Witnessing Purity, Constancy and Mutability

Ben Lippmeier

School of Computer Science
Australian National University
Ben.Lippmeier@anu.edu.au

Abstract. Restricting destructive update to values of a distinguished reference type prevents functions from being polymorphic in the mutability of their arguments. This restriction makes it easier to reason about program behaviour during transformation, but the lack of polymorphism reduces the expressiveness of the language. We present a System-F style core language that uses dependently kinded proof witnesses to encode information about the mutability of values and the purity of computations. We support mixed strict and lazy evaluation, and use our type system to ensure that only computations without visible side effects are suspended.

1 Introduction

Suppose we are writing a library that provides a useful data structure such as linked lists. A Haskell-style definition for the list type would be:

$$\textbf{data}\ \mathit{List}\ \ a = \mathit{Nil}\ \mid\ \mathit{Cons}\ \ a\ (\mathit{List}\ \ a)$$

The core language of compilers such as GHC is based around System-F [15]. Here is the translation of the standard *map* function to this representation, complete with type abstractions and applications:

$$\begin{array}{l}
\mathit{map} :: \forall a\ b.\ (a \to b) \to \mathit{List}\ \ a \to \mathit{List}\ \ b \\
\mathit{map} = \Lambda a.\ \Lambda b.\ \lambda(f : a \to b).\ \lambda(\mathit{list} : \mathit{List}\ \ a). \\
\quad \textbf{case}\ \mathit{list}\ \textbf{of} \\
\quad\quad \mathit{Nil} \quad\quad\quad\ \to \mathit{Nil}\ \ b \\
\quad\quad \mathit{Cons}\ \ x\ \ xs \to \mathit{Cons}\ \ b\ (f\ x)\ (\mathit{map}\ \ a\ b\ f\ \ xs)
\end{array}$$

Say we went on to define some other useful list functions, and then decided that we need one to destructively insert a new element into the middle of a list. In Haskell, side effects are carefully controlled and we would need to introduce a monad such as ST or IO [8] to encapsulate the effects due to the update. Destructive update is also limited to distinguished types such as *STRef* and *IORef*. We cannot use our previous list type, so will instead change it to use an *IORef*.

$$\textbf{data}\ \mathit{List}\ \ a = \mathit{Nil}\ \mid\ \mathit{Cons}\ \ a\ (\mathit{IORef}\ \ (\mathit{List}\ \ a))$$

Unfortunately, as we have changed the structure of our original data type, we can no longer use the previous definition of *map*, or any other functions we defined earlier. We must go back and refactor each of these function definitions to use the new type. We must insert calls to *readIORef* and use monadic sequencing combinators instead of vanilla let and where-expressions. However, doing so introduces explicit data dependencies into the core program. This in turn reduces the compiler's ability to perform optimisations such as deforestation and the full laziness transform [6], which require functions to be written in the "pure", non-monadic style. It appears that we need *two* versions of our list structure and its associated functions, an immutable version that can be optimised, and a mutable one that can be updated.

Variations of this problem are also present in ML and O'Caml. In ML, mutability is restricted to *ref* and *array* types [11]. In O'Caml, record types can have mutable fields, but variant types cannot [9]. Similarly to Haskell, in these languages we are forced to insert explicit reference types into the definitions of mutable data structures, which makes them incompatible with the standard immutable ones. This paper shows how to avoid this problem:

- We present a System-F style core language that uses region and effect typing to guide program optimisation. Optimisations that depend on purity can be performed on the the pure fragments of the program.
- We use region variables and dependently kinded witnesses to encode mutability polymorphism. This allows arbitrary data structures to be mutable without changing the structure of their value types.
- We use call-by-value evaluation as default, but support lazy evaluation via a primitive *suspend* operator. We use witnesses of purity to ensure that only pure function applications can be suspended.

Our goals are similar to those of Benton and Kennedy [3], but as in [15] we use a System-F based core language instead of a monadic one. Type inference and translation from source to core is discussed in [10].

2 Regions, Effects and Mutability Constraints

In Haskell and ML, references and arrays are distinguished values, and are the only ones capable of being destructively updated. This means that the structure of mutable data is necessarily different from the structure of constant data, which makes it difficult to write polymorphic functions that act on both. For example, if we use *IORef Int* as the type of a mutable integer and *Int* as the type of a constant integer, then we would need *readIORef* to access the first, but not the second. On the other hand, if we were to treat all data as mutable, then every function would exhibit a side effect. This would prevent us from using code-motion style optimisations that depend on purity.

Instead, we give integers the type $Int\ r$, where r is a region variable, and constrain r to be mutable or constant as needed. Our use of region variables is

similar to that by Talpin and Jouvelot [16], where the variable r is a name for a set of locations in the store where a run-time object may lie. We do not use regions for controlling allocation as per [17], due to the difficulty of statically determining when objects referenced by suspended computations can be safely deallocated. We define region variables to have kind %, and use this symbol because pictorially it is two circles separated by a line, a mnemonic for "this, or that". The kind of value types is *, so the *Int* type constructor has kind $Int :: \% \rightarrow *$. The type of a literal integer such as '5' is:

$$5 \;::\; \forall(r : \%).\, Int\ \ r$$

In our System-F style language, type application corresponds to instantiation, and '5' is the name of a function that allocates a new integer object into a given region. Note that unlike [16] we do not use allocation effects. This prevents us from optimising away some forms of duplicated computation, such as described in §7 of [4], but also simplifies our type system. For the rest of this paper we will elide explicit kind annotations on binders when they are clear from context.

2.1 Updating Integers

To update an integer we use the *updateInt* function which has type:

$$updateInt \;::\; \forall r_1\ r_2.\, Mutable\ \ r_1 \Rightarrow Int\ \ r_1 \rightarrow Int\ \ r_2\ \xrightarrow{Read\, r_2\ \vee\ Write\, r_1}\ ()$$

This function reads the value of its second integer argument, and uses this to overwrite the value of the first. As in [16] we annotate function types with their latent effects. We organise effects as a lattice and collect atomic effects with the $\vee$ operator. We use $\bot$ as the effect of a pure function, and unannotated function arrows are taken to have this effect. We also use a set-like subtraction operator where the effect $\sigma \setminus \sigma'$ contains the atomic effects that appear in σ but not σ'. We use ! as the kind of effects, so *Read* has kind $Read :: \% \rightarrow !$. The symbol ! is a mnemonic for "something's happening!".

Returning to the type of *updateInt*, $Mutable\ \ r_1$ is a *region constraint* that ensures that only mutable integers may be updated. When we call this function we must pass a *witness* to the fact that this constraint is satisfied, a point we will discuss further in §3.

When the number of atomic effects becomes large, using the above syntax for effects becomes cumbersome. Due to this we sometimes write effect terms after the body of the type instead:

$$\begin{aligned} updateInt \;::\; & \forall r_1\ r_2.\, Mutable\ \ r_1 \Rightarrow Int\ \ r_1 \rightarrow Int\ \ r_2\ \xrightarrow{e_1}\ () \\ & \rhd\ e_1 = Read\ \ r_2 \vee Write\ \ r_1 \end{aligned}$$

The symbol $\rhd$ is pronounced "with". Note that the effect variable e_1 is not quantified. It has been introduced for convenience only and is not a parameter of the type.

2.2 Updating Algebraic Data

Along with primitive types such as *Int*, the definition of an algebraic data type can also contain region variables. For example, we define our lists as follows:

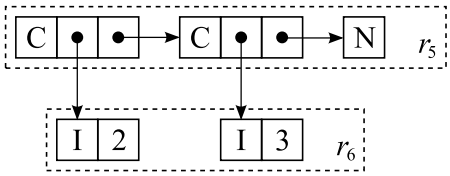
$$\textbf{data}\ \mathit{List}\ r\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ r\ a)$$

This definition is similar to the one from §1 except that we have also applied the *List* constructor to a region variable. This variable identifies the region that contains the list cells, and can be constrained to be constant or mutable as needed. The definition also introduces data constructors that have the following types:

$$\begin{aligned}\mathit{Nil} &:: \forall r\ a.\ \mathit{List}\ r\ a\\ \mathit{Cons} &:: \forall r\ a.\ a \to \mathit{List}\ r\ a \to \mathit{List}\ r\ a\end{aligned}$$

In the type of *Nil*, the fact that r is quantified indicates that this constructor allocates a new *Nil* object. Freshly allocated objects do not alias with existing objects, so they can be taken to be in any region. On the other hand, in the type of *Cons*, the type of the second argument and return value share the same region variable r, which means the new cons-cell is allocated into the same region as the existing cells. For example, evaluation of the following expression produces the store objects shown below.

$$\begin{aligned}&\mathit{list} :: \mathit{List}\ r_5\ (\mathit{Int}\ r_6)\\ &\mathit{list} = \mathit{Cons}\ r_5\ (\mathit{Int}\ r_6)\ (2\ r_6)\ (\mathit{Cons}\ r_5\ (\mathit{Int}\ r_6)\ (3\ r_6)\ (\mathit{Nil}\ r_5\ (\mathit{Int}\ r_6)))\end{aligned}$$



As the list cells and integer elements are in different regions, we can give them differing mutabilities. If the type of *list* was constrained as follows, then we would be free to update the integer elements, but not the spine.

$$\mathit{list} :: \mathit{Const}\ r_5 \Rightarrow \mathit{Mutable}\ r_6 \Rightarrow \mathit{List}\ r_5\ (\mathit{Int}\ r_6)$$

The definition of an algebraic type also introduces a set of update operators, one for each updatable component of the corresponding value. For our list type, as we could usefully update the head and tail pointers in a cons-cell, we get the following operators:

$$\begin{aligned}\mathit{update}_{\mathit{Cons},0} &:: \forall r\ a.\ \mathit{Mutable}\ r \Rightarrow \mathit{List}\ r\ a \to a \xrightarrow{\mathit{Write}\ r} ()\\ \mathit{update}_{\mathit{Cons},1} &:: \forall r\ a.\ \mathit{Mutable}\ r \Rightarrow \mathit{List}\ r\ a \to \mathit{List}\ r\ a \xrightarrow{\mathit{Write}\ r} ()\end{aligned}$$

These operators both take a list and a new value. If the list contains an outer cons-cell, then the appropriate pointer in that cell is updated to point to the new value. If the list is not a cons, then a run-time error is raised.

3 Witnesses and Witness Construction

The novel aspect of our core language is that it uses dependently kinded witnesses to manage information about purity, constancy and mutability. A witness is a special type that can occur in the term being evaluated, and its occurrence guarantees a particular property of the program. The System-Fc [15] language uses a similar mechanism to manage information about non-syntactic type equality. Dependent kinds were introduced by the Edinburgh Logical Framework (LF) [1] which uses them to encode logical rules.

Note that although our formal operational semantics manipulates witnesses during reduction, in practice they are only used to reason about the program during compilation, and are not needed at runtime. Our compiler erases witnesses before code generation, along with all other type information.

3.1 Region Handles

We write witnesses with an underline, and the first we discuss are the region allocation witnesses $\underline{\rho_n}$. These are also called *region handles* and are introduced into the program with the **letregion** r **in** t expression. Reduction of this expression allocates a fresh handle $\underline{\rho}$ and substitutes it for all occurrences of the variable r in t. To avoid problems with variable capture we require all bound variables r in the initial program to be distinct. Although region handles are not needed at runtime, we can imagine them to be operational descriptions of physical regions of the store, perhaps incorporating a base address and a range. For example, the following program adds two to its argument, while storing an intermediate value in a region named r_3.

$$
\begin{array}{lcl}
\mathit{addTwo} & :: & \forall r_1\ r_2.\ \mathit{Int}\ r_1 \xrightarrow{\mathit{Read}\, r_1} \mathit{Int}\ r_2 \\
\mathit{addTwo} & = & \Lambda r_1\ r_2.\ \lambda(x : \mathit{Int}\ r_1). \\
 & & \textbf{letregion}\ r_3\ \textbf{in}\ \mathit{succ}\ r_3\ r_2\ (\mathit{succ}\ r_1\ r_3\ x)
\end{array}
$$

This program makes use of the primitive *succ* function that reads its integer argument and produces a new value into a given region:

$$
\mathit{succ} :: \forall r_1\ r_2.\ \mathit{Int}\ r_1 \xrightarrow{\mathit{Read}\, r_1} \mathit{Int}\ r_2
$$

Note the phase distinction between region variables r_n and region handles $\underline{\rho_n}$. Region handles are bound by region variables. As no regions exist in the store before execution, region handles may not occur in the initial program. Also, although the outer call to *succ* reads a value in r_3, this effect is not observable by calling functions, so is masked and not included in the type signature of *addTwo*. This is similar to the system of [17].

3.2 Witnesses of Constancy and Mutability

The constancy or mutability of values in a particular region is represented by the witnesses $\underline{\text{const } \rho}$ and $\underline{\text{mutable } \rho}$. Once again, these witnesses may not occur in

the initial program. Instead, they are created with the *MkConst* and *MkMutable* witness type constructors which have the following kinds:

$$\begin{array}{ll} \mathit{MkConst} & :: \Pi(r : \%).\ \mathit{Const}\ r \\ \mathit{MkMutable} & :: \Pi(r : \%).\ \mathit{Mutable}\ r \end{array}$$

Both constructors take a region handle and produce the appropriate witness. To ensure that both const ρ_n and mutable ρ_n for the same ρ_n cannot be created by a given program, we require the mutability of a region to be set at the point it is introduced. We introduce new regions with **letregion**, so extend this construct with an optional witness binding that specifies the desired mutability. If a function accesses values in a given region, and does not possess either a witness of constancy or mutability for that region, then it cannot assume either. For example, the following function computes the length of a list by destructively incrementing a local accumulator, then copying out the final value.

$$\begin{array}{l} \mathit{length} :: \forall a\ r_1\ r_2.\ \mathit{List}\ r_1\ a \xrightarrow{\mathit{Read}\ r_1} \mathit{Int}\ r_2 \\ \mathit{length} = \Lambda a\ r_1\ r_2.\ \lambda(\mathit{list} : \mathit{List}\ r_1\ a). \\ \quad \textbf{letregion}\ r_3\ \textbf{with}\ \{w = \mathit{MkMutable}\ r_3\}\ \textbf{in} \\ \quad \textbf{let}\ \ (\mathit{acc} : \mathit{Int}\ r_3) = 0\ r_3 \\ \qquad (\mathit{length}' : ...) \\ \qquad\quad = \lambda(\mathit{xx} : \mathit{List}\ r_1\ a). \\ \qquad\qquad \textbf{case}\ \mathit{xx}\ \textbf{of} \\ \qquad\qquad\quad \mathit{Nil} \rightarrow \mathit{copyInt}\ r_3\ r_2\ \mathit{acc} \\ \qquad\qquad\quad \mathit{Cons}\ \_\ \mathit{xs} \rightarrow \textbf{let}\ (\_ : ()) = \mathit{incInt}\ r_3\ w\ \mathit{acc} \\ \qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{length}'\ \mathit{xs} \\ \quad \textbf{in}\ \ \mathit{length}'\ \mathit{list} \end{array}$$

where

$$\begin{array}{ll} \mathit{copyInt} & :: \forall r_1\ r_2.\ \mathit{Int}\ r_1 \xrightarrow{\mathit{Read}\ r_1} \mathit{Int}\ r_2 \\ \mathit{incInt} & :: \forall r_1.\ \mathit{Mutable}\ r_1 \Rightarrow \mathit{Int}\ r_1 \xrightarrow{\mathit{Read}\ r_1 \vee \mathit{Write}\ r_1} () \end{array}$$

The set after the **with** keyword binds an optional witness type variable. If the region variable bound by the **letregion** is r, then the right of the witness binding must be either *MkConst* r or *MkMutable* r. The type constructors *MkConst* and *MkMutable* may not occur elsewhere in the program. The *length* function above makes use of *incInt* which requires its integer argument to be in a mutable region, and we satisfy this constraint by passing it our witness to the fact.

3.3 Laziness and Witnesses of Purity

Although we use call-by-value evaluation as the default, we can suspend the evaluation of an arbitrary function application with the *suspend* operator:

$$\mathit{suspend} :: \forall a\ b\ e.\ \mathit{Pure}\ e \Rightarrow (a \xrightarrow{e} b) \rightarrow a \rightarrow b$$

suspend takes a parameter function of type $a \xrightarrow{e} b$, its argument of type a, and defers the application by building a thunk at runtime. When the value of the thunk is demanded, the contained function will be applied to its argument, yielding a result of type b. As per [7], values are demanded when they are used as the function in an application, or are inspected by a case-expression or primitive operator such as *update*. The constraint $Pure\ e$ indicates that we must also provide a witness that the application to be suspended is observably pure. Witnesses of purity are written $\underline{\text{pure}\ \sigma}$ where σ is some effect. They can be created with the *MkPurify* witness type constructor. For example, the following function computes the successor of its argument, but only when the result is demanded:

$$
\begin{array}{l}
succL\ ::\ \forall r_1\ r_2.\ Const\ r_1 \Rightarrow Int\ r_1 \xrightarrow{Read\ r_1} Int\ r_2 \\
succL\ =\ \Lambda r_1\ r_2\ (w : Const\ r_1).\ \lambda(x : Int\ r_1). \\
\qquad suspend\ (Int\ r_1)\ (Int\ r_2)\ (Read\ r_1)\ (MkPurify\ r_1\ w) \\
\qquad\qquad (succ\ r_1\ r_2)\ x
\end{array}
$$

MkPurify takes a witness that a particular region is constant, and produces a witness proving that a read from that region is pure. It has the following kind:

$$MkPurify\ ::\Pi(r : \%).\ Const\ r \rightarrow\ Pure\ (Read\ r)$$

Reads of constant regions are pure because it does not matter when the read takes place, the same value will be returned each time. Note that in our system there are several ways of writing the effect of a pure function. As mentioned in §2.1 the effect term $\bot$ is manifestly pure. However, we can also treat any other effect as pure if we can produce a witness of the appropriate kind. For example, $Read\ r_5$ is pure if we can produce a witness of kind $Pure\ (Read\ r_5)$.

3.4 Witness Joining and Explicit Effect Masking

Purity constraints extend naturally to higher order functions. Here is the definition of a lazy map function, *mapL*, which constructs the first list element when called, but only constructs subsequent elements when they are demanded:

$$
\begin{array}{l}
mapL\ ::\ \forall a\ b\ r_1\ r_2\ e. \\
\qquad Const\ r_1 \Rightarrow Pure\ e \Rightarrow (a \xrightarrow{e} b) \rightarrow List\ r_1\ a \rightarrow List\ r_2\ b \\
\\
mapL \\
=\ \Lambda a\ b\ r_1\ r_2\ e\ (w_1 : Const\ r_1)\ (w_2 : Pure\ e). \\
\quad \lambda(f : a \xrightarrow{e} b)\ (list : List\ r_1\ a). \\
\quad \textbf{mask}\ (MkPureJoin\ (Read\ r_1)\ e\ (MkPurify\ r_1\ w_1)\ w_2)\ \textbf{in} \\
\quad \textbf{case}\ list\ \textbf{of} \\
\qquad Nil \rightarrow Nil\ r_2\ b \\
\qquad Cons\ x\ xs \rightarrow Cons\ r_2\ b\ (f\ x) \\
\qquad\qquad (suspend\ (List\ r_1\ a)\ (List\ r_2\ b)\ \bot\ MkPure \\
\qquad\qquad\qquad (mapL\ a\ b\ r_1\ r_2\ e\ w_1\ w_2\ f)\ xs)
\end{array}
$$

The inner case-expression in this function has the effect $\mathit{Read}\ r_1 \vee e$. The first part is due to inspecting the list constructors, and the second is due to the application of the argument function f to the list element x. However, as the recursive call to *mapL* is suspended, *mapL* itself must be pure. One way to satisfy this constraint would be to pass a witness showing that $\mathit{Read}\ r_1 \vee e$ is pure directly to *suspend*. This works, but leaves *mapL* with a type that contains this (provably pure) effect term. Instead, we have chosen to explicitly mask this effect in the body of *mapL*. This gives *mapL* a manifestly pure type, and allows us to pass a trivial witness to *suspend* to show that the recursive call is pure.

The masking is achieved with the **mask** δ **in** t expression, which contains a witness of purity δ and a body t. The type and value of this expression is the same as for t, but its effect is the effect of t minus the terms which δ proves are pure. In our *mapL* example we prove that $\mathit{Read}\ r_1 \vee e$ is pure by combining two other witnesses, w_1 which proves that the list cells are in a constant region, and w_2 which proves that the argument function itself is pure. They are combined with the *MkPureJoin* witness type constructor which has the following kind:

$$\mathit{MkPureJoin} :: \Pi(e_1 : !).\ \Pi(e_2 : !).\ \mathit{Pure}\ e_1 \to \mathit{Pure}\ e_2 \to\ \mathit{Pure}\,(e_1 \vee e_2)$$

Our *mapL* example also uses *MkPure*, which introduces a witness that the effect $\bot$ is pure. Note that our type for *mapL* now contains exactly the constraints that are *implicit* in a lazy language such as Haskell. In Haskell, all algebraic data is constant, and all functions are pure. In our language, we can suspend function applications as desired, but doing so requires the functions and data involved to satisfy the usual constraints of lazy evaluation.

4 Language

We are now in a position to formally define our core language and its typing rules. The structure of the language is given in Fig. 1. Most has been described previously, so we only discuss the aspects not covered so far. Firstly, we use $\Diamond$ as the result kind of witness kind constructors, so a constructor such as *Mutable* has kind $\mathit{Mutable} :: \% \to \Diamond$. This says that a witness of kind $\mathit{Mutable}\ r$ guarantees a property of a region, where $\Diamond$ refers to the guarantee.

We use use τ_i as binders for value types, σ_i as binders for effect types, and δ_i as binders for type expressions that construct witness types. Δ_i refers to constructed witnesses of the form $\underline{\rho}$, $\underline{\mathrm{const}\ \rho}$, $\underline{\mathrm{mutable}\ \rho}$ or $\underline{\mathrm{pure}\ \sigma}$. φ_i can refer to any type expression.

The values in our term language are identified with v. Weak values, v°, consist of the values as well as suspended function applications $\mathit{suspend}\ \overline{\varphi}\ v_1^\circ\ v_2^\circ$. A suspension is only forced when its (strong) value is demanded by using it as the function in an application, the discriminant of a case expression, or as an argument to a primitive operator such as *update*. Store locations l_i are discussed in §4.2. The other aspects of our term language are standard. Recursion can be introduced via **fix** in the usual way, but we omit it to save space. To simplify the presentation we require the alternatives in a case-expression to be exhaustive.

Symbol Classes

a, r, e, w	$\rightarrow$ (type variable)	T	$\rightarrow$ (type constructor)
x	$\rightarrow$ (value variable)	K	$\rightarrow$ (data constructor)

Kinds

$$
\begin{array}{lll}
\kappa ::= \kappa\ \varphi \mid \Pi(a : \kappa_1).\ \kappa_2 & & \text{(kinds)} \\
\quad \mid\ * \mid \% \mid\ ! \mid \Diamond & & \text{(base kinds)} \\
\quad \mid\ \mathit{Const} \mid \mathit{Mutable} \mid \mathit{Pure} & & \text{(kind constrs)}
\end{array}
$$

Types

$$
\begin{array}{lll}
\varphi, \tau, \sigma, \delta, \Delta & & \\
\quad ::= a \mid \forall(a : \kappa).\ \tau \mid \varphi_1\ \varphi_2 \mid (\rightarrow) \mid () \mid T & & \text{(types)} \\
\quad \mid\ \sigma_1 \vee \sigma_2 \mid \bot \mid \mathit{Read} \mid \mathit{Write} & & \text{(effects)} \\
\quad \mid\ \mathit{MkConst} \mid \mathit{MkMutable} \mid \mathit{MkPure} \mid \mathit{MkPurify} \mid \mathit{MkPureJoin} & & \text{(witness constrs)} \\
\quad \mid\ \underline{\rho} \mid \underline{\text{const}\ \rho} \mid \underline{\text{mutable}\ \rho} \mid \underline{\text{pure}\ \sigma} & & \text{(witnesses)}
\end{array}
$$

Terms

$$
\begin{array}{lll}
t & ::= & v \mid t\ \varphi \mid t_1\ t_2 \mid \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w = \delta}\}\ \textbf{in}\ t \mid K\ \overline{\varphi}\ \overline{t} \\
& \mid & \textbf{case}\ t\ \textbf{of}\ \overline{K\ \overline{x : \tau} \rightarrow t'} \mid \mathit{update}_{K,i}\ \overline{\varphi}\ t_1\ t_2 \mid \mathit{suspend}\ \overline{\varphi}\ t_1\ t_2 \\
& \mid & \textbf{mask}\ \delta\ \textbf{in}\ t \\
v^\circ, u^\circ & ::= & v \mid \mathit{suspend}\ \overline{\varphi}\ v_1^\circ\ v_2^\circ \qquad \text{(weak values)} \\
v, u & ::= & x \mid l \mid () \mid \Lambda(a : \kappa).\ t \mid \lambda(x : \tau).\ t \qquad \text{(values)}
\end{array}
$$

Derived Forms

$$
\begin{array}{llll}
\kappa_1 \rightarrow \kappa_2 & \stackrel{\text{def}}{=} & \Pi(\_ : \kappa_1).\ \kappa_2 & \textbf{let}\ (x : \tau) = t_1\ \textbf{in}\ t_2 \stackrel{\text{def}}{=} (\lambda(x : \tau).\ t_2)\ t_1 \\
\kappa\ \Rightarrow \tau & \stackrel{\text{def}}{=} & \forall(\_ : \kappa).\ \tau & \textbf{letregion}\ r\ \textbf{in}\ t \stackrel{\text{def}}{=} \textbf{letregion}\ r\ \textbf{with}\ \emptyset\ \textbf{in}\ t
\end{array}
$$

Store Typing

$\Sigma \quad ::= l : \tau \mid \underline{\rho} \mid \underline{\text{const}\ \rho} \mid \underline{\text{mutable}\ \rho}$

Type Environment

$\Gamma \quad ::= a : \kappa \mid x : \tau$

Fig. 1. Core Language

$$\boxed{\Gamma \vdash_{\mathrm{K}} \kappa :: \kappa'}$$

$$\frac{\kappa \in \{*,\ \%,\ !,\ \Diamond\}}{\Gamma \vdash_{\mathrm{K}} \kappa :: \kappa}\ (\mathrm{KsRefl}) \qquad \frac{\Gamma \vdash_{\mathrm{K}} \kappa_1 :: \kappa_{11} \rightarrow \kappa_{12} \quad \Gamma \vdash_{\mathrm{K}} \varphi :: \kappa_{11}}{\Gamma \vdash_{\mathrm{K}} \kappa_1\ \varphi :: \kappa_{12}}\ (\mathrm{KsApp})$$

$$\Gamma \vdash_{\mathrm{K}} \mathit{Const} :: \% \rightarrow \Diamond \qquad \Gamma \vdash_{\mathrm{K}} \mathit{Mutable} :: \% \rightarrow \Diamond \qquad \Gamma \vdash_{\mathrm{K}} \mathit{Pure} :: ! \rightarrow \Diamond$$

Fig. 2. Kinds of Kinds

$$\boxed{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \kappa}$$

$$\Gamma,\ a : \kappa \mid \Sigma \vdash_{\mathrm{T}} a :: \kappa \ \text{(KiVar)}$$

$$\Gamma \mid \Sigma,\ \underline{\rho} \vdash_{\mathrm{T}} \underline{\rho} :: \% \ \text{(KiHandle)} \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \bot :: ! \ \text{(KiBot)}$$

$$\frac{\Gamma \vdash_{\mathrm{K}} \kappa_1 :: \kappa_1' \quad \Gamma,\ a : \kappa_1 \mid \Sigma \vdash_{\mathrm{T}} \tau :: \kappa_2}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \forall (a : \kappa_1).\ \tau :: \kappa_2}\ \text{(KiAll)} \qquad \frac{a \notin fv(\Gamma) \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_1 :: ! \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_2 :: !}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \sigma_1 \vee \sigma_2 :: !}\ \text{(KiJoin)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_1 :: \Pi (a : \kappa_1).\ \kappa_2 \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_1}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_1\ \varphi_2 :: \kappa_2[\varphi_2/a]}\ \text{(KiApp)}$$

$$\Gamma \mid \Sigma,\ \underline{\mathrm{const}\ \rho} \vdash_{\mathrm{T}} \underline{\mathrm{const}\ \rho} :: \mathit{Const}\ \underline{\rho}\ \text{(KiConst)}$$

$$\Gamma \mid \Sigma,\ \underline{\mathrm{mutable}\ \rho} \vdash_{\mathrm{T}} \underline{\mathrm{mutable}\ \rho} :: \mathit{Mutable}\ \underline{\rho}\ \text{(KiMutable)}$$

$$\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\mathrm{pure}\ \bot} :: \mathit{Pure}\ \bot\ \text{(KiPure)}$$

$$\Gamma \mid \Sigma,\ \underline{\mathrm{const}\ \rho} \vdash_{\mathrm{T}} \underline{\mathrm{pure}\ (\mathit{Read}\ \rho)} :: \mathit{Pure}\ (\mathit{Read}\ \underline{\rho})\ \text{(KiPurify)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\mathrm{pure}\ \sigma_1} :: \mathit{Pure}\ \sigma_1 \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\mathrm{pure}\ \sigma_2} :: \mathit{Pure}\ \sigma_2}{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \underline{\mathrm{pure}\ (\sigma_1 \vee \sigma_2)} :: \mathit{Pure}\ (\sigma_1 \vee \sigma_2)}\ \text{(KiPureJoin)}$$

$$\begin{array}{llll}
\Gamma \mid \Sigma \vdash_{\mathrm{T}} (\rightarrow) & :: * \rightarrow * \rightarrow ! \rightarrow * & \Gamma \mid \Sigma \vdash_{\mathrm{T}} () & :: * \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{Bool} & :: \% \rightarrow * & \Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{Read} & :: \% \rightarrow ! \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{MkConst} & :: \Pi (r : \%).\ \mathit{Const}\ r & \Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{Write} & :: \% \rightarrow ! \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{MkMutable} & :: \Pi (r : \%).\ \mathit{Mutable}\ r & \Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{MkPure} & :: \mathit{Pure}\ \bot \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{MkPurify} & :: \Pi (r : \%).\ \mathit{Const}\ r \rightarrow \mathit{Pure}\ (\mathit{Read}\ r) & & \\
\Gamma \mid \Sigma \vdash_{\mathrm{T}} \mathit{MkPureJoin} & :: \Pi (e_1 : !).\ \Pi (e_2 : !).\ \mathit{Pure}\ e_1 \rightarrow \mathit{Pure}\ e_2 \rightarrow \mathit{Pure}\ (e_1 \vee e_2) & &
\end{array}$$

Fig. 3. Kinds of Types

4.1 Typing Rules

In Fig. 2 the judgement form $\Gamma \vdash_{\mathrm{K}} \kappa :: \kappa'$ reads: with type environment Γ, kind κ has kind κ'. We could have added a super-kind stratum containing $\Diamond$, but inspired by [12] we cap the hierarchy in this way to reduce the volume of typing rules.

In Fig. 3 the judgement form $\Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \kappa$ reads: with type environment Γ and store typing Σ, type φ has kind κ. We discuss store typings in §4.3.

In Fig. 4 the judgement form $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$ reads: with type environment Γ and store typing Σ, term t has type τ and effect σ. In TyLetRegion the premise "$\overline{\delta_i}$ well formed" refers to the requirement discussed in §3.2 that the witness introduced by a **letregion** must concern the bound variable r. In TyUpdate and TyAlt, the meta-function ctorTypes returns a set containing the types of the data constructors associated with type constructor T.

$$\boxed{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma}$$

$$\Gamma,\ x : \tau \mid \Sigma \vdash x :: \tau ; \bot \ \text{(TyVar)} \qquad \Gamma \mid \Sigma,\ l : \tau \vdash l :: \tau ; \bot \ \text{(TyLoc)}$$
$$\Gamma,\ K : \tau \mid \Sigma \vdash K :: \tau ; \bot \ \text{(TyCtor)} \qquad \Gamma \mid \Sigma \vdash () :: () ; \bot \ \text{(TyUnit)}$$

$$\frac{\Gamma,\ a : \kappa \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma_2}{\Gamma \mid \Sigma \vdash \Lambda(a : \kappa).\, t_2 :: \forall(a : \kappa).\, \tau_2 ; \sigma_2} \ \text{(TyAbsT)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 :: \forall(a : \kappa_{11}).\, \varphi_{12} ; \sigma_1 \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi_2 :: \kappa_{11}}{\Gamma \mid \Sigma \vdash t_1\ \varphi_2 :: \varphi_{12}[\varphi_2/a] ; \sigma_1[\varphi_2/a]} \ \text{(TyAppT)}$$

$$\frac{\Gamma,\ x : \tau_1 \mid \Sigma \vdash t :: \tau_2 ; \sigma}{\Gamma \mid \Sigma \vdash \lambda(x : \tau_1).\, t :: \tau_1 \xrightarrow{\sigma} \tau_2 ; \bot} \ \text{(TyAbs)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_2 :: \tau_{11} ; \sigma_2 \qquad \Gamma \mid \Sigma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma} \tau_{12} ; \sigma_1}{\Gamma \mid \Sigma \vdash t_1\ t_2 :: \tau_{12} ; \sigma_1 \vee \sigma_2 \vee \sigma} \ \text{(TyApp)}$$

$$\frac{\begin{array}{c}\overline{\delta_i} \text{ well formed} \quad r \notin fv(\tau) \qquad \Gamma \vdash_{\mathrm{K}} \kappa_i :: \Diamond \\ \Gamma,\ r : \%,\ \overline{w_i : \kappa_i} \mid \Sigma \vdash t :: \tau ; \sigma \qquad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta_i :: \kappa_i\end{array}}{\Gamma \mid \Sigma \vdash \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t :: \tau ; \sigma \setminus (\mathit{Read}\ r \vee \mathit{Write}\ r)} \ \text{(TyLetRegion)}$$

$$\frac{\Gamma \mid \Sigma \vdash t :: T\ \varphi\ \overline{\varphi'} ; \sigma \quad \overline{\Gamma \mid \Sigma \vdash p_i \to t_i :: T\ \varphi\ \overline{\varphi'} \to \tau ; \sigma_i'}^{\,n}}{\Gamma \mid \Sigma \vdash \textbf{case}\ t\ \textbf{of}\ \overline{p \to t} :: \tau ; \sigma \vee \mathit{Read}\ \varphi \vee \sigma_0' \vee \sigma_1' ... \vee \sigma_n'} \ \text{(TyCase)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Mutable}\ \varphi \qquad \Gamma \mid \Sigma \vdash t' :: \tau_i[\varphi/r][\overline{\varphi'/a}] ; \sigma' \\ \Gamma \mid \Sigma \vdash t :: T\ \varphi\ \overline{\varphi'} ; \sigma \qquad K :: \forall(r : \%)(\overline{a : \kappa}).\overline{\tau} \to T\ r\ \overline{a} \in \mathrm{ctorTypes}(T)\end{array}}{\Gamma \mid \Sigma \vdash \mathit{update}_{K,i}\ \varphi\ \overline{\varphi'}\ \delta\ t\ t' :: () ; \sigma \vee \sigma' \vee \mathit{Write}\ \varphi} \ \text{(TyUpdate)}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma \quad \Gamma \mid \Sigma \vdash t_1 :: \tau_{11} \xrightarrow{\sigma} \tau_{12} ; \sigma_1 \quad \Gamma \mid \Sigma \vdash t_2 :: \tau_{11} ; \sigma_2}{\Gamma \mid \Sigma \vdash \mathit{suspend}\ \tau_{11}\ \tau_{12}\ \sigma\ \delta\ t_1\ t_2 :: \tau_{12} ; \sigma_1 \vee \sigma_2} \ \text{(TySuspend)}$$

$$\frac{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma \quad \Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma'}{\Gamma \mid \Sigma \vdash \textbf{mask}\ \delta\ \textbf{in}\ t :: \tau ; \sigma \setminus \sigma'} \ \text{(TyMaskPure)}$$

$$\boxed{\Gamma \mid \Sigma \vdash p \to t :: \tau \to \tau' ; \sigma}$$

$$\frac{\begin{array}{c}\theta = [\varphi/r\ \overline{\varphi'/a}] \\ K :: \forall(r : \%)(\overline{a : \kappa}).\overline{\tau} \to T\ r\ \overline{a} \in \mathrm{ctorTypes}(T) \quad \Gamma,\ \overline{x : \theta(\tau)} \mid \Sigma \vdash t :: \tau' ; \sigma\end{array}}{\Gamma \mid \Sigma \vdash K\ \overline{x} \to t :: T\ \varphi\ \overline{\varphi'} \to \tau' ; \sigma} \ \text{(TyAlt)}$$

Fig. 4. Types of Terms

4.2 Dynamic Semantics

During evaluation, all updatable data is held in the store (also known as the heap), which is defined in Fig. 5. The store contains bindings that map abstract store locations to store objects. Each store object consists of a constructor tag C_K and a list of store values $\overline{\pi}$, where each value can be a location, unit value, abstraction or suspension. Each binding is annotated with a region handle ρ that specifies the region that the binding belongs to. Note that store *objects* can be usefully updated, but store *values* can not.

The store also contains *properties* that specify how bindings in the various regions may be used. The properties are ρ, (const ρ) and (mutable ρ). The last two indicate whether a binding in that region may be treated as constant, or updated. When used as a property, a region handle ρ indicates that the corresponding region has been created and is ready to have bindings allocated into it. Note that the region handles of store bindings and properties are not underlined because those occurrences are not used as types.

In Fig. 6 the judgement form $H; \delta \rightsquigarrow \delta'$ reads: with store H, witness δ produces witness δ'. Operationally, properties can be imagined as protection flags on regions of the store — much like the read, write and execute bits in a hardware page table. The witness constructors *MkConst* and *MkMutable* test for these properties, producing a type-level artefact showing that the property was set. If we try to evaluate either constructor when the desired property is *not* set, then the evaluation becomes stuck.

In Fig. 7 the judgement form $H; t \longrightarrow H'; t'$ reads: in heap H term t reduces to a new heap H' and term t'. In EvLetRegion the propOf meta-function maps a witness to its associated store property. Also, note that the premise of EvLetRegion is always true, and produces the required witnesses and properties from the given witness constructions $\overline{\delta_i}$.

l	$\rightarrow$	(store location)	
ρ	$\rightarrow$	(region handle)	
o	$::=$	$\rho \mid$ const $\rho \mid$ mutable ρ	(property)
π	$::=$	$l \mid () \mid \Lambda(a : \kappa).t \mid \lambda(x : \tau).t \mid$ *suspend* $\overline{\varphi}\ \pi\ \pi'$	(store value)
μ	$::=$	$C_K\ \overline{\pi}$	(store object)
H	:	$\{\ l \overset{\rho}{\mapsto} \mu\ \} + \{\ o\ \}$	(store)

Fig. 5. Stores and Store Objects

4.3 Soundness

In the typing rules we use a store typing Σ that models the state of the heap as the program evaluates. The store typing contains the type of each store location, along with witnesses to the current set of store properties. We say the store typing *models* the store, and write $\Sigma \models H$, when all members of the store typing correspond to members of the store. Conversely, we say the store is *well typed*, and write $\Sigma \vdash H$ when it contains all the bindings and properties predicted by the store typing. Both the store and store typing grow as the program evaluates, and neither bindings, properties or witnesses are removed once added.

$$\frac{H\ ;\ \delta \rightsquigarrow \delta'}{H\ ;\ E_w[\delta] \rightsquigarrow E_w[\delta']} \qquad \begin{array}{rcl} E_w & ::= & [\,] \mid \mathit{MkPurify}\ \rho\ E_w \\ & \mid & \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ E_w\ E_w \end{array}$$

$$H[\text{ const } \rho\ ]\ ;\ \mathit{MkConst}\ \underline{\rho} \rightsquigarrow \underline{\text{const } \rho} \qquad \text{(EwConst)}$$

$$H[\text{ mutable } \rho\ ]\ ;\ \mathit{MkMutable}\ \underline{\rho} \rightsquigarrow \underline{\text{mutable } \rho} \qquad \text{(EwMutable)}$$

$$H\ ;\ \mathit{MkPure} \rightsquigarrow \underline{\text{pure } \bot} \qquad \text{(EwPure)}$$

$$H\ ;\ \mathit{MkPurify}\ \underline{\rho}\ \underline{\text{const } \rho} \rightsquigarrow \underline{\text{pure } (\mathit{Read}\ \rho)} \qquad \text{(EwPurify)}$$

$$H\ ;\ \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \underline{\text{pure } \sigma_1}\ \underline{\text{pure } \sigma_2} \rightsquigarrow \underline{\text{pure } (\sigma_1 \vee \sigma_2)} \qquad \text{(EwPureJoin)}$$

Fig. 6. Witness Construction

$$\frac{e \longrightarrow e'}{E_v[e] \longrightarrow E_v[e']} \qquad \begin{array}{rcl} E_v & ::= & [\,] \mid E_v\ \varphi \mid E_v\ t_2 \mid v\ E_v \mid \mathbf{case}\ E_v\ \mathbf{of}\ \overline{alt} \\ & \mid & K\ \overline{\varphi}\ E_v\ t_1\ \ldots \quad \mid K\ \overline{\varphi}\ v_0\ E_v\ \ldots \mid \ldots \\ & \mid & \mathit{update}_{K,i}\ \overline{\varphi}\ E_v\ t_2 \mid \mathit{update}_{K,i}\ \overline{\varphi}\ l\ E_v \\ & \mid & \mathit{suspend}\ \overline{\varphi}\ E_v\ t_2 \quad \mid \mathit{suspend}\ \overline{\varphi}\ v\ E_v \end{array}$$

$$H\ ;\ (\Lambda(a :: \kappa).\ t)\ \varphi \longrightarrow H\ ;\ t[\varphi/a] \qquad \text{(EvTAppAbs)}$$

$$H\ ;\ (\lambda(x :: \tau).\ t)\ v^\circ \longrightarrow H\ ;\ t[v^\circ/x] \qquad \text{(EvAppAbs)}$$

$$\frac{H,\ \overline{\text{propOf}(\Delta_i)}\ ;\ \overline{\delta_i[\underline{\rho}/r] \rightsquigarrow \Delta_i} \qquad \rho \text{ fresh}}{H\ ;\ \mathbf{letregion}\ r\ \mathbf{with}\ \{\overline{w_i = \delta_i}\}\ \mathbf{in}\ t \longrightarrow H,\ \rho,\ \overline{\text{propOf}(\Delta_i)}\ ;\ t[\overline{\Delta_i/w_i}][\underline{\rho}/r]} \qquad \text{(EvLetRegion)}$$

$$H[\rho]\ ;\ K\ \underline{\rho}\ \overline{\varphi}\ \overline{v^\circ} \longrightarrow H,\ l \overset{\rho}{\mapsto} C_K\ \overline{v^\circ}\ ;\ l \qquad l \text{ fresh} \qquad \text{(EvAlloc)}$$

$$H[l \overset{\rho}{\mapsto} C_K\ \overline{v^\circ}]\ ;\ \mathbf{case}\ l\ \mathbf{of}\ \ldots K\ \overline{x} \to t \ldots \longrightarrow H\ ;\ t[\overline{v^\circ}/\overline{x}] \qquad \text{(EvCase)}$$

$$\frac{H\ ;\ \delta \rightsquigarrow \delta'}{H\ ;\ \mathit{update}\ \overline{\varphi}\ \delta\ t\ t' \longrightarrow H\ ;\ \mathit{update}\ \overline{\varphi}\ \delta'\ t\ t'} \qquad \text{(EvUpdateW)}$$

$$H[\text{ mutable } \rho\ ],\ l \overset{\rho}{\mapsto} C_K\ \overline{v^\circ}^n\ ;\ \mathit{update}_{K,i}\ \overline{\varphi}\ \underline{\text{mutable } \rho}\ l\ u^\circ \longrightarrow H,\ l \overset{\rho}{\mapsto} C_K\ v_0..u_i^\circ..v_n\ ;\ () \qquad \text{(EvUpdate)}$$

$$H[\text{ mutable } \rho\ ],\ l \overset{\rho}{\mapsto} C_K\ \overline{v^\circ}\ ;\ \mathit{update}_{K',i}\ \overline{\varphi}\ \underline{\text{mutable } \rho}\ l\ u^\circ \longrightarrow H\ ;\ \text{fail} \qquad K \neq K' \qquad \text{(EvFail)}$$

$$\frac{H\ ;\ \delta \rightsquigarrow \delta'}{H\ ;\ \mathit{suspend}\ \overline{\varphi}\ \delta\ t\ t' \longrightarrow H\ ;\ \mathit{suspend}\ \overline{\varphi}\ \delta'\ t\ t'} \qquad \text{(EvSuspendW)}$$

$$H\ ;\ \mathit{suspend}\ \tau\ \tau'\ \sigma\ \underline{\text{pure } \sigma}\ (\lambda(x : \tau).\ t)\ v^\circ \longrightarrow H\ ;\ t[v^\circ/x] \qquad \text{(EvSuspend)}$$

$$H\ ;\ \mathbf{mask}\ \delta\ \mathbf{in}\ t \longrightarrow H\ ;\ t \qquad \text{(EvMask)}$$

Fig. 7. Term Evaluation

Store bindings can be modified by the *update* operator, but the typing rules for update ensure that bindings retain the types predicted by the store typing. The rule TyLoc of Fig. 4 and KiHandle, KiConst, KiMutable and KiPurify of Fig. 3 ensure that if a location or witness occurs in the term, then it also occurs in the store typing. Provided the store typing models the store, this also means that the corresponding binding or property is present in the store. From the evaluation rules in Fig. 7, the only term that adds properties to the store is **letregion**, and when it does, it also introduces the corresponding witnesses into the expression. The well-formedness restriction on **letregion** guarantees that a witnesses of mutability and constancy for the same region cannot be created. This ensures that if we have, say, the witness $\underline{\text{const } \rho}$ in the term, then there is *not* a (mutable ρ) property in the store. This means that bindings in those regions can never be updated, and it is safe to suspend function applications that read them.

Our progress and preservation theorems are stated below. We do not prove these here, but [10] contains a proof for a similar system. The system in this paper supports full algebraic data types, whereas the one in [10] is limited to booleans. Also, here we include a $r \notin \mathit{fv}(\tau)$ premise in the TyLetRegion rule, which makes the presentation easier. See [10] for a discussion of this point.

Progress. If $\emptyset \mid \Sigma \vdash t :: \tau ; \sigma$ and $\Sigma \models H$ and $\Sigma \vdash H$ and nofab(t) then either $t \in$ Value or for some H', t' we have ($H; t \longrightarrow H'; t'$ and nofab(t') or $H; t \longrightarrow H'; \text{fail}$).

Preservation. If $\Gamma \mid \Sigma \vdash t :: \tau ; \sigma$ and $H; t \longrightarrow H'; t'$ then for some Σ', σ' we have $\Gamma \mid \Sigma' \vdash t' :: \tau ; \sigma'$ and $\Sigma' \supseteq \Sigma$ and $\Sigma' \models H'$ and $\Sigma' \vdash H'$ and $\Gamma \mid \Sigma \vdash \sigma' \sqsubseteq \sigma$.

In the Progress Theorem, "nofab" is short for "no fabricated region witnesses", and refers to the syntactic constraint that *MkConst* and *MkMutable* may only appear in the witness binding of a **letregion** and not elsewhere in the program. We could perhaps recast these two constructors as separate syntactic forms of **letregion**, and remove the need for nofab, but we have chosen not to do this because we prefer the simpler syntax.

In the Preservation Theorem, note that the latent effect of the term reduces as the program progresses. The $\sqsubseteq$ relationship on effects is defined in the obvious way, apart from the following extra rule:

$$\frac{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma}{\Gamma \mid \Sigma \vdash \sigma \sqsubseteq \bot} \quad \text{(SubPurify)}$$

This says that if we can construct a witness that a particular effect is pure, then we can treat it as such. This allows us to erase read effects on constant regions during the proof of Preservation. It is needed to show that forcing a suspension does not have a visible effect, and that we can disregard explicitly masked effect terms when entering into the body of a mask-expression.

5 Related Work

The inspiration for our work has been to build on the monadic intermediate languages of [18], [3] and [13]. Note that for our purposes, the difference between using effect and monadic typing is largely syntactic. We prefer effect typing because it mirrors our operational intuition more closely, but [19] gives a translation between the two. Our system extends the previous languages with region, effect and mutability polymorphism, which improves the scope of optimisations that can be performed. In [4] and [2], Benton *et al* present similar monadic languages that include region and effect polymorphism, but do not consider mutability polymorphism or lazy evaluation.

The Capability Calculus [5] provides region based memory management, whereby a capability is associated with each region, and an expression can only access a region when it holds its capability. When the region is deallocated, its associated capability is revoked, ensuring soundness. The capabilities of [5] have similarities to the witnesses of our system, but theirs are not reified in the term being evaluated, and we do not allow ours to be revoked.

The BitC [14] language permits any location, whether on the stack, heap or within data structures to be mutated. Its operational semantics includes an explicit stack as well as a heap, and function arguments are implicitly copied onto the stack during application. BitC includes mutability annotations, but does not use region or effect typing.

6 Conclusion and Future Work

We have presented a System-F style intermediate language that supports mutability polymorphism as well as lazy evaluation, and uses dependently kinded witnesses to track the purity of effects and the mutability and constancy of regions. One of the current limitations of our system is that the results of all case alternatives must have the same type. This prevents us from choosing between, say, a mutable and a constant integer. In future work we plan to provide a new region constraint *Blocked* that represents the fact that an object could be in either a mutable or constant region. We would permit such objects to be read, but not updated, and computations that read them could not be suspended. Doing so would likely require introducing a notion of subtyping into the system, so the types of all alternatives could be coerced to a single upper bound.

The system presented in this paper has been implemented in the prototype Disciplined Disciple Compiler (DDC) which can be obtained from the `haskell.org` website.

References

1. Arnon Avron, Furio Honsell, and Ian A. Mason. An overview of the Edinburgh Logical Framework. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 323–240. Springer-Verlag, 1989.
2. Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proc. of TLDI 2007*, pages 15–26. ACM, 2007.
3. Nick Benton and Andrew Kennedy. Monads, effects and transformations. In *Electronic Notes in Theoretical Computer Science*, pages 1–18. Elsevier, 1999.
4. Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *Proc. of PPDP*, pages 87–96. ACM, 2007.
5. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. of POPL*, pages 262–275. ACM, 1999.
6. André Luís de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.
7. John Launchbury. A natural semantics for lazy evaluation. In *Proc. of POPL*, pages 144–154. ACM, 1993.
8. John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proc. of PLDI*, pages 24–35. ACM, 1994.
9. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, release 3.11, documentation and user's manual. Technical report, INRIA, 2008.
10. Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD thesis, Australian National University, 2009. (submitted June 2009).
11. David B. MacQueen. Standard ML of New Jersey. In *Proc. of the Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag, 1991.
12. Simon Peyton Jones and E. Meijer. Henk: a typed intermediate language. In *Proc. of the Workshop on Types in Compilation*, 1997.
13. Simon Peyton Jones, Mark Shields, John Launchbury, and Andrew Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *Proc. of POPL*, pages 49–61. ACM, 1998.
14. Jonathan Shapiro, Swaroop Sridhar, and Scott Doerrie. BitC language specification. Technical report, The EROS Group and Johns Hopkins University, 2008.
15. Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System-F with type equality coercions. In *Proc. of TLDI*. ACM, 2007.
16. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. of Logic in Computer Science*, pages 162–173. IEEE, 1992.
17. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, January 2006.
18. Andrew Tolmach. Optimizing ML using a hierarchy of monadic types. In *Workshop on Types in Compilation*, pages 97–113. Springer Verlag, 1998.
19. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Computation and Logic*, 4(1):1–32, 2003.