input/output streams, and a typable store. This is the way we combine world effects at the language design level.

4 Control Effects

So far, we have restricted ourselves to world effects, *i.e.*, transitions between worlds. $\lambda^{\rightarrow}_{\bigcirc}$ is designed in such a way that world effects are confined to expressions so that terms are free of world effects. When we extend $\lambda^{\rightarrow}_{\bigcirc}$ with control effects, however, it is not immediately clear which syntactic category should be permitted to produce control effects. On one hand, we could choose to confine control effects in expressions so that terms remain free of effects. Then the distinction between effect-free evaluations and effectful computations is drawn in a conventional sense. On the other hand, in order to develop $\lambda^{\rightarrow}_{\bigcirc}$ into a practical programming language, it is highly desirable to allow control effects in terms. For instance, exceptions for terms would be an easy way to handle division by zero or pattern-match failures occurring during evaluations. At the same time, however, exceptions for expressions are also useful for those instructions whose execution does not always succeed.

We hold the view that expressions are in principle a syntactic category specialized for world effects, and allow control effects *both in terms and in expressions*. The decision does not prevent us from developing control effects orthogonally to world effects, since control effects are realized with reduction rules wheras world effects are realized with world structures. In fact, there is no reason to confine control effects only in one syntactic category, since the concept of control effect is relative to what the "basic" reduction rules are anyway.

Although control effects do not conflict with world effects, they may change the meaning of *A true* or *A comp*. As an example, consider continuations for terms; for the type system and the operational semantics, we refer the reader to the literature (*e.g.*, [11]) as they are already well developed. From a logical perspective, continuations for terms change the meaning of *A true* from intuitionistic truth to classical truth [10]. This, however, does not mean that we have to change the definition of expressions accordingly, since in our formulation of lax logic, the definition of *A comp* is not subject to a particular definition of *A true*. In other words, even if we change the meaning of *A true*, the same definition of *A comp* remains valid with respect to the new definition of *A true*; therefore the same definition of expressions remains valid with respect to the new definition of terms.

5 run Construct

Haskell provides the runST construct [17, 18, 19] designed to prevent interferences between state transformers. Conceptually it is a term construct which initiates a computation, and thus provides a way to convert expressions into terms. We wish to augment $\lambda^{\rightarrow}_{\bigcirc}$ with a

similar term construct run:

$$\text{term} \quad M \quad ::= \quad \cdots \mid \mathsf{run}\ \langle E \rangle$$

The following evaluation rule shows how the run construct works:

$$\frac{E \mathbin{@} \omega_{initial} \rightharpoonup V \mathbin{@} \omega_{final}}{\mathsf{run}\ \langle E \rangle \hookrightarrow V}\ \mathit{Run}$$

That is, we create an initial world $\omega_{initial}$, compute E at $\omega_{initial}$ to obtain a value V, and use V as the result of evaluating run $\langle E \rangle$. The initial world $\omega_{initial}$ is either fixed or determined by the current world where the evaluation of run $\langle E \rangle$ is taking place. Since run is a term construct, we discard the final world ω_{final} so that no world effects produced during the computation of E affect the current world. Now we can compute E by evaluating run $\langle E \rangle$, and therefore a program can be defined as a closed term instead of a closed expression.

It is, however, not trivial to design the run construct. An immediate problem is that depending on the choice for the initial world, the same run construct may return different results (*e.g.*, in probabilistic computations). This would mean that we cannot use the equational theory for terms despite the fact that terms are free of world effects. Therefore we require that all instances of the same run construct use the same initial world.

Another problem is that the rule *Run* is not safe to use because of those world terms whose type depends on the world where they reside. That is, E may not typecheck at $\omega_{initial}$ if it contains such world terms and $\omega_{initial}$ is not accessible from the current world. Neither is V guaranteed to typecheck at the current world for a similar reason. Therefore the type system must ensure that the evaluation of the run construct never delivers a term or expression to another world where it does not typecheck.

As a special case, if all world terms are *globally valid*, *i.e.*, typecheck at every world irrespective of the accessibility relation $\leq$ (*e.g.*, real numbers and characters), we can use the following typing rule:

$$\frac{\Gamma \vdash E \mathbin{@} \omega \div A}{\Gamma \vdash \mathsf{run}\ \langle E \rangle \mathbin{@} \omega : A}\ \mathsf{Run}_0$$

A general approach, in particular for references, is to prove that an argument E to the run construct is globally valid and also computes to a globally valid value. Albeit conservative, it imposes no further restriction on the choice of the initial world for E and allows the type system to disregard the final world resulting from the computation of E. Below we develop a type system based upon this approach.

5.1 Global computability judgment

The run construct safely converts expressions into terms. It does not, however, allow every expression to be converted; if this were the case, there would be no need at all to distinguish between terms and expressions. Therefore we need to isolate those expressions that we allow to be converted into terms.

To this end, we introduce a *global computability judgment* A *gcomp*. It means that A *true* holds *at every world* after potentially producing some world effect. Since A *gcomp* is a special case of A *comp* that yields A *true* holding at every world, we can use as its proof an expression that computes to a globally valid value. If such an expression itself is also computable at every world, we can convert it into a value at any world $\omega_{current}$ as follows: first we choose an initial world $\omega_{initial}$; next we compute it at $\omega_{initial}$; finally we transfer the result back to $\omega_{current}$. It is such expressions that we allow to be converted into terms by the run construct. This in turn means that A *gcomp* must be characterized in such a way that if it holds at every world, we can deduce A *true*.

5.1.1 Logic for global computability

To begin with, we introduce a derived judgment called *global truth judgment* A *global*, which is defined as $\cdot \vdash A$ *true* and means that A *true* holds *at every world*. We allow global truth judgments as hypotheses in a hypothetical judgment. For the sake of visual clarity, we use hypothetical judgments of the form $\Delta; \Gamma \vdash J$ where we distinguish global truth judgments as Δ. We do not use A *global* for the judgment J because we can always prove $\Delta; \Gamma \vdash A$ *global* indirectly by proving $\Delta; \cdot \vdash A$ *true*.

We characterize A *gcomp* in a similar way to A *comp*, except that we use global truth judgments in place of ordinary truth judgments and that there is an additional clause relating A *gcomp* back to A *true*:

1. If $\Delta; \cdot \vdash A$ *true*, then $\Delta; \Gamma \vdash A$ *gcomp*.
2. If $\Delta; \Gamma \vdash A$ *gcomp* and Δ, A *global*$; \Gamma \vdash J$, then $\Delta; \Gamma \vdash J$ where J is either C *comp* or C *gcomp*.
3. If $\Delta; \cdot \vdash A$ *gcomp*, then $\Delta; \Gamma \vdash A$ *true*.

In the first clause, we effectively assume A *global* because $\Delta; \cdot \vdash A$ *true* proves A *global*. The second clause can be thought of as the substitution principle for A *gcomp*. The third clause expresses that if A *gcomp* holds at every world, A *true* holds. Note that it provides a way to prove not A *gcomp* but A *true*. As suggested earlier, it is the third clause that accounts for the behavior of the run construct.

Since A *gcomp* is defined as a special case of A *comp*. we revise the definition of A *comp* accordingly:

1. If $\Delta;\Gamma \vdash A\ \mathit{true}$, then $\Delta;\Gamma \vdash A\ \mathit{comp}$.
2. If $\Delta;\Gamma \vdash A\ \mathit{comp}$ and $\Delta;\Gamma, A\ \mathit{true} \vdash J$, then $\Delta;\Gamma \vdash J$ where J is either $C\ \mathit{comp}$ or $C\ \mathit{gcomp}$.

As with $A\ \mathit{comp}$, there are two ways to apply the propositions-as-types interpretation to $A\ \mathit{gcomp}$. We present the approach based upon internalizing $A\ \mathit{gcomp}$ with a modality $\odot$; the other approach is analogous to developing the connective $\rightarrow$. For the sake of brevity, we do not consider $\hat{\lambda x{:}A.\,E}$, $E_1\,\hat{}\,E_2$, I, and $\{E/x\}F$.

5.1.2 Syntax and type system

The syntax for $A\ \mathit{gcomp}$ is as follows:

type	A	::=	$\cdots \mid \odot A$
term	M	::=	$\cdots \mid u \mid \mathsf{gcmp}\ E$
expression	E	::=	$\cdots \mid \mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E$
value	V	::=	$\cdots \mid \mathsf{gcmp}\ E$

u is called a *global variable* and holds a globally valid term. $\mathsf{gcmp}\ E$ is called a *global computation term*, and encapsulates the computation of an expression that returns a globally valid value. $\mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E$ is similar to $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$ except that M evaluates to $\mathsf{gcmp}\ F$.

The new form of hypothetical judgment requires that typing judgments include a *global typing context*, corresponding to global truth judgments, as well as an ordinary typing context. In order to use expressions as proofs of $A\ \mathit{gcomp}$, we introduce *global expression typing judgments*. Since we cannot assume a specific world when proving $A\ \mathit{global}$ (we can assume a specific world when proving $A\ \mathit{gcomp}$), we also need typing judgments that do not involve worlds:

$$\text{global typing context}\quad \Delta ::= \cdot \mid \Delta, u :: A$$

term typing judgment	$\Delta;\Gamma \vdash M \mathbin{@} \omega : A$
expression typing judgment	$\Delta;\Gamma \vdash E \mathbin{@} \omega \div A$
global expr. typing judgment	$\Delta;\Gamma \vdash E \mathbin{@} \omega \doteqdot A$
world-free typing judgments	$\Delta;\Gamma \vdash M : A$
	$\Delta;\Gamma \vdash E \div A$
	$\Delta;\Gamma \vdash E \doteqdot A$

$u :: A$ in Δ means that u assumes a globally valid term of type A. $\Delta;\Gamma \vdash M \mathbin{@} \omega : A$ means that M has type A at world ω if both Δ and Γ are satisfied at the same world. $\Delta;\Gamma \vdash M : A$ means that M has type A *at an arbitrary world where both* Δ *and* Γ *are satisfied*. In particular, $\Delta;\cdot \vdash M : A$ proves that M is globally valid. We interpret the expression typing judgments in a similar way.

A global expression typing judgment proves $A\ \mathit{gcomp}$. $\Delta;\Gamma \vdash E \mathbin{@} \omega \doteqdot A$ means that E computes at world ω to a globally valid value of type A if both Δ and Γ are satisfied at the same world. $\Delta;\Gamma \vdash E \doteqdot A$ means that E computes to a globally valid value of type A *at an arbitrary world where both* Δ *and* Γ *are satisfied*. In particular, $\Delta;\cdot \vdash E \doteqdot A$ proves that E is computable at every world and also computes to a globally valid value.

The new typing rules for the judgments with worlds are as follows:

$$\frac{\Delta;\cdot \vdash M : A}{\Delta;\Gamma \vdash M \mathbin{@} \omega \doteqdot A}\ \mathsf{GTerm} \qquad \frac{\Delta;\Gamma \vdash E \mathbin{@} \omega \doteqdot A}{\Delta;\Gamma \vdash \mathsf{gcmp}\ E \mathbin{@} \omega : \odot A}\ \odot\mathsf{I}$$

$$\frac{\Delta;\Gamma \vdash M \mathbin{@} \omega : \odot A \quad \Delta, u :: A;\Gamma \vdash E \mathbin{@} \omega \doteqdot C}{\Delta;\Gamma \vdash \mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E \mathbin{@} \omega \doteqdot C}\ \odot\mathsf{E}$$

$$\frac{\Delta;\Gamma \vdash M \mathbin{@} \omega : \odot A \quad \Delta, u :: A;\Gamma \vdash E \mathbin{@} \omega \div C}{\Delta;\Gamma \vdash \mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E \mathbin{@} \omega \div C}\ \odot\mathsf{E}_{\div}$$

$$\frac{\Delta;\cdot \vdash E \doteqdot A}{\Delta;\Gamma \vdash \mathsf{run}\ \langle E\rangle \mathbin{@} \omega : A}\ \mathsf{Run} \qquad \frac{u :: A \in \Delta}{\Delta;\Gamma \vdash u \mathbin{@} \omega : A}\ \mathsf{GHyp}$$

$$\frac{\Delta;\Gamma \vdash M \mathbin{@} \omega : \bigcirc A \quad \Delta;\Gamma, x : A \vdash E \mathbin{@} \omega \doteqdot C}{\Delta;\Gamma \vdash \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E \mathbin{@} \omega \doteqdot C}\ \bigcirc\mathsf{E}_{\doteqdot}$$

The rule GTerm corresponds to the the first clause in the characterization of $A\ \mathit{gcomp}$; its premise proves that M is a globally valid term of type A. The rule $\odot\mathsf{I}$ is the introduction rule for $\odot$; the rules $\odot\mathsf{E}$ and $\odot\mathsf{E}_{\div}$ are the elimination rules for $\odot$. The rule Run corresponds to the third clause in the characterization of $A\ \mathit{gcomp}$; it states that $\mathsf{run}\ \langle E\rangle$ contains an expression E that is computable at every world and also computes to a globally valid value of type A. The rule $\bigcirc\mathsf{E}_{\doteqdot}$ is another elimination rule for the modality $\bigcirc$. The remaining typing rules are obtained from those for the judgments $\Gamma \vdash M \mathbin{@} \omega : A$ and $\Gamma \vdash E \mathbin{@} \omega \div A$ in a straightforward way.

The rules for the world-free typing judgments are derived from their counterparts for the judgments with worlds by erasing all occurrences of $@\,\omega$. For instance, the rule Run derives the following rule:

$$\frac{\Delta;\cdot \vdash E \doteqdot A}{\Delta;\Gamma \vdash \mathsf{run}\ \langle E\rangle : A}\ \mathsf{Run}^{*}$$

For those world terms whose type depends on the world where they reside (*e.g.*, references), we do not have world-free typing rules.

The rule $\bigcirc\mathsf{E}_{\doteqdot}$ shows that we can derive a global expression typing judgment from $\mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E$ even though it creates a value that is not necessary globally valid. This makes sense because we can freely use x in E as long as the final computation in E returns a globally valid value.

The following proposition confirms the meaning of the world-free typing judgments:

Proposition 5.1. *For any world* ω*:*
If $\Delta;\Gamma \vdash M : A$*, then* $\Delta;\Gamma \vdash M \mathbin{@} \omega : A$.
If $\Delta;\Gamma \vdash E \div A$*, then* $\Delta;\Gamma \vdash E \mathbin{@} \omega \div A$.
If $\Delta;\Gamma \vdash E \doteqdot A$*, then* $\Delta;\Gamma \vdash E \mathbin{@} \omega \doteqdot A$.

5.1.3 Reduction and operational semantics

We define capture-avoiding *global term substitutions* $[\![M/u]\!]N$ and $[\![M/u]\!]E$ in a standard way. We also define a *global expression substitution* $\langle\!\langle E/u \rangle\!\rangle F$ in an analogous way to expression substitutions, and extend the expression substitution for $\mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E$:

$$\begin{aligned}
\langle\!\langle M/u \rangle\!\rangle F &= [\![M/u]\!]F \\
\langle\!\langle \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ E/u \rangle\!\rangle F &= \mathsf{letcmp}\ x \triangleleft M\ \mathsf{in}\ \langle\!\langle E/u \rangle\!\rangle F \\
\langle\!\langle \mathsf{letgcmp}\ u' \blacktriangleleft M\ \mathsf{in}\ E/u \rangle\!\rangle F &= \mathsf{letgcmp}\ u' \blacktriangleleft M\ \mathsf{in}\ \langle\!\langle E/u \rangle\!\rangle F \\
\langle \mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E/x \rangle F &= \mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ \langle E/x \rangle F
\end{aligned}$$

The substitution principle for expressions derived from *A gcomp* is stated as follows:

If $\Delta; \Gamma \vdash E \doteqdot A$ *and* $\Delta, u :: A; \Gamma \vdash F \div C$, *then* $\Delta; \Gamma \vdash \langle\!\langle E/u \rangle\!\rangle F \div C$.

If $\Delta; \Gamma \vdash E \mathbin{@} \omega \doteqdot A$ *and* $\Delta, u :: A; \Gamma \vdash F \mathbin{@} \omega \doteqdot C$, *then* $\Delta; \Gamma \vdash \langle\!\langle E/u \rangle\!\rangle F \mathbin{@} \omega \doteqdot C$.

The computation rule for $\mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E$ is based upon a new reduction rule:

$$\mathsf{letgcmp}\ u \blacktriangleleft \mathsf{gcmp}\ E\ \mathsf{in}\ F \Rightarrow_{\mathsf{exp}} \langle\!\langle E/u \rangle\!\rangle F$$

$$\frac{M \hookrightarrow \mathsf{gcmp}\ F \quad \langle\!\langle F/u \rangle\!\rangle E \mathbin{@} \omega \multimap V \mathbin{@} \omega'}{\mathsf{letgcmp}\ u \blacktriangleleft M\ \mathsf{in}\ E \mathbin{@} \omega \multimap V \mathbin{@} \omega'}\ \mathit{Letgcmp}$$

The type preservation property uses the new typing judgments with worlds. So does the monotonicity of the accessibility relation $\leq$ (in Definition 2.1).

Theorem 5.2 (Type preservation).
If $M \hookrightarrow V$ *and* $\cdot; \cdot \vdash M \mathbin{@} \omega : A$, *then* $\cdot; \cdot \vdash V \mathbin{@} \omega : A$.
If $E \mathbin{@} \omega \multimap V \mathbin{@} \omega'$ *and* $\cdot; \cdot \vdash E \mathbin{@} \omega \div A$, *then* $\cdot; \cdot \vdash V \mathbin{@} \omega' \div A$ *and* $\omega \leq^* \omega'$.
If $E \mathbin{@} \omega \multimap V \mathbin{@} \omega'$ *and* $\cdot; \cdot \vdash E \mathbin{@} \omega \doteqdot A$, *then* $\cdot; \cdot \vdash V \mathbin{@} \omega' \doteqdot A$ *and* $\omega \leq^* \omega'$.

Now we can show that an expression proving *A gcomp* indeed computes to a globally valid value of type *A*. Suppose $\cdot; \cdot \vdash E \mathbin{@} \omega \doteqdot A$ and $E \mathbin{@} \omega \multimap V \mathbin{@} \omega'$. We have $\cdot; \cdot \vdash V \mathbin{@} \omega' \doteqdot A$ by the above theorem and $\cdot; \cdot \vdash V : A$ by the typing rule GTerm. Then V is globally valid by Proposition 5.1. In conjunction with the typing rule Run, this implies that the evaluation rule *Run* safely converts expressions into terms.

5.1.4 Current world as an initial world

The above type system guarantees the safety of the run construct regardless of the choice for an initial world. Here we consider a special case where the current world is used as an initial world. This is useful, for instance, when we want to build a hierarchy of stores where a child store can access all its ancestor stores.

Now a term evaluation may require the current world, and we use an evaluation judgment $M \mathbin{@} \omega \hookrightarrow V$ where ω denotes the current world. The revised typing and evaluation rules for the run construct are:

$$\frac{\Delta; \Gamma \vdash E \mathbin{@} \omega \doteqdot A}{\Delta; \Gamma \vdash \mathsf{run}\ \langle E \rangle \mathbin{@} \omega : A}\ \mathsf{Run}' \qquad \frac{E \mathbin{@} \omega \multimap V \mathbin{@} \omega_{\mathit{final}}}{\mathsf{run}\ \langle E \rangle \mathbin{@} \omega \hookrightarrow V}\ \mathit{Run}'$$

The rule Run′ says that we can use those variables in Γ when computing E but not in the result of the computation. It is, however, semantically safe to use them in the result of the computation as well, since the result is transferred back to the current world where they all typecheck. For instance, $\mathsf{run}\ \langle x \rangle$ does not typecheck even when x typechecks, but it is semantically equivalent to x. A quick fix is to incorporate the typing context Γ into the global typing context (using every ordinary variable in it as a global variable):

$$\frac{\Delta, \Gamma; \cdot \vdash E \mathbin{@} \omega \doteqdot A}{\Delta; \Gamma \vdash \mathsf{run}\ \langle E \rangle \mathbin{@} \omega : A}\ \mathsf{Run}''$$

Then we can use those variables in Γ not only for the computation of E but also in its result.

In practice, the rule *Run*′ is not easy to implement unless we use in computing E a copy of the current world. The reason is that the runtime system must recover the original world ω from the final world ω_{final} by canceling all world effects produced during the computation of E. In the case of mutable references, we can achieve this by refining the type system so that the computation of E does not update ω. Then we can recover ω from ω_{final} by discarding $\omega_{\mathit{final}} - \omega$. The Kripke-style natural deduction system in [7] (which maintains a stack of typing contexts) in conjunction with a separation between read and write effects seems to be a good basis for such a refined type system.

5.2 Constructs for the global truth judgment

Although the run construct is now safe, there are still two problems. First the rule Run typechecks an expression E under an empty typing context, and as parameters to the computation of E, we can pass only global variables produced by the letgcmp construct *during computations*. Second the run construct takes an expression instead of a (global) computation term. That is, it takes not an encapsulation of a computation but the computation itself. This limits its utility because unlike Haskell's runST construct, we can never initiate a computation encapsulated in a (global) computation term that is not bound to a global variable.

We can resolve both problems by internalizing the global truth judgment *A global* with a modality $\boxdot$. The constructs for *A global* closely resemble those for the validity judgment *A valid* in [35].

5.2.1 Logic for global truth

We characterize A *global* with the new form of hypothetical judgment, where J can be any categorical judgment:

1. $\Delta, A\ \mathit{global}; \Gamma \vdash A\ \mathit{true}$.
2. If $\Delta; \cdot \vdash A\ \mathit{true}$ and $\Delta, A\ \mathit{global}; \Gamma \vdash J$, then $\Delta; \Gamma \vdash J$.

The first clause expresses that A *global* implies A *true*. The second clause can be thought of as the substitution principle for A *global* because $\Delta; \cdot \vdash A\ \mathit{true}$ implies $\Delta; \Gamma \vdash A\ \mathit{global}$.

5.2.2 Syntax and type system

The syntax for A *global* is as follows:

$$\begin{array}{lllll} \text{type} & A & ::= & \cdots \mid \boxdot A \\ \text{term} & M & ::= & \cdots \mid \mathsf{glo}\ M \mid \mathsf{letglo}\ u = M\ \mathsf{in}\ M \\ \text{expression} & E & ::= & \cdots \mid \mathsf{letglo}\ u = M\ \mathsf{in}\ E \\ \text{value} & V & ::= & \cdots \mid \mathsf{glo}\ M \end{array}$$

$\mathsf{glo}\ M$ is called a *global term*, and contains a globally valid term M. $\mathsf{letglo}\ u = M\ \mathsf{in}\ N$ and $\mathsf{letglo}\ u = M\ \mathsf{in}\ E$ expose to N and E a globally valid term obtained by evaluating M.

The new typing rules for the judgments with worlds are as follows:

$$\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \mathsf{glo}\ M \mathbin{@} \omega : \boxdot A}\ \boxdot\mathsf{I}$$

$$\frac{\Delta; \Gamma \vdash M \mathbin{@} \omega : \boxdot A \quad \Delta, u :: A; \Gamma \vdash N \mathbin{@} \omega : C}{\Delta; \Gamma \vdash \mathsf{letglo}\ u = M\ \mathsf{in}\ N \mathbin{@} \omega : C}\ \boxdot\mathsf{E}$$

$$\frac{\Delta; \Gamma \vdash M \mathbin{@} \omega : \boxdot A \quad \Delta, u :: A; \Gamma \vdash E \mathbin{@} \omega \div C}{\Delta; \Gamma \vdash \mathsf{letglo}\ u = M\ \mathsf{in}\ E \mathbin{@} \omega \div C}\ \boxdot\mathsf{E}_{\div}$$

$$\frac{\Delta; \Gamma \vdash M \mathbin{@} \omega : \boxdot A \quad \Delta, u :: A; \Gamma \vdash E \mathbin{@} \omega \doteqdot C}{\Delta; \Gamma \vdash \mathsf{letglo}\ u = M\ \mathsf{in}\ E \mathbin{@} \omega \doteqdot C}\ \boxdot\mathsf{E}_{\doteqdot}$$

The rule $\boxdot\mathsf{I}$ is the introduction rule for $\boxdot$; the rules $\boxdot\mathsf{E}$, $\boxdot\mathsf{E}_{\div}$, and $\boxdot\mathsf{E}_{\doteqdot}$ are the elimination rules for $\boxdot$. Note that in the rule $\boxdot\mathsf{I}$, the premise uses a world-free term typing judgment with an empty typing context because we are proving A *global*. New rules for the world-free typing judgments are derived by erasing all occurrences of $@\ \omega$ as before.

It is easy to show that Proposition 5.1 continues to hold. Then we can show that a term contained in a global term is indeed globally valid. Consider a global term $\mathsf{glo}\ M$ such that $\cdot; \cdot \vdash \mathsf{glo}\ M \mathbin{@} \omega : A$. By the rule $\boxdot\mathsf{I}$, we have $\cdot; \cdot \vdash M : A$. By Proposition 5.1, the term M typechecks at every world and is thus globally valid.

5.2.3 Reduction and operational semantics

The substitution principle for terms derived from A *global* is stated as follows (global term substitutions are extended in a standard way):

If $\Delta; \cdot \vdash M : A$ *and* $\Delta, u :: A; \Gamma \vdash N \mathbin{@} \omega : C$, *then* $\Delta; \Gamma \vdash [M/u]N \mathbin{@} \omega : C$.

If $\Delta; \cdot \vdash M : A$ *and* $\Delta, u :: A; \Gamma \vdash E \mathbin{@} \omega \div C$, *then* $\Delta; \Gamma \vdash [M/u]E \mathbin{@} \omega \div C$.

If $\Delta; \cdot \vdash M : A$ *and* $\Delta, u :: A; \Gamma \vdash E \mathbin{@} \omega \doteqdot C$, *then* $\Delta; \Gamma \vdash [M/u]E \mathbin{@} \omega \doteqdot C$.

We also extend expression substitutions for $\mathsf{letglo}\ u = M\ \mathsf{in}\ E$:

$$\langle \mathsf{letglo}\ u = M\ \mathsf{in}\ E/x \rangle F = \mathsf{letglo}\ u = M\ \mathsf{in}\ \langle E/x \rangle F$$
$$\langle\!\langle \mathsf{letglo}\ u' = M\ \mathsf{in}\ E/u \rangle\!\rangle F = \mathsf{letglo}\ u' = M\ \mathsf{in}\ \langle\!\langle E/u \rangle\!\rangle F$$

New evaluation and computation rules are based upon two new reduction rules:

$$\mathsf{letglo}\ u = \mathsf{glo}\ M\ \mathsf{in}\ N \Rightarrow_{\mathsf{term}} [M/u]N$$
$$\mathsf{letglo}\ u = \mathsf{glo}\ M\ \mathsf{in}\ E \Rightarrow_{\mathsf{exp}} [M/u]E$$

$$\frac{M \hookrightarrow \mathsf{glo}\ M' \quad [M'/u]N \hookrightarrow V}{\mathsf{letglo}\ u = M\ \mathsf{in}\ N \hookrightarrow V}\ \mathit{Letglo}_{\mathit{term}}$$

$$\frac{M \hookrightarrow \mathsf{glo}\ M' \quad [M'/u]E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'}{\mathsf{letglo}\ u = M\ \mathsf{in}\ E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'}\ \mathit{Letglo}_{\mathit{exp}}$$

The type preservation property in Theorem 5.2 continues to hold with these rules.

With the constructs for A *global*, we can pass as parameters to the run construct globally valid terms produced *during evaluations*. Note that despite the availability of $\mathsf{run}\ \langle E \rangle$, we cannot implement a term of type $\odot A \supset A$ because $\odot A$ does not prove that an expression is computable at every world. We can, however, implement a term $\mathit{run}_{\boxdot\odot}$ of type $\boxdot\odot A \supset \boxdot A$ and a term $\mathit{run}_{\boxdot\bigcirc\boxdot}$ of type $\boxdot\bigcirc\boxdot A \supset \boxdot A$, both of which effectively take an encapsulation of a computation to be initiated by the run construct:

$$\begin{array}{l} \mathsf{let}\ \mathit{run}_{\boxdot\odot} = \lambda x{:}\boxdot\odot A. \\ \quad \mathsf{letglo}\ u = x\ \mathsf{in}\ \mathsf{run}\ \langle \mathsf{letgcmp}\ u' \blacktriangleleft u\ \mathsf{in}\ \mathsf{glo}\ u' \rangle \\ \mathsf{let}\ \mathit{run}_{\boxdot\bigcirc\boxdot} = \lambda x{:}\boxdot\bigcirc\boxdot A. \\ \quad \mathsf{letglo}\ u = x\ \mathsf{in} \\ \quad \mathsf{run}\ \langle \mathsf{letcmp}\ y \triangleleft u\ \mathsf{in}\ \mathsf{letglo}\ u' = y\ \mathsf{in}\ \mathsf{glo}\ u' \rangle \end{array}$$

6 Related Work

Monadic languages. Moggi [23, 24] proposes monads as a tool for modeling various notions of computation in a uniform manner. Wadler [41, 42] popularizes the idea of using monads in structuring programs and incorporating effects into purely functional languages. The idea has been adopted in the design of Haskell.

From our perspective, Haskell is a monadic language because of its built-in IO monad [34] rather than its type class Monad. The IO monad forms a monadic sublanguage distinct from the functional sublanguage. Peyton Jones [33] clarifies the distinction between the two with a semantics for Haskell. Like the operational semantics of $\lambda^{\rightarrow}_{\bigcirc}$, it is stratified into two levels: an *inner denotational semantics* for the functional sublanguage and an *outer transition semantics* for the monadic sublanguage.

The linguistic framework of Haskell is Moggi's monadic metalanguage λ_{ml} [23, 24], which has served as the *de facto* standard for monadic languages [17, 18, 3, 37, 25, 26, 43]. From a type-theoretic perspective, λ_{ml} is connected to lax logic [8] via the propositions-as-types interpretation, as shown by Benton, Biermann, and de Paiva [5]. Pfenning and Davies [35] reformulate λ_{ml} by applying Martin-Löf's methodology of distinguishing judgments from propositions [22] to lax logic. They also show that lax logic is contained in modal logic in that the modality $\bigcirc$ of lax logic (which is the same modality as in $\lambda^{\rightarrow}_{\bigcirc}$) can be encoded as a composition $\Diamond\Box$ of the possibility modality $\Diamond$ and the necessity modality $\Box$ of modal logic. The new formulation of λ_{ml} draws a syntactic distinction between values and computations, and uses the modality $\bigcirc$ for computations. It is used in the design of a security-typed monadic language [6]; its underlying modal type theory inspires type systems in [1, 2] and effect systems in [27, 28].

The idea of the syntactic distinction but without an explicit modality for computations is used by Petersen *et al.* [29]. The same idea is also used by Mandelbaum, Walker, and Harper [21]. Their language is similar to $\lambda^{\rightarrow}_{\bigcirc}$ in that the operational semantics (but not the type system) uses an accessibility relation between worlds, and in that two kinds of functions are provided: "pure" functions of type $A \rightarrow C$ and "impure" functions of type $A \rightharpoonup C$. The meaning of a world is, however, slightly different: a world is a collection of facts on what serves as a world in $\lambda^{\rightarrow}_{\bigcirc}$.

Control effects. A typical monadic language draws no distinction between control effects and world effects and confines all kinds of effects to its monadic sublanguage [26]. Haskell also follows the same principle [31, 33], but the utility of control effects for its functional sublanguage has also been recognized. Peyton Jones *et al.* [32] propose an extension to Haskell with exceptions for its functional sublanguage. Because of Haskell's unconstrained order of evaluation, they are led to interpret exceptions as values and to exploit the IO monad to catch exceptions. From our perspective, this is not a complete implementation of exceptions for the functional sublanguage; rather it is an extension of exceptions for the monadic sublanguage.

run **construct**. From an operational point of view, we can think of Haskell's runST construct [17, 18, 19] as corresponding to our run construct for mutable references that uses an empty store as an initial world (*i.e.*, $\omega_{initial} = \{\}$ in the rule *Run*). Its safety is guaranteed by an augmented type system that indexes every state transformer with a type variable and lets runST accept a state transformer only if its index type variable is universally quantifiable. The idea of indexing state transformers is used in subsequent monadic languages [3, 37, 9]; it also inspires the higher-order type of a similar construct run in a monadic language of Moggi and Sabry [25].

Although it is designed to prevent interferences between state transformers, runST permits dangling references to be exported from one state transformer to another as long as they are never dereferenced. In contrast, run completely forbids dangling references because they are not globally valid. runST also assumes a lazy store in which instructions are executed on demand, whereas run assumes a strict store in which all instructions are executed sequentially.

Haskell's unsafePerformIO [34, 33] construct is similar to runST except that it is used for the IO monad and assumes a strict store. As its name suggests, however, it is unsafe. It even allows us to write a function of type A -> C [16]. Hence its safety must be verified by programmers. Our run construct does not replace unsafePerformIO which can be applied to external world effects as well internal world effects.

Effect systems and monadic languages. Wadler and Thiemann [43] show the connection between effect systems and monadic languages. They present a translation from the effect system of Talpin and Jouvelot [38] into a monadic language extended with mutable references and prove that the translation preserves types and semantics. A similar technique has been applied to translating Standard ML into monadic intermediate languages [40, 4], developing a monadic type system for a language where effects are lexically scoped [13], and translating an ML-like language with a construct for effect masking into a monadic language with a runST construct [37].

Wadler and Thiemann also point out that Haskell's runST construct is similar to the letregion construct of the region calculus of Tofte and Talpin [39], in which a region variable plays the role of an index type variable. The fundamental difference is that runST prohibits a state transformer from accessing more than one store whereas letregion allows an expression to access a stack of regions [19]. Therefore there is no direct correspondence between runST and letregion. For this reason, Fluet and Morrisett [9] use a construct newRGNVar for creating new regions (in addition to another construct

runRGN similar to runST) in their translation of a variant of the region calculus into an extension of System F with monadic types.

In $\lambda^{\rightarrow}_{\bigcirc}$, the fact that run is a term construct implies that we cannot use it to implement letregion. The rule Run', however, enables us to implement a limited form of letregion if the type system guarantees that the computation of E does not update ω.

7 Conclusion and Future Work

Based upon the view that monads are an abstraction more general than effects and not identified with effects, we have proposed a logical analysis of effects. Our analysis leads to a language $\lambda^{\rightarrow}_{\bigcirc}$ which uses the judgmental formulation and the possible worlds interpretation of modal logic. $\lambda^{\rightarrow}_{\bigcirc}$ draws a distinction between control effects and world effects, which makes it easy to combine effects at the language design level. It also gives programmers the freedom to choose either Haskell-style programming or ML-style programming (or both if necessary), and has a logically motivated run construct for invoking computations during evaluations. As such, $\lambda^{\rightarrow}_{\bigcirc}$ provides a linguistic framework for programming languages with effects, as an alternative to one based upon monads.

In applying the propositions-as-types interpretation to the computability judgment, we have shown that lax logic is the logic underlying not only monads *but also call-by-value languages*. This departs from the traditional view of call-by-value languages as corresponding to propositional logic under a particular reduction strategy. This is, however, not surprising, since the semantics of lax logic provides what is required of call-by-value languages: sequentialization of computations. Thus lax logic is an alternative to propositional logic as a logical basis for call-by-value languages.

Our work was inspired by the reformulation of λ_{ml} by Pfenning and Davies [35]. A characteristic feature of their reformulation is a syntactic distinction between terms and expressions, where terms denote values and expressions denote computations (as in $\lambda^{\rightarrow}_{\bigcirc}$). The syntactic distinction is not merely a cosmetic change in the syntax of monadic languages. It has led to the interpretation of terms and expressions as complete languages of their own, which in turn has led to the idea of separating control effects and world effects and the idea of viewing lax logic as a basis for call-by-value languages. Ultimately we believe that the idea of the syntactic distinction conveys a design principle that is not found in other monadic languages.

We are investigating how to refine the type system with indices so that we can decide not only the type of a given computation but also the kinds of world effects it may produce. For instance, we can use an indexed typing judgment $\Gamma \vdash E \mathbin{@} \omega \div_{\mathit{effect}} A$ and indexed types $\bigcirc_{\mathit{effect}} A$ and $A \rightarrow_{\mathit{effect}} C$, where *effect* shows the kinds of world effects the computation may produce (*e.g.*, $\mathit{effect} = \{\mathsf{file\_input}, \mathsf{store\_write}\}$). Such a refined type system would allow us to combine world effects *at the program level* rather than at the language design level. It would also enable us to achieve a safe definition of the run construct when we incorporate external world effects into the operational semantics (*e.g.*, the run construct rejects a computation if it produces external world effects.) We are also investigating how to exploit modal type theory to model control effects (*e.g.*, with comonads as in [28]). Thus our goal is to extend $\lambda^{\rightarrow}_{\bigcirc}$ to account for all kinds of effects, maintaining its overall safety.

Acknowledgment. We are grateful to Frank Pfenning and Jonathan Moody for their helpful discussion on this paper. The second author has been supported by the National Science Foundation under grant number 0121633: "ITR/SY+SI: Language Technology for Trustless Software Dissemination".

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL '02*, pages 247–259. ACM Press, 2002.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *POPL '03*, pages 14–25. ACM Press, 2003.

[3] Z. M. Ariola and A. Sabry. Correctness of monadic state: an imperative call-by-need calculus. In *POPL '98*, pages 62–73, New York, NY, 1998.

[4] N. Benton, A. Kennedy, and G. Russell. Compiling standard ml to java bytecodes. In *ICFP '98*, pages 129–140. ACM Press, 1998.

[5] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, Mar. 1998.

[6] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. Technical Report CMU-CS-03-164, School of Computer Science, Carnegie Mellon University, 2003.

[7] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[8] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, Aug. 1997.

[9] M. Fluet and G. Morrisett. Monadic regions. In *ICFP '04*, 2004. To appear.

[10] T. G. Griffin. A formulae-as-type notion of control. In *POPL '90*, pages 47–58. ACM Press, 1990.

[11] R. Harper, B. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[12] M. P. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[13] R. B. Kieburtz. Taming effects with monadic typing. In *ICFP '98*, pages 51–62. ACM Press, 1998.

[14] D. J. King and P. Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Glasgow functional programming workshop*, Glasgow, 1992. Springer Verlag.

[15] S. A. Kripke. Semantic analysis of modal logic. I: Normal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[16] J. Launchbury, J. R. Lewis, and B. Cook. On embedding a microarchitectural design language within haskell. In *ICFP '99*, pages 60–69. ACM Press, 1999.

[17] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *PLDI '94*, pages 24–35. ACM Press, 1994.

[18] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[19] J. Launchbury and A. Sabry. Monadic state: axiomatization and type safety. In *ICFP '97*, pages 227–238. ACM Press, 1997.

[20] C. Lüth and N. Ghani. Composing monads using coproducts. In *ICFP '02*, pages 133–144. ACM Press, 2002.

[21] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP '03*, pages 213–225. ACM Press, 2003.

[22] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[23] E. Moggi. Computational lambda-calculus and monads. In *LICS '89*, pages 14–23. IEEE Computer Society Press, June 1989.

[24] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[25] E. Moggi and A. Sabry. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming*, 11(6):591–627, Nov. 2001.

[26] E. Moggi and A. Sabry. An abstract monadic semantics for value recursion. In *Proceedings of the 2003 Workshop on Fixed Points in Computer Science*, Apr. 2003.

[27] A. Nanevski. From dynamic binding to state via modal possibility. In *PPDP '03*, pages 207–218. ACM Press, 2003.

[28] A. Nanevski. A modal calculus for effect handling. Technical Report CMU-CS-03-149, School of Computer Science, Carnegie Mellon University, 2003.

[29] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *POPL '03*, pages 172–184. ACM Press, 2003.

[30] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[31] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL '96*, pages 295–308. ACM Press, 1996.

[32] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *PLDI '99*, pages 25–36. ACM Press, 1999.

[33] S. L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In C. A. R. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*. IOS Press, Amsterdam, 2001.

[34] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL '93*, pages 71–84. ACM Press, 1993.

[35] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[36] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proceedings of the 5th*

International Conference on Foundations of Software Science and Computation Structures, pages 342–356. Springer-Verlag, 2002.

[37] M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *ICFP '99*, pages 8–17. ACM Press, 1999.

[38] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[39] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL '94*, pages 188–201. ACM Press, 1994.

[40] A. Tolmach. Optimizing ML using a hierarchy of monadic types. *Lecture Notes in Computer Science*, 1473:97–115, 1998.

[41] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[42] P. Wadler. The essence of functional programming. In *POPL '92*, pages 1–14. ACM Press, 1992.

[43] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4, 2003.

Appendix - Definition of $\lambda^{\rightarrow}_{\bigcirc}$

Abstract syntax

$$
\begin{array}{lll}
\text{type} & A & ::= \quad A \supset A \mid \bigcirc A \mid A \rightarrow A \mid \mathcal{W} \\
\text{world term type} & \mathcal{W} & \\
\text{term} & M & ::= \quad x \mid \lambda x{:}A.\, M \mid M\ M \mid \mathsf{cmp}\ E \mid \hat{\lambda} x{:}A.\, E \mid W \\
\text{world term} & W & \\
\text{expression} & E & ::= \quad M \mid \mathsf{letcmp}\ x \lhd M\ \mathsf{in}\ E \mid E \hat{}\, E \mid I \mid \{I/x\}E \mid \{M \hat{}\, E/x\}F \\
\text{instruction} & I & \\
\text{value} & V & ::= \quad \lambda x{:}A.\, M \mid \mathsf{cmp}\ E \mid \hat{\lambda} x{:}A.\, E \mid W
\end{array}
$$

Expression substitution

$$
\begin{array}{rcl}
\langle M/x\rangle F & = & [M/x]F \\
\langle \mathsf{letcmp}\ y \lhd M\ \mathsf{in}\ E/x\rangle F & = & \mathsf{letcmp}\ y \lhd M\ \mathsf{in}\ \langle E/x\rangle F \\
\langle \{E_1/y\}E_2/x\rangle F & = & \{E_1/y\}\langle E_2/x\rangle F \\
\langle I/x\rangle E & = & \{I/x\}E \\
\langle F_1 \hat{}\, F_2/x\rangle E & = & \langle F_1/f\rangle\{f \hat{}\, F_2/x\}E \quad \textit{fresh variable } f
\end{array}
$$

Type system

$$\frac{}{\Gamma, x : A \vdash x \mathbin{@} \omega : A}\ \mathsf{Hyp} \qquad \frac{\Gamma \vdash M \mathbin{@} \omega : A}{\Gamma \vdash M \mathbin{@} \omega \div A}\ \mathsf{Term}$$

$$\frac{\Gamma, x : A \vdash M \mathbin{@} \omega : C}{\Gamma \vdash \lambda x{:}A.\, M \mathbin{@} \omega : A \supset C}\ \supset\mathsf{I} \qquad \frac{\Gamma \vdash M_1 \mathbin{@} \omega : A \supset C \quad \Gamma \vdash M_2 \mathbin{@} \omega : A}{\Gamma \vdash M_1\ M_2 \mathbin{@} \omega : C}\ \supset\mathsf{E}$$

$$\frac{\Gamma \vdash E \mathbin{@} \omega \div A}{\Gamma \vdash \mathsf{cmp}\ E \mathbin{@} \omega : \bigcirc A}\ \bigcirc\mathsf{I} \qquad \frac{\Gamma \vdash M \mathbin{@} \omega : \bigcirc A \quad \Gamma, x : A \vdash E \mathbin{@} \omega \div C}{\Gamma \vdash \mathsf{letcmp}\ x \lhd M\ \mathsf{in}\ E \mathbin{@} \omega \div C}\ \bigcirc\mathsf{E}$$

$$\frac{\Gamma, x : A \vdash E \mathbin{@} \omega \div C}{\Gamma \vdash \hat{\lambda} x{:}A.\, E \mathbin{@} \omega : A \rightarrow C}\ \rightarrow\mathsf{I} \qquad \frac{\Gamma \vdash E_1 \mathbin{@} \omega \div A \rightarrow C \quad \Gamma \vdash E_2 \mathbin{@} \omega \div A}{\Gamma \vdash E_1 \hat{}\, E_2 \mathbin{@} \omega \div C}\ \rightarrow\mathsf{E}$$

$$\frac{\Gamma \vdash E \mathbin{@} \omega \div A \quad \Gamma, x : A \vdash F \mathbin{@} \omega \div C}{\Gamma \vdash \{E/x\}F \mathbin{@} \omega \div C}\ \mathsf{Sus}$$

Operational semantics

$$\frac{V ::= \lambda x{:}A.\, M \mid \mathsf{cmp}\ E \mid \hat{\lambda} x{:}A.\, E \mid W}{V \hookrightarrow V}\ \mathit{Val}$$

$$\frac{M_1 \hookrightarrow \lambda x{:}A.\, M \quad [M_2/x]M \hookrightarrow V}{M_1\ M_2 \hookrightarrow V}\ \mathit{LApp}$$

$$\frac{M \hookrightarrow V}{M \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega}\ \mathit{Term}$$

$$\frac{M \hookrightarrow \mathsf{cmp}\ F \quad \langle F/x\rangle E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'}{\mathsf{letcmp}\ x \lhd M\ \mathsf{in}\ E \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega'}\ \mathit{Letcmp}$$

$$\frac{E_1 \mathbin{@} \omega \rightharpoonup \hat{\lambda} x{:}A.\, F \mathbin{@} \omega' \quad \langle E_2/x\rangle F \mathbin{@} \omega' \rightharpoonup V \mathbin{@} \omega''}{E_1 \hat{}\, E_2 \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega''}\ \mathit{EApp}$$

$$\frac{I \mathbin{@} \omega \rightharpoonup V \mathbin{@} \omega' \quad [V/x]E \mathbin{@} \omega' \rightharpoonup V' \mathbin{@} \omega''}{\{I/x\}E \mathbin{@} \omega \rightharpoonup V' \mathbin{@} \omega''}\ \mathit{Sus}^I$$

$$\frac{M \hookrightarrow \hat{\lambda y}{:}A.\, E' \quad \langle\langle E/y\rangle E'/x\rangle F \mathbin{@} \omega \rightharpoonup V' \mathbin{@} \omega'}{\{M \hat{}\, E/x\}F \mathbin{@} \omega \rightharpoonup V' \mathbin{@} \omega'}\ \mathit{Sus}^{\hat{}}$$