Under consideration for publication in J. Functional Programming

Witnessing Mutability, Purity and Aliasing for Program Optimisation

BEN LIPPMEIER
School of Computer Science and Engineering
University of New South Wales, Australia ∗
(*e-mail:* benl@ouroborus.net)

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 and aliasing properties of data, 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:

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

The core language of compilers such as GHC is based around System-F (Sulzmann *et al.*, 2007). Here is the translation of the standard *map* function to this representation, complete with type abstractions and applications:

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

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* (Launchbury & Peyton Jones, 1994) to encapsulate the effects due to the update. Destructive update is also limited to distinguished types such as $STRef$ and $IORef$. We

∗ This article is an expansion of a paper presented at APLAS 2009 in Seoul.

cannot use our previous list type, so will instead change it to use an $IORef$.

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

Unfortunately, changing the structure of our original data type means that we can no longer use the previous definition of *map*, or any other list functions defined earlier. We must go back and refactor each of these 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 program. This in turn reduces the compiler's ability to perform optimisations such as deforestation and the full laziness transform (de Medeiros Santos, 1995), which require functions to be written in the "pure", non-monadic style. It appears that we need *two* versions of our list type 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 (MacQueen, 1991). In O'Caml, record types can have mutable fields, but variant types cannot (Leroy *et al.*, 2008). 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 variables and region class constraints to encode mutability polymorphism. This allows arbitrary data 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 purity constraints on effect variables to represent the fact that only pure function applications should be suspended.
- We use dependently kinded witnesses to encode information about mutability and purity in the core language, and show how they can be used to reason about the correctness of program optimisations in the presence of destructive update.
- As a natural extension to the above system, we discuss witnesses of no-aliasing (distinctness), and show how they can be used to improve the scope of the previous optimisations.

Our goals in program optimisation are similar to those of (Benton & Kennedy, 1999), but as in (Sulzmann *et al.*, 2007) we use a System-F based core language instead of a monadic one. Type inference and translation from the source to core language is discussed in (Lippmeier, 2010). Our work is embodied in the *Disciplined Disciple Compiler (DDC)*, with *Disciple* being the name of the language it compiles, and *Disciplined* invoking the Type and Effect Discipline of (Talpin & Jouvelot, 1992) which forms the core of our system. DDC can be obtained from `http://ddc.ouroborus.net`.

2 Regions, Effects and Purity

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 & Jouvelot, 1992), where the variable r is a name for a set of locations in the store where a run-time object may lie. However, we do not use regions for memory management as per (Tofte *et al.*, 2006), 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:

$$5 \quad :: \quad \forall(r : \%).\ Int\ r$$

In our System-F style language, instantiation corresponds to type (and region) application, so '5' can be seen as a function that allocates a new integer object into a given region. Note that unlike (Talpin & Jouvelot, 1992) we do not use allocation effects. This prevents us from optimising away some forms of duplicated computation, but also simplifies our type system. This is discussed further in §5. For the rest of this paper we will elide explicit kind annotations on binders when they are clear from context. Also, note that in this paper we are primarily discussing the core language of DDC. In most cases, the Disciple source programs do not need to include region or effect information, as it can be inferred (Lippmeier, 2010).

2.1 Updating Integers

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

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

This function reads the value of its second argument, and uses this to overwrite the first. As in (Talpin & Jouvelot, 1992) 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 $updateInt$, the term $Mutable\ r_1$ is a *region constraint* that ensures that only mutable integers can be updated. When we apply $updateInt$ we must pass a *witness* (proof term) that guarantees that this constraint is satisfied, a point we discuss further in §3.

2.2 Constancy and Constraint Exclusivity

Suppose that we wish to *prevent* a particular value from being updated. In this case we add a *Const* constraint to the corresponding region variable. For example, we can write:

$$pi \quad :: \quad Const\ r \Rightarrow Float\ r$$

In our system we intend for the *Const* and *Mutable* constraints to have the same meaning as the `const` and `mutable` type qualifiers of C (Foster *et al.*, 1999). However, note that Haskell-style type class constraints do not provide an equivalent exclusivity property. For example, suppose we also added a *Mutable* constraint to the above type:

$$pi \quad :: \quad Mutable\ r \Rightarrow Const\ r \Rightarrow Float\ r$$

Although this type looks suspect, it is perfectly valid. That is, valid but not useful. We can write a term that is assigned this type, but such a term cannot be used. To access the inner *Float* we would need to pass witnesses that guarantee both the mutability and constancy of *r*, and we ensure that both of these cannot exist in the same program. The mechanism we use to achieve this is discussed in §3. Note that exclusivity of constraints is limited to constraints on region variables. In the (full) Disciple language there is no way to prevent the programmer from defining a value type to be, say, an instance of both *Fractional* and *Integral*, as per Haskell.

2.3 Updating Algebraic Data

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

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

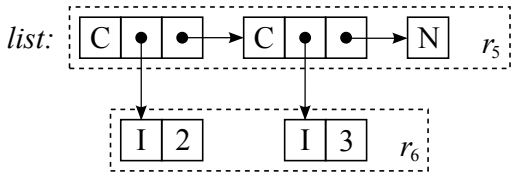
This definition is similar to the one from §1 except that we have also applied the *List* constructor to a region variable. This variable identifies the region that contains the list cells, and can be constrained to be constant or mutable as needed. The definition also introduces data constructors that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

$$\begin{array}{lll} Nil & :: & \forall r\ a.\ List\ r\ a \\ Cons & :: & \forall r\ a.\ a \rightarrow List\ r\ a\ \rightarrow List\ r\ a \end{array}$$

In the type of *Nil*, the fact that *r* is quantified indicates that this constructor allocates a new *Nil* object each time it is applied. Freshly allocated objects do not alias existing objects, so they can be taken to be in any region. On the other hand, in the type of *Cons*, the second argument and return type share the same region variable *r*, which means the new cons-cell is allocated into the same region as the existing cells.

For example, the evaluation of the following expression produces the store objects shown below.

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



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

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

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:

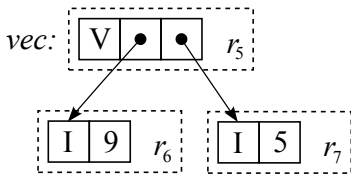
$$\begin{array}{lcl} update_{Cons,0} & :: & \forall r\ a.\ Mutable\ r \Rightarrow List\ r\ a \rightarrow a \xrightarrow{Write\ r} () \\ update_{Cons,1} & :: & \forall r\ a.\ Mutable\ r \Rightarrow List\ r\ a \rightarrow List\ r\ a \xrightarrow{Write\ r} () \end{array}$$

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.

2.4 Types with Several Region Variables

Algebraic data types can contain more than one region variable. For example, we could define a vector of two integers as follows:

data $Vec\ r_1\ r_2\ r_3 = V\ (Int\ r_2)\ (Int\ r_3)$
$vec = V\ r_5\ r_6\ r_7\ (9\ r_6)\ (5\ r_7)$



Note that the outer V constructor is in region r_5, while the two integer objects are in regions r_6 and r_7. Having three region variables in this type gives us three degrees of freedom for mutability. The first variable r_5 is called the *primary region variable* and corresponds to the outer constructor of the structure. Constraining this variable to be mutable allows us to update the pointers so they point to different objects. Constraining r_6 or r_7 to be mutable allows us to update the values in the integer objects directly.

Once again, note that all objects in the above structure are potentially mutable. Also, note that we do not need to modify the shape of their types, such as by inserting Ref constructors, to achieve this. We call this property *arbitrary destructive update*.

2.5 Laziness and Purity Constraints

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

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

The *suspend* operator 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 (Launchbury, 1993), 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 only function applications that do not exhibit visible side effects may be suspended.

As usual, once an application has been suspended, determining *when* it will finally be evaluated can be very difficult. Adding the purity constraint ensures that programs using laziness return predictable results, independent of when their thunks happen to be forced. This makes code analysis substantially easier, for both the programmer and compiler. Note that in the semantics of a purely functional lazy language such as Haskell, every function application is automatically suspended, and thus comes with its own implicit purity constraint. By making both suspension and purity *explicit*, we gain another degree of freedom, which allows us to support impure features such as arbitrary destructive update as well.

3 Witnesses and Witness Construction

We use dependently kinded proof witnesses to encode information about mutability and purity in the core language. A witness is a special type that can occur in the program being evaluated, and its occurrence guarantees a particular property of the program. The System-Fc (Sulzmann *et al.*, 2007) language uses a similar mechanism to encode information about non-syntactic type equality. Dependent kinds were introduced by the Edinburgh Logical Framework (LF) (Avron *et al.*, 1989) which uses them to encode logical rules.

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

The first witnesses we discuss are the region allocation witnesses $\underline{\rho}$. Syntactically, this expression consists of two parts. The plain ρ is a label (name) for a particular region in the store, and the underline turns the label into a *type*. Region allocation witnesses are also called *region handles* and they are introduced into the program with the **letregion** r **in** t expression. The reduction of this expression allocates a fresh label ρ and substitutes its handle $\underline{\rho}$ 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 unique. 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. As an example, the following function adds two to its argument, while storing an intermediate value in the region 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 function makes use of the primitive *succ* function that reads its integer argument and produces a new value into a given region:

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

The following reduction illustrates the operation of *addTwo*. We write machine states as $H;t$ where H is the store and t is the term being reduced. For now, we will say that the store contains bindings written $l \overset{\rho}{\mapsto} d$ where l is an abstract location, d is the data contained at that location, and ρ is the label of the region that binding belongs to. We will discuss the form of data more precisely in §4.2. The store also contains plain region labels ρ which indicate that the corresponding region has been allocated and is ready to have bindings added to it.

$$\begin{array}{lll} & \ldots & \\ \xrightarrow{*} & \rho_5 & ;\ \mathit{addTwo}\ \underline{\rho_5}\ \underline{\rho_5}\ (23\ \underline{\rho_5}) \\ \longrightarrow & \rho_5,\ l_1 \overset{\rho_5}{\mapsto} 23 & ;\ \mathit{addTwo}\ \underline{\rho_5}\ \underline{\rho_5}\ l_1 \\ \xrightarrow{*} & \rho_5,\ l_1 \overset{\rho_5}{\mapsto} 23 & ;\ \textbf{letregion}\ r_3\ \textbf{in}\ \mathit{succ}\ r_3\ \underline{\rho_5}\ (\mathit{succ}\ \underline{\rho_5}\ r_3\ l_1) \\ \longrightarrow & \rho_5,\ l_1 \overset{\rho_5}{\mapsto} 23,\ \rho_6 & ;\ \mathit{succ}\ \underline{\rho_6}\ \underline{\rho_5}\ (\mathit{succ}\ \underline{\rho_5}\ \underline{\rho_6}\ l_1) \\ \xrightarrow{*} & \rho_5,\ l_1 \overset{\rho_5}{\mapsto} 23,\ \rho_6,\ l_2 \overset{\rho_6}{\mapsto} 24 & ;\ \mathit{succ}\ \underline{\rho_6}\ \underline{\rho_5}\ l_2 \\ \longrightarrow & \ldots & \end{array}$$

At the beginning, region ρ_5 has already been allocated, and the reduction of $(23\ \underline{\rho_5})$ adds a new binding into the heap. The **letregion** expression allocates a fresh region label ρ_6 and its handle $\underline{\rho_6}$ is substituted for r_3. 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 a program starts, region handles cannot not occur in the initial program. Also, as **letregion** simultaneously allocates a region and introduces its handle, the occurrence of a handle in the program guarantees the fact that the corresponding region has been created and is ready to have bindings added into it. To say this another way: the region handle "witnesses" the fact that the corresponding region has been created.

Returning to the definition of *addTwo*, note that although the outer call to *succ* reads a value in r_3, this effect is not observable by calling functions. This means it can be masked and not included in the type signature. This similar to system of (Talpin & Jouvelot, 1992).

3.2 Witnesses of Constancy and Mutability

The constancy or mutability of values in a particular region is encoded by the witnesses $\underline{\text{const}\ \rho}$ and $\underline{\text{mutable}\ \rho}$. Once again, these are types that cannot occur in the initial program.

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

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

Both constructors take a region handle and produce the corresponding witness. We use the dependent binding operator Π because while r is a (region) type, the body of the abstraction, *Const r* is a *kind*. This gives us "dependent kinds". This can be contrasted to systems with "dependent types", which have functions from values to *types*.

To ensure that both $\underline{\text{const } \rho_n}$ and $\underline{\text{mutable } \rho_n}$ for a particular $\underline{\rho_n}$ cannot occur in the same program, we require the mutability (or constancy) of a region to be set at the point it is created. This is done by extending the **letregion** construct with an optional set of witness bindings. For example, the following function computes the length of a list by destructively incrementing a local accumulator, then copying out the final value.

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

where

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

The set after the **with**-keyword binds the type expression that produces a witness of mutability for r_3. Similarly to the case with region handles, there is a phase distinction between type expressions that *construct* witnesses, like *MkMutable* r_3, and the actual normal form witnesses, like $\underline{\mathit{mutable}\ \rho_3}$. In our reduction semantics the former reduces to the latter, and only the former may appear in the initial program text. We discuss the details of reduction in §4.

To ensure that "rogue" witnesses cannot be created, we place three simple syntactic restrictions on how the constructors *MkMutable* and *MkConst* may be used:

1. They may only appear in the set of witness bindings associated with a **letregion**.
2. Either *MkMutable* or *MkConst* may be used in the set, but not both.
3. If the **letregion** binds a region variable r, then only that region variable may be used in the associated set.

An alternative to these rules would be to define two separate forms of **letregion**, perhaps **letconstregion** and **letmutableregion**. However, we stick with using a generic **letregion**,

and a set of witness bindings, because it will be easier to extend with the witnesses of no-aliasing discussed in §6.

Returning to the definition of *length*, note that as we call *incInt* to destructively increment the integer value in r_3, this region must be mutable. This fact is encoded by the *Mutable* constraint in the type for *incInt*, which we satisfy by passing in our witness *w*. If the region r_3 happened to be constant instead of mutable, then by the syntactic restrictions on **letregion**, we would not have a suitable witness of mutability to pass to *incInt*. This provides the constraint exclusivity property discussed in §2.2

3.3 Laziness and Witnesses of Purity

As discussed in §2.5, although our language uses call-by-value evaluation as default, we can also suspend an arbitrary function application with the suspend operator:

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

The constraint *Pure e* indicates that only observably pure applications may be suspended, so when we call *suspend* we must pass a witness to this fact. Witnesses of purity are written $\underline{\text{pure}\ \sigma}$ where σ is some effect. They can be created with the $MkPurify$ witness constructor, which has the following kind:

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

This constructor takes a witness that a particular region is constant, and produces a witness that a read from it is pure. Reads of constant regions are pure because it does not matter when the read takes place, the same value will be returned each time. Here is an example that uses $MkPurify$:

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

This function simply suspends an application of *succ* to its integer argument. As the argument appears in a region named r_1, computing its successor causes the effect $Read\ r_1$. For this effect to be pure, we require r_1 to be constant, hence the $Const\ r_1$ constraint in the type of *lazySucc*. A caller of *lazySucc* must pass a witness that satisfies this constraint, which we bind to *w*, and then use to construct a witness that $Read\ r_1$ is pure. We call this process *purifying* an effect, and the constraint $Const\ r_1$ is called the *purifier* of $Read\ r_1$.

Alternatively, if the effect of the application to be suspended is simply $\bot$, then we can introduce a witness that it is pure by using the following witness constructor:

$$MkPureBot \quad :: \quad Pure\ \bot$$

Note that there are several ways of writing the effect of a pure function. Firstly, the effect $\bot$ is manifestly pure, as it contains no atomic effect terms. However, we can also treat any other effect as pure provided we can produce a witness of the appropriate kind. For example, $Read\ r_5$ is pure if we can produce a witness of kind $Pure\ (Read\ r_5)$. This relationship is encoded by our ObsPure typing rule, which we discuss further in §4.6

3.4 Witness Joining and Higher Order Functions

Purity constraints extend naturally to higher order functions. For example, here is a lazy version of the map function that constructs the first element of the resulting list when it is called, but leaves subsequent elements to be constructed when they are demanded.

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

The inner **case**-expression in this definition has the effect $\mathit{Read}\ r_1 \vee e$. The first part is due to inspecting the input list, and the second is due to the application of the argument function f to the element x. However, as the recursive call to *lazyMap* is suspended, *lazyMap* itself must be pure. We prove this fact by using two existing witnesses. The first, w_1, guarantees that r_1 is constant, and by applying $\mathit{MkPurify}$ to it we get a witness that $\mathit{Read}\ r_1$ is pure. The second, w_2, guarentees that the effect bound to the variable e is pure. We join these two witnesses into a compound one with the following constructor:

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

In the definition of *lazyMap*, the use of *MkPureJoin* gives the witness passed to *suspend* the following kind, which satisfies the purity constraint:

$$\mathit{MkPureJoin}\ (\mathit{Read}\ r_1)\ e\ (\mathit{MkPurify}\ r_1\ w_1)\ w_2 \quad :: \quad \mathit{Pure}\ (\mathit{Read}\ r_1 \vee e)$$

3.5 Explicit effect masking

In the previous section, although we were able to construct a witness that *lazyMap* was pure, its type signature still contained the effect term $\mathit{Read}\ r_1 \vee e$. Alternatively, we can give *lazyMap* a manifestly pure effect by using explicit effect masking:

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

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. As $lazyMap'$ itself is now manifestly pure, we can use the *MkPureBot* constructor to satisfy the purity constraint on *suspend*.

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. Starting at the top of the strata we use super-kinds to classify witness kind constructors, and to ensure they are applied to the right kind of type. For our three baked-in constructors we have the following super-kinds:

$$\begin{array}{lll} \mathit{Const} & :: & \% \to \Diamond \\ \mathit{Mutable} & :: & \% \to \Diamond \\ \mathit{Pure} & :: & ! \to \Diamond \end{array}$$

The first signature says that *Const* may only be applied to a region type, such as with $\mathit{Const}\ r_1$. The result of a witness kind constructor is always $\Diamond$, pronounced "prop". A signature such as $\mathit{Const} :: \% \to \Diamond$ is read "a witness classified by *Const* guarantees a property of a region". We use $\Box$, pronounced "box" as the super-kind for kinds that do not encode such a property, such as $\%$ and $* \to *$.

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 a normal form witness such as $\underline{\rho}$, $\underline{\text{const } \rho}$, $\underline{\text{mutable } \rho}$ or $\underline{\text{pure } \sigma}$. We use φ_i to refer to an arbitrary 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. We require the alternatives of a case-expression to be exhaustive, and data constructors, *suspend* and $update$ to be fully applied. 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.

4.1 Typing Rules

In Fig. 2 the judgement form $\Gamma \mid \Sigma \vdash_\kappa \kappa :: \omega$ reads: with type environment Γ and store typing Σ, kind κ has super-kind ω. The store typing is an abstract model of the current state of the store, and is discussed further in §4.5. KsAbs is the rule for dependent kind abstraction. Note that a kind signature such as $\% \to *$ is desugared to this form, resulting in $\Pi(\_ : \%).*$.

In Fig. 3 the judgement form $\Gamma \mid \Sigma \vdash_\tau \varphi :: \kappa$ reads: with type environment Γ and store typing Σ, type φ has kind κ. Rules KiApp, KiVar and KiAll are standard. Note that KiJoin and KiBot are defined on effect types only. The remainder of the rules give kinds for our "baked-in" constructors. We have included *List* and *Int* as representative data type

Symbol Classes

a, r, e, w	$\rightarrow$ (type variables)	T	$\rightarrow$ (data type constructors)
x, y	$\rightarrow$ (value variables)	K	$\rightarrow$ (data constructors)
l	$\rightarrow$ (store locations)	ρ	$\rightarrow$ (region labels)

Super Kinds

$\omega ::= \Box \mid \Diamond \mid \kappa \rightarrow \omega$

Kinds

$\kappa ::= \kappa\ \varphi \mid \Pi(a : \kappa_1).\ \kappa_2$	(kinds)
$\mid\ *\ \mid\ \%\ \mid\ !$	(atomic kinds)
$\mid\ Const \mid Mutable \mid Pure$	(witness kind constrs)

Types

$\varphi, \tau, \sigma, \delta, \Delta$

$::= a \mid \forall(a : \kappa).\ \tau \mid \varphi_1\ \varphi_2$	(types)
$\mid\ (\rightarrow) \mid () \mid T$	(data type constrs)
$\mid\ \sigma_1 \vee \sigma_2 \mid \bot \mid Read \mid Write$	(effect type constrs)
$\mid\ MkConst \mid MkMutable \mid MkPure \mid MkPurify \mid MkPureJoin$	(witness type constrs)
$\mid\ \underline{\rho} \mid \underline{\text{const}\ \varphi} \mid \underline{\text{mutable}\ \varphi} \mid \underline{\text{pure}\ \sigma}$	(witness types)

Terms

$t ::= x \mid v \mid t\ \varphi \mid t_1\ t_2 \mid$ **letregion** r **with** $\{\overline{w = \delta}\}$ **in** $t \mid K\ \overline{\varphi}\ \overline{t}$
$\mid$ **case** t **of** $\overline{K\ \overline{x : \tau} \rightarrow t'} \mid update_{K,i}\ \overline{\varphi}\ t_1\ t_2 \mid suspend\ \overline{\varphi}\ t_1\ t_2$
$\mid$ **mask** δ **in** t

$v^\circ, u^\circ ::= v \mid suspend\ \overline{\varphi}\ v_1^\circ\ v_2^\circ$	(weak values)
$v,\ u ::= l \mid () \mid \Lambda(a : \kappa).\ t \mid \lambda(x : \tau).\ t$	(strong values)

Derived Forms

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

Type Environment

$\Gamma ::= a : \kappa \mid x : \tau \mid \Gamma_1, \Gamma_2$

Store Typing

$\Sigma ::= l : \tau,\ \underline{\rho} \mid \underline{\text{const}\ \rho} \mid \underline{\text{mutable}\ \rho} \mid r \sim \underline{\rho} \mid \Sigma_1, \Sigma_2$

Fig. 1. Core Language

constructors, though in a full language the programmer would be able to define their own. The function type constructor $(\rightarrow)$ and the unit type constructor $()$ are special because they are referred to directly in the typing rules of Fig. 4. We discuss the kinds of witness types in §4.5.

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. Rules TyVar through TyApp are standard. In TyLetRegion the premise "$\overline{\delta_i}$ well formed" refers to the syntactic restrictions on the introduced witnesses that were discussed in §3.2.

Note the premise $r \notin fv(\tau)$, also in TyLetRegion. In an entirely strict language this premise would allow us to allocate storage for the region r in a stack-like manner, freeing it once the body of the **letregion** had finished evaluating (Tofte *et al.*, 2006). This would be possible because when the variable r does not appear in the type of the result (or in the type environment), objects in this region are not reachable from the calling context. This is not true in a language that supports laziness, such as ours. In our case a "value"

$$\boxed{\Gamma \mid \Sigma \vdash_{\kappa} \kappa :: \omega}$$

$$\frac{\Gamma \mid \Sigma \vdash_{\kappa} \kappa_1 :: \kappa_{11} \to \omega \quad \Gamma \mid \Sigma \vdash_{\kappa} \varphi :: \kappa_{11}}{\Gamma \mid \Sigma \vdash_{\kappa} \kappa_1\ \varphi :: \omega}\ (\text{KsApp}) \qquad \frac{\kappa \in \{*,\ \%,\ !\}}{\Gamma \mid \Sigma \vdash_{\kappa} \kappa :: \Box}\ (\text{KsAtom})$$

$$\frac{\Gamma \mid \Sigma \vdash_{\kappa} \kappa_1 :: \omega_1 \quad \Gamma,\ a : \kappa_1 \mid \Sigma \vdash_{\kappa} \kappa_2 :: \omega_2}{\Gamma \mid \Sigma \vdash_{\kappa} \Pi(a : \kappa_1).\ \kappa_2 :: \omega_2}\ (\text{KsAbs})$$

$$\Gamma \mid \Sigma \vdash_{\kappa} \mathit{Const} :: \% \to \Diamond \quad \Gamma \mid \Sigma \vdash_{\kappa} \mathit{Mutable} :: \% \to \Diamond \quad \Gamma \mid \Sigma \vdash_{\kappa} \mathit{Pure} :: ! \to \Diamond$$

Fig. 2. Super-kinds of Kinds

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

$$\frac{\Gamma \mid \Sigma \vdash_{\tau} \varphi_1 :: \Pi(a : \kappa_1).\ \kappa_2 \quad \Gamma \mid \Sigma \vdash_{\tau} \varphi_2 :: \kappa_1}{\Gamma \mid \Sigma \vdash_{\tau} \varphi_1\ \varphi_2 :: \kappa_2[\varphi_2/a]}\ (\text{KiApp}) \qquad \Gamma,\ a : \kappa \mid \Sigma \vdash_{\tau} a :: \kappa\ (\text{KiVar})$$

$$\frac{\Gamma \mid \Sigma \vdash_{\kappa} \kappa_1 :: \omega_1 \quad \Gamma,\ a : \kappa_1 \mid \Sigma \vdash_{\tau} \tau_2 :: \kappa_2}{\Gamma \mid \Sigma \vdash_{\tau} \forall(a : \kappa_1).\ \tau_2 :: \kappa_2}\ (\text{KiAll})$$

$$\frac{\Gamma \mid \Sigma \vdash_{\tau} \sigma_1 :: ! \quad \Gamma \mid \Sigma \vdash_{\tau} \sigma_2 :: !}{\Gamma \mid \Sigma \vdash_{\tau} \sigma_1 \vee \sigma_2 :: !}\ (\text{KiJoin}) \qquad \Gamma \mid \Sigma \vdash_{\tau} \bot :: !\ (\text{KiBot})$$

$$\begin{array}{llll}
\Gamma \mid \Sigma \vdash_{\tau} (\to) & :: * \to * \to ! \to * & \Gamma \mid \Sigma \vdash_{\tau} () & :: * \\
\Gamma \mid \Sigma \vdash_{\tau} \mathit{List} & :: \% \to * \to * & \Gamma \mid \Sigma \vdash_{\tau} \mathit{Int} & :: \% \to * \\
\Gamma \mid \Sigma \vdash_{\tau} \mathit{Read} & :: \% \to ! & \Gamma \mid \Sigma \vdash_{\tau} \mathit{Write} & :: \% \to ! \\
\Gamma \mid \Sigma \vdash_{\tau} \mathit{MkConst} & :: \Pi(r : \%).\ \mathit{Const}\ r & & \\
\Gamma \mid \Sigma \vdash_{\tau} \mathit{MkMutable} & :: \Pi(r : \%).\ \mathit{Mutable}\ r & \Gamma \mid \Sigma \vdash_{\tau} \mathit{MkPureBot} & :: \mathit{Pure}\ \bot \\
\Gamma \mid \Sigma \vdash_{\tau} \mathit{MkPurify} & :: \Pi(r : \%).\ \mathit{Const}\ r \to \mathit{Pure}\ (\mathit{Read}\ r) & & \\
\Gamma \mid \Sigma \vdash_{\tau} \mathit{MkPureJoin} & :: \Pi(e_1 : !).\ \Pi(e_2 : !).\ \mathit{Pure}\ e_1 \to \mathit{Pure}\ e_2 \to \mathit{Pure}(e_1 \vee e_2) & &
\end{array}$$

Fig. 3. Kinds of Types

with a type such as *Int* r_1 may be represented as a suspended function application, and the suspension may contain references to objects in regions besides r_1. However, we have included the $r \notin fv(\tau)$ premise anyway because it makes the proof of soundness easier. This is discussed further in §4.7.

In TyCase, the act of inspecting the discriminant causes the effect *Read* φ, where φ corresponds to the data type's first (primary) region variable. The primary variable corresponds to the region containing the outer constructor of the object, which was discussed in §2.4.

In TyUpdate, we must pass the *update* operator a witness δ of kind *Mutable* φ. This witness proves that the region holding the constructor to be updated is indeed mutable. Performing the update causes a write effect on this same region. Note that we use the symbol φ to represent the region as it can be instantiated by both region variables r and handles $\underline{\rho}$. The meta-function ctorTypes(T) returns a set containing the types of all data constructors associated with a type constructor T.

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

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

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

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

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

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

$$\frac{\begin{array}{ll} \overline{\delta_i} \text{ well formed} & \overline{\Gamma \mid \Sigma \vdash_{\mathrm{T}} \delta_i :: \kappa_i}^{\,i} \qquad \overline{\Gamma \mid \Sigma \vdash_{\kappa} \kappa_i :: \Diamond}^{\,i} \\ r \notin fv(\tau) & \Gamma, r : \%, \overline{w_i : \kappa_i} \mid \Sigma \vdash t :: \tau ; \sigma \end{array}}{\Gamma \mid \Sigma \vdash \textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t :: \tau ; \sigma} \ \text{(TyLetRegion)}$$

$$\frac{\begin{array}{l} \overline{\Gamma \mid \Sigma \vdash t :: \tau_i[\varphi/r\ \overline{\varphi'/a}] ; \sigma_i}^{\,i \leftarrow 0..n} \\ \Gamma \mid \Sigma \vdash_{\mathrm{T}} \varphi :: \% \quad K :: \forall (r : \%).\forall (\overline{a : \kappa}).\overline{\tau} \rightarrow T\ r\ \overline{a} \in \mathrm{ctorTypes}(T) \end{array}}{\emptyset \mid \Sigma \vdash K\ \varphi\ \overline{\varphi'}\ \overline{t} :: T\ \varphi\ \overline{\varphi'} ; \sigma_0 \vee \sigma_1 \ldots \vee \sigma_n} \ \text{(TyData)}$$

$$\frac{\Gamma \mid \Sigma \vdash t :: T\ \varphi\ \overline{\varphi'} ; \sigma \quad \overline{\Gamma \mid \Sigma \vdash p_i \rightarrow t_i :: T\ \varphi\ \overline{\varphi'} \rightarrow \tau ; \sigma_i'}^{\,i \leftarrow 0..n}}{\Gamma \mid \Sigma \vdash \textbf{case}\ t\ \textbf{of}\ \overline{p \rightarrow t} :: \tau ; \sigma \vee \mathit{Read}\ \varphi \vee \sigma_0' \vee \sigma_1' \ldots \vee \sigma_n'} \ \text{(TyCase)}$$

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

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

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

$$\Gamma \mid \Sigma \vdash () :: () ; \bot \ \text{(TyUnit)}$$

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

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

Fig. 4. Types of Terms

In TySuspend, we must pass the *suspend* operator a witness δ of kind *Pure* σ. This proves that the application to be suspended is pure. Note that although the application itself must be pure, the expressions that compute the function and argument need not be, hence the effect term $\sigma_1 \vee \sigma_2$.

TyMaskPure allows us to explicitly mask an effect, provided we can produce a witness of its purity. This was discussed in §3.5.

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 weak values $\overline{v^\circ}$. Each binding is annotated with a region label ρ that specifies the region it belongs to.

The store also contains *properties* that specify how bindings in the various regions may be used. Properties are named after our witnesses from Fig. 1, and are ρ, $(\text{const } \rho)$ and $(\text{mutable } \rho)$. 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. The last two indicate whether a binding in a region may be treated as constant, or updated. Note the difference between witnesses and properties. Although the two are written similarly, witnesses occur in the term being reduced, whereas properties occur in the store. In our notation, witnesses are always written with an underline, while properties are not.

l	$\rightarrow$	(store location)	
ρ	$\rightarrow$	(region handle)	
o	$::=$	$\rho \mid \text{const } \rho \mid \text{mutable } \rho$	(property)
μ	$::=$	$C_K\ \overline{v^\circ}$	(store object)
H	:	$\{\ l \overset{\rho}{\mapsto} \mu\ \} + \{\ o\ \}$	(store)

Fig. 5. Stores and Store Objects

4.3 Witness Construction

Fig. 6 gives the relationship between properties, witnesses and witness constructors. 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 associated property is *not* set, then the evaluation becomes stuck. Note that although we use this notion of type-level "evaluation" in our semantics, it is not needed at runtime. In our compiler, witnesses are erased during code generation along with all other type information.

The axiom EwPureBot produces a witness that $\bot$ is a pure effect. We need this axiom because although witness constructors like *MkPureBot* may appear in the source program, witnesses like $\underline{\text{pure } \bot}$ may not. The rules EwPurify and EwPureJoin are used to construct a new witness from existing ones. The first takes a witness that ρ is constant, and produces

$$\boxed{H\,;\,\delta \rightsquigarrow \delta'}$$

$$E_w ::= [\,] \mid \mathit{MkPurify}\ \underline{\rho}\ E_w \mid \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ E_w\ E_w$$

$$\frac{H\ ;\ \delta \rightsquigarrow \delta'}{H\ ;\ E_w[\delta] \rightsquigarrow E_w[\delta']}\ \text{(EwContext)}$$

$$\begin{array}{rcll}
H[\,\mathit{const}\ \rho\,]\ ;\ \mathit{MkConst}\ \underline{\rho} & \rightsquigarrow & \underline{\text{const}\ \rho} & \text{(EwConst)} \\
H[\,\mathit{mutable}\ \rho\,]\ ;\ \mathit{MkMutable}\ \underline{\rho} & \rightsquigarrow & \underline{\text{mutable}\ \rho} & \text{(EwMutable)} \\
H\ ;\ \mathit{MkPureBot} & \rightsquigarrow & \underline{\text{pure}\ \bot} & \text{(EwPureBot)} \\
H\ ;\ \mathit{MkPurify}\ \underline{\rho}\ \underline{\text{const}\ \rho} & \rightsquigarrow & \underline{\text{pure}\ (\mathit{Read}\ \rho)} & \text{(EwPurify)} \\
H\ ;\ \mathit{MkPureJoin}\ \sigma_1\ \sigma_2\ \underline{\text{pure}\ \sigma_1}\ \underline{\text{pure}\ \sigma_2} & \rightsquigarrow & \underline{\text{pure}\ (\sigma_1 \vee \sigma_2)} & \text{(EwPureJoin)}
\end{array}$$

Fig. 6. Witness Construction

a witness that a read from this region is pure. The second takes witnesses that two separate effects are pure, and produces a witness that their sum is pure.

4.4 Term Evaluation

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 second premise of EvLetRegion is always true, and is used to convert the given witness expressions $\overline{\delta_i}$ to their associated witnesses $\overline{\Delta_i}$. EvAlloc allocates a new binding into the store, which is tagged with the region label applied to its constructor. EvUpdateBind says that to update the value of a binding in region ρ we must pass an appropriate witness $\underline{\text{mutable}\ \rho}$, and the corresponding property $(\text{mutable}\ \rho)$ must be present in the store. In our soundness theorem discussed in §4.6, the fact that an update expression of this form can always progress ensures that the binding is indeed mutable. Similarly, in EvSuspendApp we must supply a witness $\underline{\text{pure}\ \sigma}$ that the application to be suspended is pure. EvUpdateFail handles the case where we try to update a binding using an update operator for the wrong data constructor. As different constructors can have a different number of parameters, of different types, we halt the program with the special value "fail". EvMask shows that a **mask** expression has no operational effect, besides requiring an appropriate witness of purity.

4.5 Store typings

The store typing Σ models the state of the heap as the program progresses, and was defined in Fig. 1. The store typing contains the type of each store location, along with witnesses to the current set of store properties. It also contains region similarity bindings $r \sim \underline{\rho}$ which we use to account for the fact that region handles are substituted for region variables in our proof of Preservation. This is discussed further in §4.7.

We say that 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*

$$\boxed{H\,;\,t \longrightarrow H\,;\,t'}$$

$$\begin{array}{rcll} E_v & ::= & [\,] \mid E_v\ \varphi \mid E_v\ t_2 \mid v\ E_v \mid \textbf{case}\ E_v\ \textbf{of}\ \overline{alt} \\ & \mid & K\ \overline{\varphi}\ E_v\ t_1\ \ldots & \mid K\ \overline{\varphi}\ v^\circ\ E_v\ \ldots \mid \ldots \\ & \mid & update_{K,i}\ \overline{\varphi}\ E_v\ t_2 & \mid update_{K,i}\ \overline{\varphi}\ l\ E_v \\ & \mid & suspend\ \overline{\varphi}\ E_v\ t_2 & \mid suspend\ \overline{\varphi}\ v\ E_v \end{array}$$

$$\frac{H\,;\,t \longrightarrow H'\,;\,t'}{H\,;\,E_v[t] \longrightarrow H'\,;\,E_v[t']}\ \text{(EvContext)}$$

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

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

$$\frac{H' = H,\ \rho,\ \overline{\text{propOf}(\Delta_i)} \qquad \overline{H'\,;\,\delta_i[\underline{\rho}/r] \rightsquigarrow \Delta_i} \qquad \rho\ \text{fresh}}{H\,;\,\textbf{letregion}\ r\ \textbf{with}\ \{\overline{w_i = \delta_i}\}\ \textbf{in}\ t \longrightarrow H'\,;\,t[\underline{\rho}/r\ \overline{\Delta_i/w_i}]}\ \text{(EvLetRegion)}$$

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

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

$$\begin{array}{rl} H[\text{ mutable } \rho,\ l \overset{\rho}{\mapsto} C_K\ \overline{v^\circ}]\ ;\ update_{K,i}\ \underline{\rho}\ \overline{\varphi}\ \underline{\text{mutable } \rho}\ \ l\ u^\circ & \\ \longrightarrow H,\ l \overset{\rho}{\mapsto} C_K\ v_0^\circ..u_i^\circ..v_n^\circ\ ;\ () & \text{(EvUpdateBind)} \end{array}$$

$$\begin{array}{rll} H[\text{ mutable } \rho,\ l \overset{\rho}{\mapsto} C_K\ \overline{v^\circ}]\ ;\ update_{K',i}\ \underline{\rho}\ \overline{\varphi}\ \underline{\text{mutable } \rho}\ \ l\ u^\circ & & \\ \longrightarrow H\ ;\ \text{fail} \qquad \overline{K \neq K'} & \text{(EvUpdateFail)} \end{array}$$

$$\frac{H\,;\,\delta \rightsquigarrow \delta'}{H\,;\,suspend\ \overline{\varphi}\ \delta\ t\ t' \longrightarrow H\,;\,suspend\ \overline{\varphi}\ \delta'\ t\ t'}\ \text{(EvSuspendWit)}$$

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

$$\frac{H\,;\,\delta \rightsquigarrow \delta'}{H\,;\,\textbf{mask}\ \delta\ \textbf{in}\ t \longrightarrow H\,;\,\textbf{mask}\ \delta'\ \textbf{in}\ t}\ \text{(EvMaskWit)}$$

$$H\,;\,\textbf{mask}\ \underline{\text{pure } \sigma}\ \textbf{in}\ t \longrightarrow H\,;\,t \quad \text{(EvMaskApp)}$$

Fig. 7. Term Evaluation

typed, and write $\Sigma \vdash H$ when it contains all the bindings and properties predicted by the store typing. These are defined formally in the appendix. Both the store and store typing grow as the program evaluates, and neither bindings, properties or witnesses are removed once added.

Although most of the typing rules from Figs 2, 3 and 4 simply pass the store typing through unchanged, it is used in TyLoc from Fig. 4 to determine type of a store location l. However, as kind expressions can contain types, and hence store locations as well, the store typing must be a part of all judgement forms.

Store typings are also used in Fig. 8, which gives kinds to witnesses. The judgement form $\Gamma \mid \Sigma \vdash_{\text{T}} \Delta :: \kappa$ reads: with type environment Γ and store typing Σ, witness Δ has kind κ. This is similar to the one in Fig. 3. Importantly, in the first four rules in Fig. 8, when we give a kind to a witness we *require* that witness to also be in the store typing. Provided the store typing models the store, this also means that the corresponding property is in the

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

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

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

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

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

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

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

Fig. 8. Kinds of Witnesses

store. This is the crux of our soundness proof: when a witness appears in the term, the corresponding property is guaranteed to be present in the store.

4.6 Soundness

Our proof of soundness is split into Progress and Preservation (subject reduction) theorems in the usual way.

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

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

In the Progress Theorem, "nofab" is short for "no fabricated region witnesses", and refers to the syntactic constraint discussed in §3.2 that *MkConst* and *MkMutable* may only appear in the witness binding of a **letregion**, and not elsewhere in the program.

In the Preservation Theorem, note that the latent effect of the term reduces as the program progresses. The $\sqsubseteq_\Sigma$ relationship on effects is defined in the obvious way, apart from two points. The first is that we allow region handles to be substituted for region variables if there is a corresponding $r \sim \underline{\rho}$ binding in the store typing. The second is that we add the following extra rule:

$$\frac{\emptyset \mid \Sigma \vdash_{\mathrm{T}} \delta :: \mathit{Pure}\ \sigma}{\sigma \sqsubseteq_\Sigma \bot}\ \text{(ObsPure)}$$

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. Formal proofs of Progress and Preservation are given in the appendix.

As mentioned in the previous section, our typing rules ensure that if a witness occurs in the term being reduced, then it also occurs in the store typing. Provided the store typing models the store, this also means that the corresponding property is present in the store.

From the evaluation rules of Fig. 7, the only term that adds properties to the store is **letregion**, and when it does, it also introduces the corresponding witnesses into its body. The well-formedness restriction on **letregion** guarantees that a witness 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 $(\text{mutable}\ \rho)$ 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.

As a final point, note that although we support destructive update of arbitrary data, we do not suffer the potential unsoundness commonly associated with ML-style mutable references (Garrigue, 2004). The fact that type abstractions are explicit naturally gives our language *polymorphism-by-name* semantics (Leroy, 1993). Polymorphic expressions are re-evaluated at each use, after call-by-name, and a given mutable object cannot be updated in an unsound way.

4.7 Allocation and Region Phase Changes

As opposed to the region calculus of (Tofte *et al.*, 2006), we cannot use the region annotations in our program to allocate and deallocate storage in a stack-like manner. To see why this is, consider the following reduction:

$$\begin{array}{rlcl} & \rho_1 & ; & \textbf{letregion}\ r_2\ \textbf{in let}\ x = 23\ r_2\ \textbf{in} \\ & & & \mathit{suspend}\ (\mathit{Int}\ r_2)\ (\mathit{Int}\ \underline{\rho_1})\ \ldots\ (\mathit{succ}\ r_2\ \underline{\rho_1})\ x \\ \xrightarrow{*} & \rho_1,\ \rho_2,\ l \overset{\rho_2}{\mapsto} 23 & ; & \mathit{suspend}\ (\mathit{Int}\ \underline{\rho_2})\ (\mathit{Int}\ \underline{\rho_1})\ \ldots\ (\mathit{succ}\ \underline{\rho_2}\ \underline{\rho_1})\ l \end{array}$$

The initial expression creates a fresh region and allocates a heap binding for the value 23. It then suspends an application of *succ* to this value. In the resulting expression, note that although we have eliminated the **letregion** construct, we have retained a reference to the new region via heap location l. At this point we cannot simply remove (deallocate) the binding $l \overset{\rho_2}{\mapsto} 23$ from the heap, as then l would be out of scope.

Although we cannot use region based memory management, we retain the $r \notin fv(\tau)$ premise in the TyLetRegion rule of Fig. 4. If we had not done this then the initial expression in the following reduction would be well-typed:

$$\begin{array}{rlcl} & \emptyset & ; & \textbf{letregion}\ r_1\ \textbf{in}\ \mathit{succ}\ r_1\ r_1\ (23\ r_1) \\ \longrightarrow & \rho_1 & ; & \mathit{succ}\ \underline{\rho_1}\ \underline{\rho_1}\ (23\ \underline{\rho_1}) \\ \longrightarrow & \rho_1,\ l_1 \overset{\rho_1}{\mapsto} 23, & ; & \mathit{succ}\ \underline{\rho_1}\ \underline{\rho_1}\ l_1 \\ \longrightarrow & \rho_1,\ l_1 \overset{\rho_1}{\mapsto} 23,\ l_2 \overset{\rho_1}{\mapsto} 24 & ; & l_2 \end{array}$$

Note that while the initial expression has type $\mathit{Int}\ r$, the final one has type $\mathit{Int}\ \underline{\rho}$. This reveals the phase difference between region variables and region handles. On one hand, maintaining this difference is convenient when reasoning about the soundness of optimisations. On the other hand, if we were to permit this example to be well-typed then we would need to weaken our Preservation theorem so that the result of a reduction only had to have a type *similar* to the initial expression. By this we mean that we allow region handles to be substituted for their corresponding region variables. Note that the effect of an

expression also undergoes the phase change. Whereas the effect of the initial expression is *Read* r_1, the effect of the second is *Read* $\underline{\rho}$. In our previous work of (Lippmeier, 2010) we used the $r \sim \underline{\rho}$ elements of the store typing to define similarity judgements for both types and effects. However, in this work we have chosen to simplify our typing rules and proof by retaining the $r \notin fv(\tau)$ premise in TyLetRegion and rejecting expressions like the one above. We still need a similarity judgement for effects, but this is easier to manage, and we have left the details to the appendix.

4.8 Observation Criterion

The observation criterion says that if the reduction of some expression has an effect, but that effect is not visible to the calling context, then we can safely ignore it. An example of this was given in §3.1. Although there is a singular observation criterion, there are multiple ways of implementing it, depending on our interpretation of "not visible". For our system a simple rule is as follows:

$$\frac{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma \quad r \notin fv_T(\Gamma) \quad r \notin fv(\tau)}{\Gamma \mid \Sigma \vdash t :: \tau ; \sigma \setminus (Read\ r \vee Write\ r)} \qquad \text{(TyObs)}$$

This rule says that if an expression has a read or write effect on some region variable r, but that variable is not free in the type environment or the type of the expression, then we can remove it from subsequent consideration. The premise $r \notin fv_T(\Gamma)$ refers to free variables in type bindings only. For example, we allow a plain $r : \%$ to appear in the environment provided r it is not also free in the τ' of any $x : \tau'$. The reasoning why TyObs is valid is standard and fashioned after (Talpin & Jouvelot, 1992). As r does not appear in any type binding in the environment, the term t cannot hold a direct reference to pre-existing objects in this region. It may hold an indirect reference via a suspended function application, but as suspended applications must be pure their final values cannot be affected once the suspension is created. As r does not appear in the return type, we cannot hold a direct reference to any object in r after the term has finished evaluating.

Although TyObs is valid, we have omitted it from Fig. 4 because our soundness proof does not cover it. Unlike the other rules, TyObs is not syntax directed, which complicates both type checking and case analysis in the proof. However, in our practical implementation it suffices to invoke TyObs only when checking an abstraction, effectively combining it with TyAbs. Due to this we define a new rule that combines the two:

$$\frac{\Gamma,\, x : \tau_1 \mid \Sigma \vdash t_2 :: \tau_2 ; \sigma \quad \begin{array}{l} \overline{r_i \notin fv_T(\Gamma) \cup fv(\tau_1) \cup fv(\tau_2)}^{\,i} \\ \sigma' = \sigma \setminus \overline{(Read\ r_i \vee Write\ r_i)}^{\,i} \end{array}}{\Gamma \mid \Sigma \vdash \lambda(x : \tau_1).t_2 :: \tau_1 \xrightarrow{\sigma'} \tau_2 ; \bot} \qquad \text{(TyAbsMask)}$$

Here, we have folded in multiple instances of TyObs so that effects on several region variables can be masked at the same point. Finally, note that as usual, our proof of "soundness" simply ensures that the reduction of an expression does not "get stuck". It does not, say, guarantee that optimising transforms based on masked effect information preserve the meaning of the original program. Progress and Preservation provide a basic foundation for

such reasoning, but it is a topic for future work. See for example (Benton *et al.*, 2007) and (Naumann, 2007) which address this.

5 Sharing, Purity and the Full Laziness Transform

In this section we consider the *full-laziness* transform (Peyton Jones & Lester, 1995) as an example optimisation using our system. In the following function, note that if we could lift the n binding out of the inner function then we could save its recomputation for each application. We have elided some of the type annotations to make the presentation clearer.

$$
\begin{array}{l}
\Lambda r_3\ e_1.\ \lambda(g : \mathit{Int}\ r_3 \xrightarrow{e_1} \mathit{Int}\ r_3). \\
\textbf{letregion}\ r_4\ \textbf{with}\ w_1 = \mathit{MkConst}\ r_4\ \textbf{in} \\
\textbf{let}\quad (xs : \mathit{List}\ r_4\ (\mathit{Int}\ r_3)) = \ldots \\
\qquad f \quad = \lambda y.\ \textbf{let}\ n\ =\ g\ (\mathit{length}\ xs)\ \textbf{in}\ n\ +\ y \\
\textbf{in}\quad \mathit{map}\ f\ xs
\end{array}
$$

We will specialise this function for the case when g is pure. This will allow us to perform the lift, but as we cannot guarantee that f will be applied at least once, we wrap the lifted expression in a suspension to guard against the case where it diverges. Doing this requires a witness that the n binding is pure. We will assume that *length* has the same type as in §3.2, namely:

$$\mathit{length}\ ::\ \forall a\ r_1\ r_2.\ \mathit{List}\ r_1\ a \xrightarrow{\mathit{Read}\ r_1} \mathit{Int}\ r_2$$

As type of the list xs is $\mathit{List}\ r_4\ (\mathit{Int}\ r_3)$, applying *length* to it causes the effect $\mathit{Read}\ r_4$. Applying the function g to the result then causes the effect e_1, so the effect of the whole n binding is $\mathit{Read}\ r_4 \vee e_1$. We know that $\mathit{Read}\ r_4$ is pure due to the witness $\mathit{MkConst}\ r_4$. We can then require e_1 to be pure by adding a new type parameter that binds an appropriate witness of purity. This ensures that the specialised version of our function can only be applied to an argument that meets this constraint. The result of the transform is as follows:

$$
\begin{array}{l}
\Lambda r_3\ e_1.\ \Lambda(w_2\ :\ \mathit{Pure}\ e_1).\ \lambda(g : \mathit{Int}\ r_3 \xrightarrow{e_1} \mathit{Int}\ r_3). \\
\textbf{letregion}\ r_4\ \textbf{with}\ w_1 = \mathit{MkConst}\ r_4\ \textbf{in} \\
\textbf{let}\quad (xs : \mathit{List}\ r_4\ (\mathit{Int}\ r_3)) = \ldots \\
\qquad n \quad = \mathit{suspend}\ (\mathit{MkPureJoin}\ (\mathit{MkPurify}\ r_4\ w_1)\ w_2)\ (\lambda\_.\ g\ (\mathit{length}\ xs))\ () \\
\qquad f \quad = \lambda y.\ n\ +\ y \\
\textbf{in}\quad \mathit{map}\ f\ xs
\end{array}
$$

5.1 Idempotency and Allocation Effects

In general, if we wish to lift a binding out of an enclosing lambda abstraction, that binding must satisfy two constraints. The first is that it can be safely reordered with other expressions in the abstraction, and the second is that it is idempotent. The first is needed because moving the binding changes the order of evaluation relative to the original abstraction. The second is needed because the lifted expression will tend to be evaluated less often, which is the whole purpose of performing the transform. Note that a function that returns a freshly allocated object is not idempotent. Now, although related work such as (Talpin & Jouvelot,

1992) and (Benton *et al.*, 2007) uses allocation effects to account for this, we do not. In our system, an expression that produces a fresh object but has no other visible effects is still considered pure. Consider then the following expression:

$$
\begin{array}{lll}
\Lambda r_1.\ \Lambda(w_1 : \mathit{Mutable}\ r_1). & & \\
\textbf{let} & (f\ : () \to \mathit{Int}\ r_1) & = \lambda\_.\ 5\ r_1 \\
& (x\ : \mathit{Int}\ r_1) & = f\ () \\
& \_ & = \mathit{updateInt}\ r_1\ r_1\ w_1\ x\ (42\ r_1) \\
\textbf{in} & f\ () &
\end{array}
$$

This expression defines an inner function f that returns a freshly allocated '5' object each time it is applied. It then applies it, and destructively updates the result. As it stands, the result of the entire expression is also 5. However, if we were to "optimise" it by lifting the inner expression $(5\ r_1)$ out of the abstraction for f then this would change its meaning:

$$
\begin{array}{lll}
\Lambda r_1.\ \Lambda(w_1 : \mathit{Mutable}\ r_1). & & \\
\textbf{let} & (y\ : \mathit{Int}\ r_1) & = 5\ r_1 \\
& (f\ : () \to \mathit{Int}\ r_1) & = \lambda\_.\ y \\
& (x\ : \mathit{Int}\ r_1) & = f\ () \\
& \_ & = \mathit{updateInt}\ r_1\ r_1\ w_1\ x\ (42\ r_1) \\
\textbf{in} & f\ () &
\end{array}
$$

Although our typing rules assign the expression $(5\ r_1)$ the pure effect $\bot$, the meaning of our program has changed, which makes our optimisation invalid. The trouble is that $(5\ r_1)$ is not idempotent, as it allocates a new object each time it is evaluated. However, the fact that it is not idempotent only matters because we destructively update its result. Without destructive update, there is no way to distinguish an expression that returns a fresh object, from one that returns the same object every time.

This reveals that we do not actually need allocation effects to justify the full laziness transform. To lift a given expression out of an abstraction it is sufficient to show that it does not read or write mutable data, and that the resulting value cannot be updated. With this in mind, a poignant question is whether there any useful transformations that can *only* be justified using the information provided by allocation effects. The answer, of course, is yes. Consider the following rewrite from (Benton *et al.*, 2007).

$$
\begin{array}{lll}
 & \textbf{let} & x = \mathit{exp};\ y = \mathit{exp}\ \textbf{in}\ (x,y) \\
\longrightarrow & \textbf{let} & x = \mathit{exp}\ \textbf{in}\ (x,x)
\end{array}
$$

This is valid provided *exp* does not read from any regions that it writes to, and performs no visible allocation. If it were to allocate its return value then we could distinguish the two elements of the pair by updating one and comparing both. However, note that the transform is also valid if *exp* does not read from any regions that it writes to, and returns a value that is constrained to be constant. This yields a weaker transform, though we are not aware of any existing work that quantitatively compares the two. Note that this transform has a similar structure to full-laziness: they both save repeated computation by increasing the sharing of data. In summary, we choose to forgo allocation effects, simplify our analysis, and reduce the volume of effect information. However, we are not aware of any technical barrier to adding them as an extension to the existing system.

6 Local Unboxing and Aliasing

It is well known that purely functional programs support more optimising rewrites than their impure counterparts. The great enablers are the let-floating transforms (Peyton Jones *et al.*, 1996) that allow us to shift definitions into their use sites in order to expose opportunities for further transformation. In the absence of side effects we can arbitrarily change the order of evaluation of a program without changing its meaning.

The fact that a program uses computational effects does not rule out performing similar optimisations, provided we know which subexpressions have the potential to interfere with others. An important part of this is to be able to reason about the aliasing properties of data.

For example, suppose we are compiling the following function:

$$
\begin{array}{l}
\Lambda r_1\ r_2.\ \Lambda(w_2 : \mathit{Mutable}\ r_2).\ \lambda(x : \mathit{Int}\ r_1).\ \lambda(y : \mathit{Int}\ r_2). \\
\textbf{letregion}\ r_3\ \textbf{with}\ w_3 = \mathit{MkConst}\ r_3\ \textbf{in} \\
\textbf{let}\quad (z : \mathit{Int}\ r_3) \quad = \mathit{add}\ r_1\ r_3\ r_3\ x\ (1\ r_3) \\
\qquad\ \_ \qquad\qquad\ = \mathit{updateInt}\ r_2\ r_3\ w_2\ y\ (0\ r_3) \\
\textbf{in}\quad \mathit{add}\ r_1\ r_3\ r_1\ x\ z
\end{array}
$$

Where *add* and *updateInt* have the following types:

$$
\begin{array}{lcl}
\mathit{add} & :: & \forall r_1\ r_2\ r_3.\ \mathit{Int}\ r_1 \rightarrow \mathit{Int}\ r_2\ \xrightarrow{\mathit{Read}\ r_1 \vee \mathit{Read}\ r_2}\ \mathit{Int}\ r_3 \\
\mathit{updateInt} & :: & \forall r_1 r_2.\ \mathit{Mutable}\ r_1 \Rightarrow\ \mathit{Int}\ r_1 \rightarrow\ \mathit{Int}\ r_2\ \xrightarrow{\mathit{Read}\ r_2 \vee \mathit{Write}\ r_1}\ ()
\end{array}
$$

The above function adds one to its first argument, updates its second, then adds its first argument to the previous result. Now, suppose that we wish to optimise this function by performing the local unboxing transform (Leroy, 1997). This should allow us to keep the intermediate values in registers, reducing the number of objects we need to store in the heap. We write the type of unboxed integers as $\mathit{Int}_\#$, unboxed integer literals like $23_\#$, and use the following primitives:

$$
\begin{array}{lcl}
\mathit{box} & :: & \forall r_1.\ \mathit{Int}_\# \rightarrow \mathit{Int}\ r_1 \\
\mathit{unbox} & :: & \forall r_1.\ \mathit{Int}\ r_1\ \xrightarrow{\mathit{Read}\ r_1}\ \mathit{Int}_\# \\
\mathit{add}_\# & :: & \mathit{Int}_\# \rightarrow \mathit{Int}_\# \rightarrow \mathit{Int}_\#
\end{array}
$$

By design, these primitives do not support the destructive update of unboxed integers. This effectively places the program in SSA form. It also means that $\mathit{Int}_\#$ does not need to be annotated with a region variable, as reading values of this type does not cause an effect. Using these primitives we can rewrite our original function as follows — taking the liberty of passing the unboxed constant $1_\#$ directly to *add*, converting to A-normal form, and annotating each binding with the effect it causes.

$$
\begin{array}{lll}
\Lambda r_1\ r_2.\ \Lambda(w_2 : \mathit{Mutable}\ r_2).\ \lambda(x : \mathit{Int}\ r_1).\ \lambda(y : \mathit{Int}\ r_2). & & \\
\textbf{letregion}\ r_3\ \textbf{with}\ w_3 = \mathit{MkConst}\ r_3\ \textbf{in} & & \\
\textbf{let} \quad (x_1 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x & \mathit{Read}\ r_1 \\
\quad\ (z_1 : \mathit{Int}_\#) & = \mathit{add}_\#\ x_1\ 1_\# & \bot \\
\quad\ (z : \mathit{Int}\ r_3) & = \mathit{box}\ r_3\ z_1 & \bot \\
\quad\ \_ & = \mathit{updateInt}\ r_2\ r_3\ w_2\ y\ (0\ r_3) & \mathit{Read}\ r_3 \vee \mathit{Write}\ r_2 \\
\quad\ (x_2 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x & \mathit{Read}\ r_1 \\
\quad\ (z_2 : \mathit{Int}_\#) & = \mathit{unbox}\ r_3\ z & \mathit{Read}\ r_3 \\
\quad\ (\mathit{res} : \mathit{Int}_\#) & = \mathit{add}_\#\ x_2\ z_2 & \bot \\
\textbf{in} \quad \mathit{box}\ r_1\ \mathit{res} & & \bot
\end{array}
$$

As the bindings for z_1, z, and *res* are manifestly pure, we can float them directly into their use sites. This is also possible for the z_2 binding, as $\mathit{Read}\ r_3$ is pure because r_3 is constant. Doing this exposes an opportunity to use the identity $\forall x\ r.\ \mathit{unbox}\ r\ (\mathit{box}\ r\ x) \equiv x$

$$
\begin{array}{lll}
\Lambda r_1\ r_2.\ \Lambda(w_2 : \mathit{Mutable}\ r_2).\ \lambda(x : \mathit{Int}\ r_1).\ \lambda(y : \mathit{Int}\ r_2). & & \\
\textbf{letregion}\ r_3\ \textbf{with}\ w_3 = \mathit{MkConst}\ r_3\ \textbf{in} & & \\
\textbf{let} \quad (x_1 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x & \mathit{Read}\ r_1 \\
\quad\ \_ & = \mathit{updateInt}\ r_2\ r_3\ w_2\ y\ (0\ r_3) & \mathit{Read}\ r_3 \vee \mathit{Write}\ r_2 \\
\quad\ (x_2 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x & \mathit{Read}\ r_1 \\
\textbf{in} \quad \mathit{box}\ r_1\ (\mathit{add}_\#\ x_2\ (\mathit{unbox}\ r_3\ (\mathit{box}\ r_3\ (\mathit{add}_\#\ x_1\ 1_\#)))) & & \bot
\end{array}
$$

Now, although there is still an obvious optimisation opportunity left in this code, we must be mindful of potentially interfering effects. The bindings for x_1 and x_2 appear to do the same thing, but note the intermediate update of y. If x and y were bound to the same heap object, meaning that they were aliases, then the second call to *unbox* would always return zero. On the other hand, if we could guarantee that x and y were *not* aliases, then we could simply set $x_2 = x_1$ and save the second unboxing.

We will consider two ways to achieve this: the first using the witnesses of constancy we have already discussed, and the second using witnesses of distinctness which we will cover in the next section. For the first way, we specialise the function for the case where r_1 is constant, by requiring a witness of of constancy:

$$
\begin{array}{lll}
\Lambda r_1\ r_2.\ \Lambda(w_1 : \mathit{Const}\ r_1).\ \Lambda(w_2 : \mathit{Mutable}\ r_2).\ \lambda(x : \mathit{Int}\ r_1).\ \lambda(y : \mathit{Int}\ r_2). & & \\
\textbf{letregion}\ r_3\ \textbf{with}\ w_3 = \mathit{MkConst}\ r_3\ \textbf{in} & & \\
\textbf{let} \quad (x_1 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x & \mathit{Read}\ r_1 \\
\quad\ \_ & = \mathit{updateInt}\ r_2\ r_3\ w_2\ y\ (0\ r_3) & \mathit{Read}\ r_3 \vee \mathit{Write}\ r_2 \\
\textbf{in} \quad \mathit{box}\ r_1\ (\mathit{add}_\#\ x_1\ (\mathit{unbox}\ r_3\ (\mathit{box}\ r_3\ (\mathit{add}_\#\ x_1\ 1_\#)))) & & \bot
\end{array}
$$

Note that by requiring r_1 to be constant, we have also ensured that x and y cannot be aliases. Or rather, we have ensured that the function cannot be called if x and y are aliases. As discussed in §3.2 our type system ensures that witness of constancy and mutability for the same region cannot be created. If the caller provides a witness of constancy for r_1, then if x and y are the same object then there is no way it can also provide a witness of mutability for r_2.

One potential disadvantage of this approach is that the witness variable w_1 does not occur in the body of the abstraction that binds it. Here, we are using the witness to create a context where a particular transformation is valid, instead of simply passing it to a primitive function like *update* or *suspend*. During further optimisation we must be mindful not to erase the binders for such witnesses, else the function may be called with arguments that do not meet the properties we assumed when specialising it.

6.1 Witnesses of Distinctness

In the example in the previous section we eliminated the second call to *unbox* by specialising for the case where the first argument was constant. This got the job done, but was a bigger hammer than necessary. The only property that we really needed was that the two arguments did not alias, that is, they were distinct. With this in mind, we define a new witness kind and its associated constructor that expresses this directly.

$$\begin{array}{lll} \mathit{Distinct} & :: \% \to \% \to \Diamond & \text{(witness kind constructor)} \\ \mathit{MkDistinct} & :: \Pi(r_1 : \%).\Pi(r_2 : \%).\ \mathit{Distinct}\ r_1\ r_2 & \text{(witness type constructor)} \end{array}$$

A witness of kind $\mathit{Distinct}\ r_1\ r_2$ guarantees that regions r_1 and r_2 are not aliases, meaning that objects in region r_1 do not alias objects in region r_2. We write the type level witness as $\underline{\text{distinct}\ \varphi_1\ \varphi_2}$, for some region types φ_1 and φ_2. Before discussing how these witnesses are introduced into the program, we will revisit the example from the previous section. This time, we specialise it for the case where the two arguments are distinct, without also requiring that the first is constant:

$$\begin{array}{lll} \multicolumn{3}{l}{\Lambda r_1\ r_2.\ \Lambda(w_1 : \mathit{Distinct}\ r_1\ r_2).\ \Lambda(w_2 : \mathit{Mutable}\ r_2).\ \lambda(x : \mathit{Int}\ r_1).\ \lambda(y : \mathit{Int}\ r_2).} \\ \multicolumn{3}{l}{\textbf{letregion}\ r_3\ \textbf{with}\ w_3 = \mathit{MkConst}\ r_3\ \textbf{in}} \\ \textbf{let} & (x_1 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x \qquad\qquad\qquad\qquad\qquad\quad \mathit{Read}\ r_1 \\ & \_ & = \mathit{updateInt}\ r_2\ r_3\ w_2\ y\ (0\ r_3) \qquad\quad \mathit{Read}\ r_3 \vee \mathit{Write}\ r_2 \\ \textbf{in} & \mathit{box}\ r_1\ (\mathit{add}_\#\ x_1\ (\mathit{unbox}\ r_3\ (\mathit{box}\ r_3\ (\mathit{add}_\#\ x_1\ 1_\#)))) & \bot \end{array}$$

As we have this new witness, the effects $\mathit{Read}\ r_1$ and $\mathit{Write}\ r_2$ are guaranteed not to interfere. Due to this, it is also safe to move the x_1 binding across the call to *updateInt* so that is closer to its use site. This will help to reduce the number of registers required when compiling to native machine code. We can also apply the *unbox*/*box* identity to yield our final version:

$$\begin{array}{lll} \multicolumn{3}{l}{\Lambda r_1\ r_2.\ \Lambda(w_1 : \mathit{Distinct}\ r_1\ r_2).\ \Lambda(w_2 : \mathit{Mutable}\ r_2).\ \lambda(x : \mathit{Int}\ r_1).\ \lambda(y : \mathit{Int}\ r_2).} \\ \multicolumn{3}{l}{\textbf{letregion}\ r_3\ \textbf{with}\ w_3 = \mathit{MkConst}\ r_3\ \textbf{in}} \\ \textbf{let} & \_ & = \mathit{updateInt}\ r_2\ r_3\ w_2\ y\ (0\ r_3) \qquad \mathit{Read}\ r_3 \vee \mathit{Write}\ r_2 \\ & (x_1 : \mathit{Int}_\#) & = \mathit{unbox}\ r_1\ x \qquad\qquad\qquad\quad \mathit{Read}\ r_1 \\ \textbf{in} & \mathit{box}\ r_1\ (\mathit{add}_\#\ x_1\ (\mathit{add}_\#\ x_1\ 1_\#)) & \bot \end{array}$$

With this code, although there is nothing preventing a caller from instantiating both r_1 and r_2 with the same type (say, $\underline{\rho_5}$) our system ensures that the resulting function cannot be called. We do this by ensuring that there is no way to produce a witness of kind $\mathit{Distinct}\ \underline{\rho_5}\ \underline{\rho_5}$. In fact, this part turns out to be surprisingly easy.

As with witnesses of constancy and mutability, we tie the creation of witnesses of distinctness to **letregion**. Recall from EvLetRegion in Fig. 7 that the reduction of an expression such as **letregion** r **with** $\{\overline{w_i = \delta_i}\}$ **in** t first allocates a fresh region label ρ, then substitutes the handle $\underline{\rho}$ for all occurrences of r in both the witness set $\overline{\delta_i}$ and the body t. The fact that $\underline{\rho}$ is fresh means that it cannot alias with any existing handles. Also, as region handles do not appear in binding positions, they behave like type constructors and are not substituted for one another. Clearly, we must prevent terms such as $MkDistinct\ r_1\ r_1$ from appearing in the initial program, but as with $MkConst$ and $MkMutable$, requiring all occurrences of $MkDistinct$ to appear in the witness set associated with a **letregion** reduces this to a simple syntactic check. For example, consider the following function:

$\Lambda(r_1 : \%)$. ...
letregion r_2 **in** ...
letregion r_3**with** $\{w_3 = MkDistinct\ r_1\ r_3;\ w_4 = MkDistinct\ r_2\ r_3\}$
in ... w_3 ... w_4 ...

Now, suppose we apply this function to the region handle $\underline{\rho_1}$, and then reduce the outer **letregion**. Doing this leaves us with the following state:

$$H, \rho_1, \rho_2\ ;\quad \textbf{letregion}\ r_3\ \textbf{with}\ \{w_3 = MkDistinct\ \underline{\rho_1}\ r_3;\ w_4 = MkDistinct\ \underline{\rho_2}\ r_3\}\ \textbf{in}\ \ldots\ w_3\ \ldots\ w_4\ \ldots$$

By EvLetRegion, the next step is to allocate a fresh region handle (say $\underline{\rho_3}$), substitute it into the witness set, reduce the witness expressions to normal form, and then substitute these for w_3 and w_4 in the body. The corresponding evaluation and kinding rules are as follows:

$$H,\ \rho_1,\ \rho_2\ ;\ MkDistinct\ \underline{\rho_1}\ \underline{\rho_2} \quad \rightsquigarrow\ \underline{\text{distinct}\ \rho_1\ \rho_2} \quad \rho_1 \neq \rho_2 \qquad \text{(EwDistinct)}$$

$$\Gamma \mid \Sigma,\ \rho_1,\ \rho_2 \ \vdash_{\kappa}\ \underline{\text{distinct}\ \rho_1\ \rho_2} \quad ::\ Distinct\ \underline{\rho_1}\ \underline{\rho_2} \quad \rho_1 \neq \rho_2 \qquad \text{(KiDistinct)}$$

Completing the reduction leaves us with:

$$H, \rho_1, \rho_2, \rho_3\ ;\ \ldots\ \underline{\text{distinct}\ \rho_1\ \rho_3}\ \ldots\ \underline{\text{distinct}\ \rho_2\ \rho_3}\ \ldots$$

Once these witnesses have been created, the fact that we cannot substitute for their embedded region handles means that they cannot be changed. This aspect of our system arises naturally from the phase distinction between region variables and region handles, as first discussed in §4.7. As the properties encoded by each witness are true at the time of creation, and as neither witnesses or the underlying properties of the store can be changed during evaluation, they must also be true at the time of use.

Also, by inspection of our kinding rules, the only closed (with no free variables), normal form type that can have a kind like $Distinct\ \underline{\rho_1}\ \underline{\rho_3}$ is $\underline{\text{distinct}\ \rho_1\ \rho_3}$. From this it follows that when we apply a function to type of kind $Distinct\ \underline{\rho_1}\ \underline{\rho_3}$ that type must be $\underline{\text{distinct}\ \rho_1\ \rho_3}$. Our Progress theorem shows that such a reduction is always possible. It then follows that when we pass such a witness, the corresponding property of the store is always true.

Note that our use of witnesses to guarantee non-aliasing is weaker than that provided by a "real" aliasing analysis such as (Guo *et al.*, 2005) or (Smith *et al.*, 1999). These systems can recover (an approximation to) aliasing relationships between objects at every point in the program, from an arbitrary sequence of update statements. In contrast, in our system

the occurrence of a witness of kind $Distinct\ r_1\ r_2$ ensures that objects in regions r_1 and r_2 never alias, ever. The advantage of our approach is that it's practically free. With respect to the concrete implementation all we had to add was the kind and superkind of *MkDistinct* and *Distinct*, and a trivial syntactic check on the form of **letregion**. Now we can implement aliasing aware optimisations without also writing a heavy-duty analysis or adding further type machinery.

7 Related Work

The inspiration for our work has been to build on the monadic intermediate languages of (Tolmach, 1998), (Benton & Kennedy, 1999) and (Peyton Jones *et al.*, 1998). 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 (Wadler & Thiemann, 2003) gives a translation between the two. Our system extends the previous with region, effect and mutability polymorphism, which improves the scope of the optimisations that can be performed. The work of (Benton *et al.*, 2007) and (Benton & Buchlovsky, 2007), presents monadic languages that include region and effect polymorphism, but does not consider mutability polymorphism or lazy evaluation. The roots of effect polymorphism lie in (Lucassen & Gifford, 1988) which also uses a System-F based language.

Our notion of a "witness" is derived from from a long line of work on typed intermediate languages (Tarditi *et al.*, 1996), typed assembly languages (Morrisett *et al.*, 1999) and *proof carrying code* (Necula, 1997). However, in the last two of these systems the focus is guaranteeing the safety of programs that use low level operations such as strong update, that is, destructive update that changes the *type* of a heap location as well as its value. In contrast, we do not support such an operation, and our focus is on justifying the transformation of specialised versions of a function.

The Capability Calculus (Crary *et al.*, 1999) 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 Vault language of (DeLine & Fähndrich, 2001) is related. Although the capabilities of (Crary *et al.*, 1999) have similarities to the witnesses of our system, theirs are not reified in the term being evaluated, and we do not allow ours to be revoked.

An important consideration of our work is to keep the programs digestible by humans. As our system is used as the core language of a larger compiler we spend a substantial amount of time staring at programs written in it. The fact that information about mutability, purity and aliasing is reified and explicit in the core program helps us, as compiler developers, to reason about the compilation process. In this respect our approach is closer to (Terauchi & Aiken, 2005). This system also uses type level witnesses that are explicitly threaded through the program. They then provide a mechanism to check for *witness race freedom*, meaning that a program using side effects has enough data dependencies to ensure that a parallel reduction of it has a deterministic result.

FABLE (Swamy *et al.*, 2008) is a dependently typed extension of System-F that associates type level security labels with the data or actions they protect. These labels can be

used to statically enforce security policies such as: search queries read from a network socket must be checked for malicious instructions before being passed onto a backend database. Although these labels could express the *Mutable* and *Const* constraints of our own system, they are treated more like data constructors than type class constraints. FABLE contains a syntactically separated policy language that is used to explicitly add, remove and test for the labels attached to some object. However, their use of dependent types means that type checking is undecidable without the use of an auxiliary termination checker, whereas our language is less general but straightforward to check.

With respect to mutability polymorphism the stated goal of the BitC (Shapiro *et al.*, 2008) language is to permit 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. Finally, (Gupta, 1995) presents an extension to the parallel language Id, which allows objects to be destructively initialised and then treated as constant from then on. As in our own system, Gupta uses region variables to track the mutability of objects. Instead of using region constraints, region variables are only attached to the types of mutable objects, leaving the types of constant objects annotated with the null region ε.

8 Conclusions and Future Work

We have presented a System-F style intermediate language that supports mutability polymorphism as well as lazy evaluation. We have used dependently kinded witnesses to track the mutability, constancy and aliasing of regions, and the purity of effects. 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 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.

Another current limitation is that we have no way to destructively initialise an object, and then treat it as constant from that point onward. For example, in the *length* function from §3.2 we copied the final counter value before returning it because we did not want a *Mutable* constraint in the overall type of *length*. We are currently working on a system to provide a local witness of mutability to an expression that performs such initialisation. We must then ensure that this witness, and hence the capability to perform further updates, cannot be exported out of the context that performs the initialisation.

Acknowledgements Many thanks to Clem Baker-Finch and Simon Winwood for discussions on witnesses and aliasing. Thanks also to Erik de Castro Lopo, Ben Sinclair and others for test cases and patches to DDC. This work was supported in part by the School of Computer Science, Australian National University, and funded in part by the Australian Research Council under grant number LP0989507.

References

Avron, Arnon, Honsell, Furio, & Mason, Ian A. (1989). An overview of the Edinburgh logical framework. *Pages 323–240 of: Current trends in hardware verification and automated theorem proving*. Springer-Verlag.

Benton, Nick, & Buchlovsky, Peter. (2007). Semantics of an effect analysis for exceptions. *Pages 15–26 of: Proc. of TLDI: Types in Language Design and Implementation*. ACM.

Benton, Nick, & Kennedy, Andrew. (1999). Monads, effects and transformations. *Pages 1–18 of: Electronic notes in theoretical computer science*. Elsevier.

Benton, Nick, Kennedy, Andrew, Beringer, Lennart, & Hofmann, Martin. (2007). Relational semantics for effect-based program transformations with dynamic allocation. *Pages 87–96 of: Proc. of PPDP: Principles and Practice of Declarative Programming*. ACM.

Crary, Karl, Walker, David, & Morrisett, Greg. (1999). Typed memory management in a calculus of capabilities. *Pages 262–275 of: Proc. of POPL: Principles of Programming Languages*. ACM.

de Medeiros Santos, André Luís. (1995). *Compilation by Transformation in Non-Strict Functional Languages*. Ph.D. thesis, University of Glasgow.

DeLine, Robert, & Fähndrich, Manuel. (2001). Enforcing high-level protocols in low-level software. *Pages 59–69 of: Proc. of PLDI: Programming Language Design and Implementation*. New York, NY, USA: ACM.

Foster, Jeffrey, Fähndrich, Manuel, & Aiken, Alexander. (1999). A theory of type qualifiers. *Sigplan notices*, **34**(5), 192–203.

Garrigue, Jacques. (2004). Relaxing the value restriction. *Pages 196–213 of: Proc. of FLOPS: International Symposium on Functional and Logic Programming*. Springer-Verlag.

Guo, Bolei, Bridges, Matthew J., Triantafyllis, Spyridon, Ottoni, Guilherme, Raman, Easwaran, & August, David I. (2005). Practical and accurate low-level pointer analysis. *Pages 291–302 of: Proc. of CGO: International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society.

Gupta, Shail Aditya. (1995). *Functional encapsulation and type reconstruction in a strongly-typed, polymorphic language*. Ph.D. thesis, Massachusetts Institute of Technology.

Launchbury, John. (1993). A natural semantics for lazy evaluation. *Pages 144–154 of: Proc. of POPL: Principles of Programming Languages*. ACM.

Launchbury, John, & Peyton Jones, Simon. (1994). Lazy functional state threads. *Pages 24–35 of: Proc. of PLDI: Programming Language Design and Implementation*. ACM.

Leroy, Xavier. (1993). Polymorphism by name for references and continuations. *Pages 220–231 of: Popl '93: Proceedings of the 20th acm sigplan-sigact symposium on principles of programming languages*. New York, NY, USA: ACM.

Leroy, Xavier. (1997). *The effectiveness of type-based unboxing*. Tech. rept. Boston College Computer Science Department.

Leroy, Xavier, Doligez, Damien, Garrigue, Jacques, Rémy, Didier, & Vouillon, Jérôme. (2008). *The Objective Caml system, release 3.11, documentation and user's manual.* Tech. rept. INRIA.

Lippmeier, Ben. (2010). *Type inference and optimisation for an impure world*. Ph.D. thesis, Australian National University.

Lucassen, J. M., & Gifford, D. K. (1988). Polymorphic effect systems. *Pages 47–57 of: Proc. of POPL: Principles of Programming Languages*. ACM.

MacQueen, David B. (1991). Standard ML of New Jersey. *Pages 1–13 of: Proc. of PLILP: Programming Language Implementation and Logic Programming*. Springer-Verlag.

Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From System-F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, **21**(3), 527–568.

Naumann, David A. (2007). Observational purity and encapsulation. *Theoretical computer science*, **376**(3), 205–224.

Necula, George C. (1997). Proof-carrying code. *Pages 106–119 of: Proc. of POPL: Principles of Programming Languages*. ACM.

Peyton Jones, Simon, & Lester, David. (1995). A modular fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, **21**, 479–506.

Peyton Jones, Simon, Partain, Will, & Santos, André. (1996). Let-floating: moving bindings to give faster programs. *Pages 1–12 of: Proc. of ICFP: International Conference on Functional Programming*. ACM.

Peyton Jones, Simon, Shields, Mark, Launchbury, John, & Tolmach, Andrew. (1998). Bridging the gulf: a common intermediate language for ML and Haskell. *Pages 49–61 of: Proc. of POPL: Principles of Programming Languages*. ACM.

Shapiro, Jonathan, Sridhar, Swaroop, & Doerrie, Scott. (2008). *BitC language specification*. Tech. rept. The EROS Group and Johns Hopkins University.

Smith, Frederick, Walker, David, & Morrisett, Greg. (1999). Alias types. *Pages 366–381 of: In European Symposium on Programming*. Springer-Verlag.

Sulzmann, Martin, Chakravarty, Manuel M. T., Peyton Jones, Simon, & Donnelly, Kevin. (2007). System-F with type equality coercions. *Proc. of TLDI: Types in Language Design and Implementation*. ACM.

Swamy, Nikhil, Corcoran, Brian J., & Hicks, Michael. (2008). Fable: A language for enforcing user-defined security policies. *Proc. of IEEE Symposium on Security and Privacy*. Society Press.

Talpin, Jean-Pierre, & Jouvelot, Pierre. (1992). The type and effect discipline. *Pages 162–173 of: Proc. of Logic in Computer Science*. IEEE.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., & Lee, P. (1996). TIL: a type-directed optimizing compiler for ML. *Pages 181–192 of: Proc. of PLDI: Programming Language Design and Implementation*. ACM.

Terauchi, Tachio, & Aiken, Alex. (2005). Witnessing side effects. *Proc. of ICFP: International Conference on Functional Programming*.

Tofte, Mads, Birkedal, Lars, Elsman, Martin, Hallenberg, Niels, Olesen, Tommy Højfeld, & Sestoft, Peter. 2006 (January). *Programming with regions in the MLKit (revised for version 4.3.0)*. Tech. rept. IT University of Copenhagen, Denmark.

Tolmach, Andrew. (1998). Optimizing ML using a hierarchy of monadic types. *Pages 97–113 of: Proc. of TIC: Types in Compilation*. Springer Verlag.

Wadler, Philip, & Thiemann, Peter. (2003). The marriage of effects and monads. *ACM Transactions on Computation and Logic*, **4**(1), 1–32.