Under consideration for publication in J. Functional Programming

Mechanized soundness for a type and effect system with region deallocation

BEN LIPPMEIER
School of Computer Science and Engineering
University of New South Wales, Australia
(*e-mail:* benl@cse.unsw.edu.au)

Abstract

Previous formalizations of region calculi with deallocation have been limited to a monomorphic or first-order fragment, or have not supported mutable references because they do not model the store. Extending the existing monomorphic approach with System-F style polymorphism quickly leads to trouble, because type substitution breaks a key freshness invariant on region identifiers. We present a new approach that uses frame stacks to manage region lifetimes, and to provide the required freshness invariants in a way compatible with type substitution. Our language is polymorphic, higher-order, supports mutable references, and is mechanized with the Coq proof assistant.

1 Introduction

A region is a set of store locations where an object may exist at runtime. Region and effect systems (Lucassen & Gifford, 1988) are used to reason about how computations that mutate objects in a shared heap may interfere. These systems encode information about *separation* and *aliasing*, and are related to similarly named separation logics (Reynolds, 2002) and alias type systems (Smith *et al.*, 2000). Some region typing systems also enable region based *memory management*, where a group of objects can be allocated into a single region of the store, and then deallocated in constant time when no longer needed by the program. Region-based memory management has been used as an alternative to tracing garbage collection, largely because the time to perform a deallocation cycle is small and constant, rather than being proportional to the amount of live data.

The original work on region typing (Lucassen, 1987) suggests that regions could be used for memory management, but does not provide an operational semantics, or proof of soundness for this feature. Later work by Tofte and Talpin (1993) provides a proof of soundness phrased in terms of translation correctness from a pure source language, though their system does not support destructive update as per the original. As noted by Helsen and Thiemann (2000), Tofte and Talpin's work proves soundness of their region calculus, and translation correctness from the pure source language in one go. This makes it non-obvious how to add destructive update back to the underlying region calculus, as it is not supported by the pure source language. This observation lead Helsen and Theimann to develop a simpler proof which factors soundness and translation correctness into separate statements, though their evaluation semantics is simplified and does not include a mutable store. In

later work Calcagno *et al.* (2002) give a syntactic soundness proof whose semantics *does* include a mutable store, but the presentation lacks the polymorphism of the original region calculus.

It turns out that adding System-F style polymorphism to the language of (Calcagno *et al.*, 2002) in the obvious way would break a critical freshness invariant on region identifiers, so to prove soundness for such a system we must at least refactor the existing semantics. We describe the exact problem in §1.6. The related language of (Henglein *et al.*, 2004) instead depends on implicit alpha conversion to maintain freshness, but also does not include a store. With this in mind, we make the following contributions:

- We present a mechanized semantics and soundness proof for an explicitly typed, polymorphic functional language based on call-by-value System-F with 1) a region and effect system with separate *Read*, *Write* and *Alloc* effects; 2) stack based regions in the style of Tofte and Talpin; 3) destructive update of mutable references. We refer to this language as System-F^{re} (System-F with regions and effects).
- Our semantics refactors the existing systems, using an explicit stack to remember which regions are currently live, and gives liveness conditions between the stack and store. Our use of an explicit stack avoids breaking the required freshness invariants when polymorphism is added, and is the key difference to prior work.
- Our language also supports *region extension*, which was described in the original work of (Lucassen, 1987) but does not appear in any of the latter formalizations. Region extension allows a function to destructively initialize mutable objects without the associated write effects being visible to the calling context.

The Coq proof script is available as an on-line supplement to this paper. The overall aim is to define a verified compiler intermediate representation which is amenable to the same sort of optimising program transformations as the Glasgow Haskell Compiler (GHC) core language (Peyton Jones & Santos, 1998), but with direct support for region based memory management and computational effects. The examples we present are explicitly typed, though we intend to define an implicitly typed surface language in future work.

1.1 Regions and Effects

Before presenting the formal semantics we give a quick overview of how System-F^{re} supports regions. The following example summarizes their basic use:

```
let tally : Nat → Nat
    = λ start : Nat.
      private r in
      let acc : Ref r Nat    = alloc r start in
      let eat : Nat -(Read r + Write r)-> Unit
          = (λz : Nat. let current : Nat = read r acc
                       in write r acc (current + z)) in
      ... eat 1 ... eat 5 ... eat 2 ...
      in read r acc
in tally 0
```

The *tally* function takes a natural number *start*, and then creates a `private` region named *r*, where *r* is a type variable which is in scope in the body of the `private` construct.

The *tally* function then allocates a new accumulator *acc*, represented by a mutable reference in region *r*. This accumulator has type *Ref r Nat*, revealing the region it lies in. The inner function *eat* reads the current value of the accumulator, then writes a new value consisting of the current value added to the parameter *z*. We have written ...*eat* 1... and so on a place-holder for some representative applications of *eat*. The last line of *tally* reads the final value of the accumulator and returns it to the caller.

At runtime, a new store region will be allocated when entering the body of the `private` construct, and then deallocated when leaving it (just before returning from *tally*). In the body of the `private` construct we refer to the new region with the region variable *r*. The inner function *eat* is assigned the type $Nat \xrightarrow{Read\ r + Write\ r} Unit$, which includes effect $(Read\ r + Write\ r)$ indicating that *eat* reads and writes to an object in region *r*. Atomic effects terms like *Read r* are collected in an upwards semi-lattice ordered by set inclusion, where we write $+$ for l.u.b. An unannotated arrow has effect $\bot$ (pronounced "pure").

Importantly, the type of *tally* itself does not include an effect on region *r*, because reads and writes to objects in the private region *r* are not visible to the calling context. This *masking* of non-observable effects comes from the original work on effect systems (Lucassen, 1987), and provides a powerful abstraction mechanism: allowing us to treat functions that use local side effects as observationally pure. Monadic state threads (Launchbury & Peyton Jones, 1994) provide a similar encapsulation mechanism, though without distinguishing between separate read and write effects.

1.2 Region Deallocation

When leaving the scope of a (`private` *r* `in` *body*) construct, all objects in region *r* are deallocated. To ensure that further evaluation does not attempt to access the deallocated objects, the type system requires that the bound region variable *r* is not free in the type of *body*. For example, consider the following erroneous function:

$$
\begin{array}{l}
broken : Nat \rightarrow Nat \xrightarrow{Read\ r} Nat \\
\quad = \ \lambda y : Nat.\ \texttt{private}\ r\ \texttt{in} \\
\qquad \texttt{let}\ ref : Ref\ r\ Nat = \texttt{alloc}\ r\ y \\
\qquad \texttt{in}\ \ (\lambda z : Nat.\ (\texttt{read}\ r\ ref) + z)
\end{array}
$$

When *broken* is applied to its first argument it creates a private region *r* and then allocates a new mutable reference *ref* into this region. The result of applying *broken* to this first argument is a functional value that takes a second argument (for *z*) and reads the original reference *ref*. This behavior is invalid, because the storage for *ref* will be deallocated when leaving the scope of the original `private` construct. If the caller of *broken* tries to apply the inner function then the call to `read` will fail.

In System-F^{re} and related systems, effect typing is used to expose which store objects a functional value may access when applied. In the above example, the inner function has type $Nat \xrightarrow{Read\ r} Nat$, which violates the requirement that *r* not be free in the type of the body of the outer `private` construct. Indeed, the overall type for *broken* was already suspect, because the private region variable *r* is not in scope.

1.3 Region and Effect Polymorphism

Region and effect polymorphism are necessary tools for a practical language. For example, suppose we want a function that reads two references to *Nat* values and returns their sum in another reference. In System-F^{re} we would write this as follows, using Λ for type abstraction, and overloading $(+)$ as the addition operator on plain values of type *Nat*.

$$
\begin{aligned}
&\mathit{add} : \forall r_1\ r_2\ r_3 : \mathit{Region}.\ \mathit{Ref}\ r_1\ \mathit{Nat} \rightarrow \mathit{Ref}\ r_2\ \mathit{Nat} \xrightarrow{\mathit{Read}\ r_1 + \mathit{Read}\ r_2 + \mathit{Alloc}\ r_3} \mathit{Ref}\ r_3\ \mathit{Nat} \\
&\quad = \Lambda r_1\ r_2\ r_3 : \mathit{Region}.\ \lambda (y : \mathit{Ref}\ r_1\ \mathit{Nat}).\ \lambda (z : \mathit{Ref}\ r_2\ \mathit{Nat}). \\
&\qquad \mathtt{let}\ y' : \mathit{Nat}\ = \mathtt{read}\ r_1\ y\ \mathtt{in} \\
&\qquad \mathtt{let}\ z' : \mathit{Nat}\ = \mathtt{read}\ r_2\ z\ \mathtt{in} \\
&\qquad \mathtt{alloc}\ r_3\ (y' + z')
\end{aligned}
$$

If System-F^{re} was used as as the core language of a compiler, then we could view *Nat* as the type of primitive unboxed natural numbers, and type $\mathit{Ref}\ r_1\ \mathit{Nat}$ as the type of boxed natural numbers. The *add* function above then performs boxed addition, which must be region polymorphic so that we can add boxed naturals from any region. In the type of *add* we use *Region* as the kind of region variables. Region variables are type variables with this special kind. As an example of effect polymorphism, the following function takes a functional parameter and applies it to the provided argument twice:

$$
\begin{aligned}
&\mathit{twice} : \forall a\ : \mathit{Data}.\ \forall e : \mathit{Effect}.\ (a \xrightarrow{e} a) \rightarrow a \xrightarrow{e} a \\
&\qquad = \Lambda a : \mathit{Data}.\ \ \Lambda e : \mathit{Effect}.\ \lambda f : a \xrightarrow{e} a.\ \lambda z : a.\ f\ (f\ z)
\end{aligned}
$$

In the type of *twice*, we use *Data* as the kind of data types, which is similar to the kind $*$ (star) used in Haskell. All representable values have types with kind *Data*. We use *Effect* as the kind of effect types. Note that in the signature for *twice*, the fact that this function applies its functional argument is revealed in by the *e* annotation on the right-most function arrow.

1.4 Region Extension

Region extension allows store objects to be destructively initialized without revealing the associated write effects to the calling context. For example, the following function ties a recursive knot through the store, producing a reference to a function that always diverges when applied.

$$
\begin{aligned}
&\mathit{tie} : \forall r_1 : \mathit{Region}.\ \mathit{Unit} \xrightarrow{\mathit{Alloc}\ r_1} \mathit{Ref}\ r_1\ (\mathit{Nat} \xrightarrow{\mathit{Read}\ r_1} \mathit{Nat}) \\
&\quad = \Lambda r_1 : \mathit{Region}.\ \lambda \_ : \mathit{Unit}. \\
&\qquad \mathtt{extend}\ r_1\ \mathtt{with}\ r_2\ \mathtt{in} \\
&\qquad \mathtt{let}\ \mathit{zero}\ :\ \mathit{Ref}\ r_2\ \mathit{Nat} \qquad\qquad = \mathtt{alloc}\ r_2\ 0\ \mathtt{in} \\
&\qquad \mathtt{let}\ \mathit{foo}\ \ :\ \mathit{Nat} \xrightarrow{\mathit{Read}\ r_2} \mathit{Nat} \qquad = (\lambda \_ : \mathit{Nat}.\ \mathtt{read}\ r_2\ \mathit{zero})\ \mathtt{in} \\
&\qquad \mathtt{let}\ \mathit{ref}\ \ :\ \mathit{Ref}\ r_2\ (\mathit{Nat} \xrightarrow{\mathit{Read}\ r_2} \mathit{Nat}) = \mathtt{alloc}\ r_2\ \mathit{foo}\ \mathtt{in} \\
&\qquad \mathtt{let}\ \mathit{loop}\ :\ \mathit{Nat} \xrightarrow{\mathit{Read}\ r_2} \mathit{Nat} \qquad = (\lambda x : \mathit{Nat}.\ \mathtt{let}\ f : \mathit{Nat} \xrightarrow{\mathit{Read}\ r_2} \mathit{Nat} = \mathtt{read}\ r_2\ \mathit{ref} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{in}\ \ f\ x)\ \mathtt{in} \\
&\qquad \mathtt{let}\ \_\ \ \ :\ \mathit{Unit} \qquad\qquad\qquad\qquad = \mathtt{write}\ r_2\ \mathit{ref}\ \mathit{loop} \\
&\qquad \mathtt{in}\ \ \mathit{ref}
\end{aligned}
$$

The construct (`extend` r_1 `with` r_2 `in` *body*) creates a new region r_2 which is private to *body*, evaluates *body*, and then merges all objects that were allocated in r_2 into r_1. The introduced variable r_2 is only in scope in *body*. In a concrete implementation, the merging process would be carried out in constant time, by modifying runtime region descriptors or other store meta-data, and not by copying the objects themselves.

Using a separate region variable r_2 allows us to assign an effect to the `extend` expression that properly reflects the overall modification to the store. In particular, although the body of our example performs reads and writes on region r_2, because region r_2 is local to the enclosing function, these effects are not visible to the calling context and can be masked. As far as the calling context is concerned, once *tie* has returned, all that has happened is that a new *ref* object has been allocated into region r_1. The overall effect of the `extend` expression is thus *Alloc* r_1, which is also the effect of the enclosing function.

1.5 Dangling references

As with the languages described by Lucassen (1987) and Tofte & Talpin (1993), System-F^{re} allows the expression under evaluation to refer to objects in regions that have been deallocated, provided those objects are never accessed. Consider the following example:

$$
\begin{array}{l}
\mathit{dangle} : \ \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\quad = \lambda y : \mathit{Nat}.\ \texttt{private}\ r_1\ \texttt{in} \\
\qquad \texttt{let}\ \mathit{goodbye} : \mathit{Ref}\ r_1\ \mathit{Nat} = \texttt{alloc}\ r_1\ y \\
\qquad \texttt{in}\ \ (\lambda z : \mathit{Nat}.\ \texttt{private}\ r_2\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \mathit{dref} : \mathit{Ref}\ r_2\ (\mathit{Ref}\ r_1\ \mathit{Nat}) = \texttt{alloc}\ r_2\ \mathit{goodbye}\ \ \texttt{in}\ z)
\end{array}
$$

When *dangle* is applied to its first argument, it creates a private region named r_1 and allocates a reference *goodbye* into this region. It then returns an inner function that allocates a new reference *dref* into a separate private region each time it is called. However, when the inner function is returned outside the scope of the enclosing `private` r_1 construct, region r_1 will be deallocated, and *goodbye* along with it. This means that every time the inner function is subsequently called, it will allocate a new *dref* object that refers to the missing *goodbye*. We say that the inner function holds a *dangling reference* to *goodbye*.

Importantly, the *dangle* function above is accepted as valid because the original *goodbye* object is never accessed after it has been deallocated. In terms of a concrete implementation, it is acceptable for the inner function to hold a pointer to *goodbye* after the object itself has been deallocated, provided this pointer is never dereferenced (followed). This restriction is enforced by the type system. If the inner function erroneously accessed the *goodbye* object then it would have a *Read r* or *Write r* effect. This would violate the restriction that the type of the body of the `private` r_1 construct not mention r_1, as per the *broken* example from §1.2.

1.6 The problem with polymorphism and private regions

The operational semantics for a language with region based memory management typically introduces a phase distinction that separates source level region *variables* from run-time region *identifiers*. Region variables are written by the user in their program, while region

identifiers identify chunks of memory at runtime, and are allocated by the system during evaluation. In our notation we write source level region variables as r and runtime region identifiers as p. Using the phase distinction, a candidate evaluation rule for `private` expressions would be something like the following, though we will improve on this in a moment.

$$ss \mid \texttt{private}\ r\ \texttt{in}\ x \longrightarrow ss \mid x[(\texttt{rgn}\ p)/r],\quad p \text{ fresh} \quad \text{(PrivPlain)}$$

This rule takes a *state* consisting of a store ss, and an expression (`private` r `in` x) , and produces a new store and expression — using x as a meta-variable for whole e(x)pressions. In the above rule we allocate a fresh region identifier p, wrap it into a type by writing `rgn` p, then substitute this type into the body of the `private` expression. We refer to a type like `rgn` p as a *region handle* to distinguish it from the plain region identifier p. In terms of a concrete implementation, we imagine p as being the numeric index of a region relative to the number of regions allocated so far, and `rgn` p as being a pointer to some runtime data structure that specifies where it lies in memory.

Although rule (PrivPlain) describes allocation well enough, it says nothing about *deallocation*. In the language described by Lucassen (1987), regions are allocated and deallocated in a stack-like order, following the nesting structure of `private` constructs in the source program, but we have not retained enough information to do this.

The language of Lucassen (1987) tracks the order in which regions are allocated by wrapping the result in an *auxiliary expression* that holds the new region identifier:

$$ss \mid sp \mid \texttt{private}\ r\ \texttt{in}\ x \longrightarrow ss \mid p,\ sp \mid \texttt{*private*}\ p\ \texttt{in}\ x[(\texttt{rgn}\ p)/r],\quad p \notin sp \quad \text{(PrivAux)}$$

This rule takes a state consisting of a store ss, store properties sp and expression, and produces a new store, store properties and expression. The store properties is a set which records the region identifiers that have been used so far, and the side condition $p \notin sp$ ensures the new identifier p is fresh relative to this set. The auxiliary `*private*` expression creates a context for its body to evaluate in. The intention is for the body expression to reduce to a value v, and then for store objects in region p to be deallocated using a rule like the following:

$$ss \mid sp \mid \texttt{*private*}\ p\ \texttt{in}\ v \longrightarrow \text{deallocRegion}\ p\ ss \mid sp \mid v \quad \text{(PrivDealloc)}$$

Now, although the (PrivAux) and (PrivDealloc) rules above provide a moral understanding of how allocation and deallocation should work, to complete a formal proof of soundness we must deal with subtle issues of name collision and capture. The following example demonstrates name collision:

$$\texttt{*private*}\ p\ \texttt{in}\ (\texttt{let}\ x = (\texttt{*private*}\ p\ \texttt{in}\ 5)\ \texttt{in}\ < \ldots \text{read from object in region } p \ldots >$$

Reduction of the above expression will fail because evaluation of the `let` binding will deallocate all objects that might be read in the `let` body. To avoid this problem we must ensure that every region identifier mentioned by a `*private*` expression is distinct.

The following example demonstrates a related issue, which looks like the familiar variable capture problem from pure lambda calculus. Suppose we have the following type application:

$$(\Lambda r : \textit{Region}.\ \texttt{*private*}\ p\ \texttt{in}\ f\ r\ z)\ (\texttt{rgn}\ p)$$

Reducing this application with the usual System-F style evaluation rule requires we perform the substitution $(\texttt{*private*}\ p\ \texttt{in}\ f\ r\ z)[(\texttt{rgn}\ p)/r]$. Note that in Lucassen's language (1987) this is not quite a "variable capture" problem, because region identifiers like p are treated as constructors (or atoms) rather than variables. The `*private*` expression *holds* the region identifier p but does not bind it. In the larger semantics, such an identifier may appear in the description of the run-time store, so it cannot be treated as a variable name local to a single `*private*` expression. The informal commentary in (Lucassen, 1987) and the later work (Lucassen & Gifford, 1988) makes it clear that above substitution is invalid, though it does not give a complete soundness proof for their language.

Calcagno *et al.* (2002) give a complete syntactic soundness proof for a monomorphic version of the region calculus (where they write `region` for `*private*`). In this work they give the following typing rule:

$$\frac{te \mid se \vdash x\ ::\ t\ ;\ e \qquad p \notin \mathrm{fr}(te,\ e)}{te \mid se \vdash \texttt{*private*}\ p\ \texttt{in}\ x\ ::\ t\ ;\ \mathrm{mask}\ p\ e}\ (\mathrm{TyPrivate})$$

The judgment $te \mid se \vdash x\ ::\ t\ ;\ e$ reads: "under type environment te and store environment se, expression x has type t and effect e". The type environment maps value variables to their types, and the store environment maps store locations to their types. In the consequent, the meta-expression $(\mathrm{mask}\ r\ e)$ masks out (erases) the atomic effects in e that mention region identifier p. The premise $p \notin \mathrm{fr}(te,\ e)$ ensures that p is not one of the free region identifiers mentioned in the type environment or effect of the body expression x. Now, although the above rule works for a monomorphic language, to add System-F style polymorphism and produce a syntactic soundness proof we must deal with the capture problem. During the proof, to show Preservation for type applications we require a type substitution lemma like the following, where *ke* is the kind environment that maps type variables to their kinds.

$$\begin{array}{ll}
\text{If} & ke,\ a : k_2 \mid te \mid se\ \vdash\ x_1\ ::\ t_1\ ;\ e_1 \\
\text{and} & ke\ \vdash\ t_2 :: k_2 \\
\text{then} & ke \mid te[t_2/a] \mid se\ \vdash x_1[t_2/a]\ ::\ t_1[t_2/a]\ ;\ e_2[t_2/a]
\end{array}$$

Sadly, this lemma is not true for the obvious extension of (TyPrivate), where we simply add a kind environment to the two typing judgments. A counter-example is just the invalid substitution $(\texttt{*private*}\ p\ \texttt{in}\ f\ r\ z)[(\texttt{rgn}\ p)/r]$ we mentioned before, with appropriate types for f and z.

$$\begin{array}{ll}
\text{If} & r : \mathit{Region} \mid f : \forall(r' : \mathit{Region}).\ \mathit{Ref}\ r'\ \mathit{Nat} \to \mathit{Unit},\ \ z : \mathit{Ref}\ r\ \mathit{Nat} \mid \cdot \\
 & \qquad \vdash\ \texttt{*private*}\ p\ \texttt{in}\ f\ r\ z\ ::\ \mathit{Unit}\ ;\ \bot \\
\text{and} & \cdot\ \vdash\ \texttt{rgn}\ p :: \mathit{Region} \\
\text{then} & \cdot \mid f : \forall(r' : \mathit{Region}).\ \mathit{Ref}\ r'\ \mathit{Nat} \to \mathit{Unit},\ \ z : \mathit{Ref}\ (\texttt{rgn}\ p)\ \mathit{Nat} \mid \cdot \\
 & \qquad \vdash\ \texttt{*private*}\ p\ \texttt{in}\ f\ (\texttt{rgn}\ p)\ z\ \ ::\ \mathit{Unit}\ ;\ \bot
\end{array}$$

The consequent in the above statement does not type check under the extended (TyPrivate) rule because the region identifier p appears in the type environment.

How do we fix this? If a `*private*` expression actually did bind the region identifier p then we could perhaps perform an alpha-conversion to avoid capture. However, as we mentioned earlier, `*private*` is not a binding construct, and the region identifier p may

also appear in the runtime description of the store. Instead, we could perhaps add a global renaming process that rewrote the machine state to avoid conflicting region identifiers — but this is a kludge. As we describe next, the above substitution, and hypothetical renaming process, would never be performed during the evaluation of a well-typed program.

Assume the initial state of the machine consists of a closed source expression in an empty store. The source expression will not contain any `*private*` expressions because no regions have yet been created. Now, the only evaluation rule that introduces `*private*` expressions would be one like (PrivAux) which we repeat here:

$$ss \mid sp \mid \texttt{private}\ r\ \texttt{in}\ x \longrightarrow ss \mid p,\ sp \mid \texttt{*private*}\ p\ \texttt{in}\ x[(\texttt{rgn}\ p)/r],\ \ p \notin sp \quad \text{(PrivAux)}$$

When a `*private*` expression is introduced into the program the fresh region identifier p is allocated and then the region handle containing it is substituted into the body. As the *context* of the `private` expression thus has no knowledge of p, it cannot also contain the type argument $(\texttt{rgn}\ p)$ that gave rise to the offending substitution. The counter-example above is really a problem with the formalization of the semantics, rather than a problem with the underlying language. However, such an informal apology about contexts does not count as a formal proof.

As discussed in the rest of this paper, to avoid capture we give up on auxiliary `*private*` expressions. We instead extend the machine state with a frame stack, which makes the context of the expression under evaluation explicit. We also use the stack to manage the lifetime of regions. The context information previously expressed by `*private*` is thus held in a different part of the machine state, and there is never a need to to substitute types into it. Instead of the freshness criteria that appeared in (TyPrivate) we use liveness conditions between the stack and store, as we will see in the next section.

2 Language

The grammar for System-F^{re} is shown in Fig. 1. We use a hybrid presentation with both named variables and de Bruijn indices. The Coq formalization uses de Bruijn indices natively, but as an aid to the reader we also include suggestive variable names when describing the language and stating the main theorems. We write concrete de Bruijn indices as underlined natural numbers, like $\underline{0}$, $\underline{1}$, $\underline{2}$ and so on.

Kinds

A kind can be a constructor *Data*, *Region* or *Effect*. These are the primitive kinds of data types, region types and effect types respectively. We use $(\rightsquigarrow)$ as the function kind constructor, using a different arrow symbol to distinguish it from the type constructor $(\rightarrow)$.

Types

Type variables are written $ix\{a\}$, where ix is the de Bruijn index and $\{a\}$ is a suggestive variable name. In all cases, if the reader prefers the concrete de Bruijn representation for binders, then they can simply erase the names between $\{\}$ braces.

We join effect types with $type + type$, and the kinding rules in Fig. 2 constrain both type arguments to have the *Effect* kind. The effect of a pure computation is written $\bot$.

The type $tycon_n$ is a primitive type constructor of arity n. To simplify the presentation, all type constructors except $(\rightarrow)$ must be fully applied. In a larger language, partial application of the other constructors could be encoded using type synonyms.

Indices

ix	$\rightarrow$	(debruijn index)
p	$\rightarrow$	(region identifier)
l	$\rightarrow$	(store location)

Names

a, r, e	$\rightarrow$	(type variables)
z	$\rightarrow$	(term variables)

Kinds (k)

$kind$	::=	$Data \mid Region \mid Effect$	(kind constructors)
	$\mid$	$kind \rightsquigarrow kind$	(function kind)

Types (t, r, e)

$type$	::=	$ix\{a\}$	(debruijn index{variable})
	$\mid$	$\forall\{a\} : kind.\ type \mid type\ \ type$	(universal quantifier, type application)
	$\mid$	$type + type \mid \bot$	(effect join and bottom)
	$\mid$	$tycon_n$	(type constructor)
	$\mid$	$\texttt{rgn}\ p$	(region handle)

Type Constructors (tc)

$tycon_0$	::=	$(\rightarrow) \mid Unit \mid Bool \mid Nat$
$tycon_1$	::=	$Read \mid Write \mid Alloc$
$tycon_2$	::=	Ref

Values (v)

val	::=	$ix\{z\} \mid const$	(debruijn index{variable}, constant)
	$\mid$	$\lambda\{z\} : type.\ exp \mid \Lambda\{a\} : kind.\ exp$	(value and type abstraction)
	$\mid$	$\texttt{loc}\ l$	(store location)

Expressions (x)

exp	::=	$val \mid val\ \ val \mid val\ \ type$	(value, value and type application)
	$\mid$	$\texttt{let}\ \{z\} : type\ =\ exp\ \texttt{in}\ exp$	(let-binding)
	$\mid$	$op_n\ \overline{val}^{\,n}$	(fully applied pure operator)
	$\mid$	$\texttt{private}\ \{r\}\ \texttt{in}\ exp$	(define a private region)
	$\mid$	$\texttt{extend}\ type\ \texttt{with}\ \{r\}\ \texttt{in}\ exp$	(extend an existing region)
	$\mid$	$\texttt{alloc}\ \ type\ \ val$	(allocate a store binding)
	$\mid$	$\texttt{read}\ \ type\ \ val$	(read a store binding)
	$\mid$	$\texttt{write}\ \ type\ \ val\ \ val$	(write a store binding)

Constants (c)

$const$	::=	$\texttt{unit} \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots$	(data constants)

Pure Operators (o)

op_1	::=	$\texttt{isZero} \mid \texttt{succ}$	(pure operators)

Fig. 1. System-F^{re} grammar.

The type `rgn` p is a region handle which contains a natural number p that identifies a runtime region. A region handle is a *static capability*, whose existence in a well typed expression indicates that region p currently exists; can have new store bindings allocated into it, and can be read from and written to. Region handles are similar to the capabilities of (Walker *et al.*, 2000), except that they appear directly in the type language rather than being present in a separate environment of the typing judgment. Region handles can also be captured in function closures and appear in functional values held in the store.

Values and Expressions

Values are the expressions that cannot be reduced further. Expression variables are written $ix\{z\}$, where ix is again a de Bruijn index and $\{z\}$ a suggestive variable name.

The value `loc` l is a store location that contains a natural number l giving the address of a mutable reference. All store locations have type *Ref* t_1 t_2, where t_1 is the region that location is in and t_2 is the type of the value at the location.

Function applications and primitive operators work on values rather than reducible expressions. As we will see in §4, this restriction ensures that the dynamic semantics only needs to deal with two reduction contexts, namely the right of a `let` binding, and a context that includes a new region variable. The expression `let` $\{z\} : t_1 = x_1$ `in` x_2 first reduces x_1 to a value before substituting it for z in x_2 (or for index $\underline{0}$ using the de Bruijn notation).

The expression $op_n\ \overline{val}^n$ is a fully applied, pure primitive operator, where n is the arity of the operator. The pure operators do not affect the store.

The expression `private` $\{r\}$ `in` x creates a new region, then substitutes the corresponding region handle for variable r in body x. It then reduces this new body to a value v and deallocates the region. The result of the overall expression is v.

The expression `extend` t_1 `with` $\{r_2\}$ `in` x takes the handle t_1 of an existing region, and creates a new region for r_2, which is available in body x. It then reduces x to a value v. Once the reduction of x has completed, all store bindings in the new region r_2 are merged into the original region with handle t_1. The result of the overall expression is v.

The expression `alloc` t v takes a region handle t, a value v, and allocates a new store binding in that region containing the given value. The expression `read` t v takes a region handle t, a store location v in that region, and reads the value at the location. Finally, `write` t $v_1 v_2$ takes a region handle t, a location v_1 in that region, a new value v_2, and overwrites the value in the store binding at location v_1 with the new value v_2.

3 Static Semantics

Fig. 3 shows the environments we use in both the kinding rules of Fig. 2 and typing rules of Fig. 4. In the Coq formalization these are de Bruijn environments, indexed by number, starting from the *right*. Following our hybrid presentation we also include suggestive variable names. We index environments from the right so that they appear as stacks that grow to the left. For example, consider the following kind environment:

$$\{a_1\} : \mathit{Data},\ \ \{e_1\} : \mathit{Effect},\ \ \{r_1\} : \mathit{Region}$$

Here, index $\underline{0}$ refers to the element $\{r_1\} : \mathit{Region}$ and index $\underline{1}$ to $\{e_1\} : \mathit{Effect}$. For this reason we write the corresponding type variables as $\underline{0}\{r_1\}$ and $\underline{1}\{e_1\}$ respectively. In typing judgments we write an empty environment using a single $\cdot$ dot.

In Fig. 3, the kind environment *kienv* and type environment *tyenv* consist of a list of kinds and types as usual. The store environment *stenv* gives the type of each location in a store, and the store properties *stprops* records the identifiers of regions that have been created so far. When a region is deallocated all the bindings in that region of the store are marked as dead. However, we retain the corresponding entry in the list of store properties so that any dangling references to those bindings are still well typed.

$$\boxed{kienv \mid stprops \vdash_t type :: kind} \quad \text{(KindT)}$$

$$\frac{k = \text{get}\ ix\{a\}\ ke}{ke \mid sp \vdash_t ix\{a\} :: k}\ \text{(KiVar)} \qquad \frac{\texttt{region}\ p \in sp}{ke \mid sp \vdash_t \texttt{rgn}\ p :: Region}\ \text{(KiRgn)}$$

$$\frac{ke, \{a\} : k \mid sp \vdash_t t :: Data}{ke \mid sp \vdash_t \forall\{a\} : k.\ t :: Data}\ \text{(KiForall)} \qquad \frac{}{ke \mid sp \vdash_t \bot :: Effect}\ \text{(KiBot)}$$

$$\frac{\begin{array}{c}k_{12} \neq Region \\ ke \mid sp \vdash_t t_1 :: k_{11} \rightsquigarrow k_{12} \qquad ke \mid sp \vdash_t t_2 :: k_{11}\end{array}}{ke \mid sp \vdash_t t_1\ t_2 :: k_{12}}\ \text{(KiApp)}$$

$$\frac{ke \mid sp \vdash_t t_1 :: Effect \qquad ke \mid sp \vdash_t t_2 :: Effect}{ke \mid sp \vdash_t t_1 + t_2 :: Effect}\ \text{(KiSum)}$$

$$\frac{k = \text{kindOfTyCon0}\ tc}{ke \mid sp \vdash_t tc :: k}\ \text{(KiCon0)} \qquad \frac{\begin{array}{c}ke \mid sp \vdash_t t_1 :: k_1 \\ k_1 \rightsquigarrow k = \text{kindOfTyCon1}\ tc\end{array}}{ke \mid sp \vdash_t tc\ t_1 :: k}\ \text{(KiCon1)}$$

$$\frac{\begin{array}{c}ke \mid sp \vdash_t t_1 :: k_1 \\ ke \mid sp \vdash_t t_2 :: k_2 \\ k_1 \rightsquigarrow (k_2 \rightsquigarrow k) = \text{kindOfTyCon2}\ tc\end{array}}{ke \mid sp \vdash_t tc\ t_1\ t_2 :: k}\ \text{(KiCon2)}$$

$$\begin{array}{lll}
\text{kindOfTyCon0} & (\rightarrow) & = Data \rightsquigarrow Effect \rightsquigarrow Data \rightsquigarrow Data \\
& Unit & = Data \\
& Bool & = Data \\
& Nat & = Data \\
\text{kindOfTyCon1} & Read & = Region \rightsquigarrow Effect \\
& Write & = Region \rightsquigarrow Effect \\
& Alloc & = Region \rightsquigarrow Effect \\
\text{kindOfTyCon2} & Ref & = Region \rightsquigarrow Data \rightsquigarrow Data
\end{array}$$

Fig. 2. Kinds of Types

kienv	::=	$\overline{\{a\} : kind}$	(kind environment)	*tyenv*	::=	$\overline{\{z\} : type}$	(type environment)
stenv	::=	$\overline{\{l\} : type}$	(store environment)	*stprops*	::=	$\overline{\texttt{region}\ p}$	(store properties)

Fig. 3. Environments

3.1 Kinds of Types

In Fig. 2 the judgment $ke \mid sp \vdash_t t :: k$ reads: "with kind environment *ke* and store properties *sp*, type *t* has kind *k*."

Rule (KiVar) retrieves the kind of a type variable from position *ix* in the kind environment *ke*, using the 'get' meta function.

Rule (KiRgn) requires a region handle to have a corresponding entry in the store properties list. The store properties model which regions currently exist in the store, so we know that if a region handle exists in the program then the corresponding region exists in the runtime store.

Rule (KiForall) requires the body type to have kind *Data* to mirror the corresponding formation rule for type abstractions, (TvLAM) in Fig. 4.

Rules (KiSum) and (KiBot) require the types used as arguments to a type sum to have *Effect* kind. During the development of this work we tried an alternate presentation where effects and effect sums were separated into their own syntactic class, instead of including them in the general type language, but this introduced much superficial detail in the formalization. If effects were separated into their own syntactic class then we would also need separate effect abstraction and effect application forms in the expression language. We would also need to prove administrative properties about de Bruijn lifting and substitution separately for effects as well as general types. Including effect sums in the general type language turned out to be much simpler.

Rule (KiApp) prevents the result of a type application from having *Region* kind. This restriction is needed to provide the canonical forms Lemma 3.1 which states that every closed type of kind *Region* is a region handle. If we allowed type applications to have *Region* kind then this would not be true. We have not thought of a situation where relaxing this restriction would be useful.

Rules (KiCon0) - (KiCon2) give the kinds of primitive type constructors, using the auxiliary meta-level functions 'kindOfTyCon0' – 'kindOfTyCon2'. These auxiliary functions are used for proof engineering reasons: they reduce the number of kinding rules and allow us to add new type constructors without disturbing the body of the proof.

3.2 Properties of the kinding rules

Lemma 3.1. *A closed type of kind Region is a region handle.*

If $\cdot \mid sp \vdash_t t :: \mathit{Region}$
then $(\text{exists } p.\ t = \texttt{rgn}\ p)$

Used in the proof of Progress (Theorem 4.2) to ensure that the region types passed to primitive operators like `read` and `write` are indeed region handles. □

Lemma 3.2. *We can insert a new element into the kind environment at position ix, provided we lift existing references to elements higher than this across the new one.*

If $ke \mid sp \vdash_t t :: k_1$
then insert $ix\ k_2\ ke \mid sp \vdash_t t \uparrow^{ix} :: k_1$

The syntax $t \uparrow^{ix}$ is the de Bruijn index lifting operator for type expressions. The application (insert ix k_2 ke) inserts element k_2 at position ix in the list ke, using the meta-function 'insert'. □

Lemma 3.3. *Adding a new store property to the start or end of the list preserves the inferred kind of a type.*

If $ke \mid sp \vdash_t t :: k$
then $ke \mid sp,\ p \vdash_t t :: k$

If $ke \mid sp \vdash_t t :: k$
then $ke \mid p,\ sp \vdash_t t :: k$

These weakening lemmas are used when adding allocating a new region in the store. In this paper presentation we overload the comma operator to add elements to the beginning and end of an environment, as well as to append two environments. □

3.3 Types of Expressions

Fig. 4 contains the mutually recursive judgments that assign a type to a value, and a type and effect to an expressions. The judgment $ke \mid te \mid se \mid sp \vdash_v v :: t$ reads: "with kind environment *ke*, type environment *te*, store environment *se* and store properties *sp*, value *v* has type *t*". Similarly judgment $ke \mid te \mid se \mid sp \vdash_x x :: t ; e$ reads: "... expression *x* has type *t* and effect *e*".

Rule (TvVar) retrieves the type of an expression variable from the type environment *te*. The second premise requires all expression variables to have kind *Data*, which is needed for Lemma 3.4, to ensure that the data type and effect produced by the typing judgments have the corresponding kinds. Although this restriction is not commonly enforced in semi-formal presentations of the ambient System-F calculus, it is useful in a mechanized proof so we do not need to manage a separate statement of well-formedness for the type environment.

Rule (TvLoc) retrieves the type of a location from the store environment *se*. As with rule (TyVar), we require the types in the store environment to have kind *Data*. All store locations are mutable references, so we can always attach the corresponding region variable to their types.Values of primitive types such as *Nat* and *Bool* are not tagged with a region variable, and do not appear naked in the store.

Rule (TvLam) checks the body of the function abstraction x_2 in a type environment extended with the parameter of the abstraction. The type of the overall abstraction includes the effect e_2 of evaluating its body. The first premise rejects functional values for which no corresponding arguments exist, and ensures that types of higher kind are not added to the type environment. For example, the expression $(\lambda\{z\} : \mathit{Ref}\,.\,5)$ can never be applied because there is no way to introduce an argument of type *Ref* (of kind $\mathit{Region} \rightsquigarrow \mathit{Data} \rightsquigarrow \mathit{Data}$) other than by wrapping it in a similarly bogus function abstraction.

Rule (TvLAM) checks the body of a type abstraction x_2 in a kind environment extended with the parameter of the abstraction. As we are using the de Bruijn representation for binders, when we push the new kind k_2 onto the *front* of the kind environment, we must also lift type indices in the existing type and store environments across the new element. In this paper we overload the $\uparrow$ symbol to represent the lifting operators for type and store environments, as well as for individual types and expressions. We write a plain $\uparrow$ as shorthand for $\uparrow^0$, meaning that lifting starts at index $\underline{0}$.

We require the body of a type abstraction to be pure (have effect $\bot$) to avoid the well known soundness problem with polymorphic mutable references (Leroy, 1993). In ML dialects this problem is typically mitigated by some version of the *value restriction* (Garrigue, 2002). Using an effect system it is possible to handle the problem more gracefully, while still allowing the body of a type abstraction to have side effects (Talpin & Jouvelot, 1994). However, the matter of polymorphic mutable references is orthogonal to region deallocation, so for this paper we just require the body of a type abstraction to be pure.

Rule (TvConst) uses the meta-function 'typeOfConst' to get the type of a constant, which helps keep the number of typing rules down.

Rule (TxVal) injects values into the syntax of expressions, indicating that values are always pure.

Rule (TxLet) checks the body of the let-expression x_2 in a type environment extended with the type of the binding t_1. This is a non-recursive let-binding, so we check the bound expression with the original type environment *te*. Similarly to the (TvLam) rule, we require the bound variable to have a type of kind *Data*. Without this premise we could prove that the other typing rules require the right of the binding x_1 to have kind *Data* anyway, but writing this fact explicitly avoids needing to prove it separately.

Note that the effect of a `let`-expression includes the effect of evaluating the binding e_1 as well as the body e_2, so the overall expression has effect $e_1 + e_2$. This is also the only rule that performs an effect join. We gain this property from the fact that we use a `let`-normalized presentation, where applications are always between values.

Rule (TxApp) performs a function application that unleashes the effect e_1, which then appears in the consequent.

Rule (TxAPP) performs type application by substituting the argument t_2 into the type of the body t_{12}. In Fig. 4 we write $t_{12}[t_2/\underline{0}\{a\}]$ to indicate that t_2 is substituted for type index $\underline{0}$ in t_{12}.

Rule (TxPrivate) checks the body of a `private` construct in a kind environment extended with a new region variable. As with rule (TvLAM), we need to lift indices in the type and store environment across this new element. As discussed in §1.1, because the region $\{r\}$ is entirely local to the body of the `private` expression, we can mask effects on it. This masking is performed by the type expression $(\text{maskOnVarT}\ \underline{0}\{r\}\ e)$ which replaces the atomic *Read*, *Write* and *Alloc* effects in e that mention r with the pure effect $\bot$. As the body of a `private` expression is checked in a environment extended with the new region, we then need to lower indices in the data type and effect of the body (t and e) before producing the type and effect of the overall expression (t' and e'). In Fig. 4 we write the lowering operator as $\downarrow$.

The lowering operator $\downarrow$ only succeeds if its argument type does *not* contain the type index $\underline{0}$, corresponding to the bound variable $\{r\}$. In rule (TxPrivate), the effect being lowered is guaranteed not to include $\underline{0}\{r\}$ because we mask all terms that contain this index. However, in an ill-typed program it would be possible for the type expression t to include a use of $\underline{0}\{r\}$. The fact that the lowering of t succeeds only when it does not contain $\underline{0}\{r\}$ is equivalent to including the premise $r \notin fv(t)$ (checking that the free variables of t do not include r). This latter premise is seen in presentations of effect system that use named binders rather than de Bruijn indices.

Rule (TxExtend) checks the body of an `extend` construct in a kind environment extended with the new region variable $\{r_2\}$. The type of the overall expression is then the type of the body, but with the outer region type t_1 substituted for $\underline{0}\{r_2\}$. This substitution reflects the fact that once the evaluation of the body has completed, all store bindings in the inner region $\{r_2\}$ will be merged into the outer region, represented by t_1. As with Rule (TxPrivate) we also mask the effects on the new region, though in this case the overall expression is assigned an *Alloc* t_1 effect to reflect the fact that store bindings that were allocated into the new region are retained instead of being deallocated.

Rules (TxAlloc), (TxRead) and (TxWrite) are straightforward. Each act on a reference in a region of type t_1 and produce the corresponding effect.

Rule (TxOpPrim) uses the auxiliary function 'typeOfOp1' to get the types of each primitive operator.

$$\boxed{kienv \mid tyenv \mid stenv \mid stprops \vdash_v val :: type} \qquad \text{(TypeV)}$$

$$\frac{t = \text{get } ix\{z\}\ te \qquad ke \mid sp \vdash_t t :: Data}{ke \mid te \mid se \mid sp \vdash_v ix\{z\} :: t} \qquad \text{(TvVar)}$$

$$\frac{Ref\ r\ t = \text{get } l\ se \qquad ke \mid sp \vdash_t Ref\ r\ t :: Data}{ke \mid te \mid se \mid sp \vdash_v \texttt{loc}\ l :: Ref\ r\ t} \qquad \text{(TvLoc)}$$

$$\frac{ke \mid sp \vdash_t t_1 :: Data \qquad ke \mid te, \{z\} : t_1 \mid se \mid sp \vdash_x x_2 :: t_2 \,;\, e_2}{ke \mid te \mid se \mid sp \vdash_v \lambda\{z\} : t_1 . x_2 :: t_1 \xrightarrow{e_2} t_2} \qquad \text{(TvLam)}$$

$$\frac{ke, \{a\} : k_1 \mid te\uparrow \mid se\uparrow \mid sp \vdash_x x_2 :: t_2 \,;\, \bot}{ke \mid te \mid se \mid sp \vdash_v \Lambda\{a\} : k_1 . x_2 :: \forall\{a\} : k_1 . t_2} \qquad \text{(TvLAM)}$$

$$\frac{t = \text{typeOfConst } c}{ke \mid te \mid se \mid sp \vdash_v c :: t} \qquad \text{(TvConst)}$$

$$\boxed{kienv \mid tyenv \mid stenv \mid stprops \vdash_x exp :: type \,;\, type} \qquad \text{(TypeX)}$$

$$\frac{ke \mid te \mid se \mid sp \vdash_v v_1 :: t_1}{ke \mid te \mid se \mid sp \vdash_x v_1 :: t_1 \,;\, \bot} \qquad \text{(TxVal)}$$

$$\frac{\begin{array}{l} ke \mid sp \vdash_t t_1 :: Data \\ ke \mid te \mid se \mid sp \vdash_x x_1 :: t_1 \,;\, e_1 \qquad ke \mid te, \{z\} : t_1 \mid se \mid sp \vdash_x x_2 :: t_2 \,;\, e_2 \end{array}}{ke \mid te \mid se \mid sp \vdash_x \texttt{let}\ \{z\} : t_1 = x_1\ \texttt{in}\ x_2 :: t_2 \,;\, e_1 + e_2} \qquad \text{(TxLet)}$$

$$\frac{ke \mid te \mid se \mid sp \vdash_v v_1 :: t_{11} \xrightarrow{e_1} t_{12} \qquad ke \mid te \mid se \mid sp \vdash_v v_2 :: t_{11}}{ke \mid te \mid se \mid sp \vdash_x v_1\ v_2 :: t_{12} \,;\, e_1} \qquad \text{(TxApp)}$$

$$\frac{ke \mid te \mid se \mid sp \vdash_v v_1 :: \forall\{a\} : k_{11} . t_{12} \qquad ke \mid sp \vdash_t t_2 :: k_{11}}{ke \mid te \mid se \mid sp \vdash_x v_1\ t_2 :: t_{12}[t_2/\underline{0}\{a\}] \,;\, \bot} \qquad \text{(TxAPP)}$$

$$\frac{\begin{array}{r} e' = (\text{maskOnVarT } \underline{0}\{r\}\ e)\downarrow \\ t' = t\downarrow \quad ke, \{r\} : Region \mid te\uparrow \mid se\uparrow \mid sp \vdash_x x :: t \,;\, e \end{array}}{ke \mid te \mid se \mid sp \vdash_x \texttt{private}\ \{r\}\ \texttt{in}\ x :: t' \,;\, e'} \qquad \text{(TxPrivate)}$$

$$\frac{\begin{array}{r} ke \mid sp \vdash_t t_1 :: Region \qquad e' = (\text{maskOnVarT } \underline{0}\{r_2\}\ e)\downarrow \\ t_3' = t_3[t_1/\underline{0}\{r_2\}] \quad ke, \{r_2\} : Region \mid te\uparrow \mid se\uparrow \mid sp \vdash_x x :: t_3 \,;\, e \end{array}}{ke \mid te \mid se \mid sp \vdash_x \texttt{extend}\ t_1\ \texttt{with}\ \{r_2\}\ \texttt{in}\ x :: t_3' \,;\, e' + Alloc\ t_1} \qquad \text{(TxExtend)}$$

$$\frac{ke \mid sp \vdash_t t_1 :: Region \qquad ke \mid te \mid se \mid sp \vdash_v v_2 :: t_2}{ke \mid te \mid se \mid sp \vdash_x \texttt{alloc}\ t_1\ v_2 :: Ref\ t_1\ t_2 \,;\, Alloc\ t_1} \qquad \text{(TxAlloc)}$$

$$\frac{ke \mid sp \vdash_t t_1 :: Region \qquad ke \mid te \mid se \mid sp \vdash_v v_2 :: Ref\ t_1\ t_2}{ke \mid te \mid se \mid sp \vdash_x \texttt{read}\ t_1\ v_2 :: t_2 \,;\, Read\ t_1} \qquad \text{(TxRead)}$$

$$\frac{\begin{array}{r} ke \mid te \mid se \mid sp \vdash_v v_2 :: Ref\ t_1\ t_2 \\ ke \mid sp \vdash_t t_1 :: Region \quad ke \mid te \mid se \mid sp \vdash_v v_3 :: t_2 \end{array}}{ke \mid te \mid se \mid sp \vdash_x \texttt{write}\ t_1\ v_2\ v_3 :: Unit \,;\, Write\ t_1} \qquad \text{(TxWrite)}$$

$$\frac{t_{11} \xrightarrow{e} t_{12} = \text{typeOfOp1 } op \qquad ke \mid te \mid se \mid sp \vdash_v v_1 :: t_{11}}{ke \mid te \mid se \mid sp \vdash_x op\ v_1 :: t_{12} \,;\, e} \qquad \text{(TxOpPrim)}$$

Fig. 4. Types of Values and Expressions

typeOfConst `unit` = *Unit*

typeOfConst `tt` = *Bool*
typeOfConst `ff` = *Bool*

typeOfConst `0` = *Nat*
typeOfConst `1` = *Nat* ...

typeOfOp1 `isZero` = *Nat* → *Bool*
typeOfOp1 `succ` = *Nat* → *Nat*

Fig. 5. Types of Primitive Constants and Operators

3.3.1 Properties of the typing rules

Lemma 3.4. *The data type and effect produced by a typing derivation have the appropriate kinds.*

If $ke \mid te \mid se \mid sp \vdash_x x :: t ; e$
then $ke \mid sp \vdash_t t :: Data$ and $ke \mid sp \vdash_t e :: Effect$ □

Lemma 3.5. *Two typing derivations for the same expression produce the same data type and effect.*

If $ke \mid te \mid se \mid sp \vdash_x x :: t_1 ; e_1$ and $ke \mid te \mid se \mid sp \vdash_x x :: t_2 ; e_2$
then $(t_1 = t_2)$ and $(e_1 = e_2)$ □

Lemma 3.6. *We can insert a new element into the kind environment at position ix, provided we lift existing references to elements higher than ix over the new one.*

If $ke \mid te \mid se \mid sp \vdash_x x_1 :: t_1 ; e_1$
then insert $ix\ k_2\ ke \mid te\uparrow^{ix} \mid se\uparrow^{ix} \mid sp \vdash_x x_1\uparrow^{ix}_t :: t_1\uparrow^{ix} ; e_1\uparrow^{ix}$

In this lemma the lifting operator $\uparrow^{ix}_t$ applies to the *type* indices in an expression, which is indicated by the t subscript on the arrow. □

Lemma 3.7. *We can insert a new element into the type environment at position ix, provided we lift existing references to elements higher than ix over the new one.*

If $ke \mid te \mid se \mid sp \vdash_x x_1 :: t_1 ; e_1$
then $ke \mid$ insert $ix\ t_2\ te \mid se \mid sp \vdash_x x_1\uparrow^{ix}_x :: t_1 ; e_1$

In this lemma the lifting operator $\uparrow^{ix}_x$ applies to the *expression* indices in x, which is indicated by the x subscript on the arrow. □

Lemma 3.8. *Substitution of types in expressions.*

If get $ix\ ke = k_2$
and $ke \mid te \mid se \mid sp \vdash_x x_1 :: t_1 ; e_1$
and delete $ix\ ke \mid sp \vdash_t t_2 :: k_2$
then delete $ix\ ke \mid te[t_2/ix] \mid se[t_2/ix] \mid sp \vdash_x x_1[t_2/ix] :: t_1[t_2/ix] ; e_1[t_2/ix]$

This is the type substitution lemma discussed in the motivation in §1.6. In the proof of Preservation it is used to show that the result of a type application has the correct type. □

Lemma 3.9. *Substitution of values in expressions.*

If get $ix\ te = t_2$
and $ke \mid te \mid se \mid sp \vdash_x x_1 :: t_1 ; e_1$
and $ke \mid$ delete $ix\ te \mid se \mid sp \vdash_v v_2 :: t_2$
then $ke \mid$ delete $ix\ te \mid se \mid sp \vdash_x x_1[v_2/ix] :: t_1 ; e_1$

This lemma is used in the proof of Preservation to show the result of a function application has the correct type.

□

Lemma 3.10. *Adding closed types to the end of the store environment preserves the inferred type and effect.*

If $ke \mid te \mid se_1 \mid sp \vdash_x x :: t_1 ; e_1$ and (Forall t in se_2. ClosedT t)
then $ke \mid te \mid se_2, se_1 \mid sp \vdash_x x :: t_1 ; e_1$

The 'ClosedT' predicate checks that its argument does not contain free type indices. Throughout this paper, when we describe a judgement informally instead of giving an explicit definition we write it with in prefix form (as with ClosedT t) instead of mixfix using a turnstyle symbol $\vdash$. See the accompanying Coq script for full details.

□

Lemma 3.11. *Adding a new property to the start or end of the store properties lists preserves the inferred type and effect.*

If $ke \mid te \mid se \mid sp \vdash_x x :: t ; e$
then $ke \mid te \mid se \mid sp, p \vdash_x x :: t ; e$

If $ke \mid te \mid se \mid sp \vdash_x x :: t ; e$
then $ke \mid te \mid se \mid p, sp \vdash_x x :: t ; e$

□

4 Dynamic Semantics

Fig. 6 contains the small step dynamic semantics for System-F^{re}. We use a frame stack to hold the continuation when evaluating a `let` binding, as well as to remember the set of currently live regions. The single step semantics is defined in the next section, and there is a trace of an example expression in Fig. 14.

4.1 Small Step Evaluation

The small step semantics of Fig. 6 is split into two judgment forms, one for the pure evaluation rules that do not affect the store or frame stack, and one for the others. We will describe the form of the store and frame stack when we discuss the specific rules that act upon it. The judgment $x \longrightarrow x'$ reads: "expression x evaluates to expression x'". The judgment $ss \mid sp \mid fs \mid x \longrightarrow ss' \mid sp' \mid fs' \mid x'$ reads: "with store ss, store properties sp and frame stack fs, evaluation of expression x produces a new store ss', properties sp', frame stack fs' and expression x'." We refer to a quadruple $(ss \mid sp \mid fs \mid x)$ as a *machine state*, so our evaluation judgment takes one machine state and produces a new one.

4.1.1 Pure Evaluation Rules

Rules (SpAppSubst) and (SpAPPSubst) are the usual value and type application rules for System-F based languages. Because expressions contain both expression and type indices, we disambiguate the substitution operator with a subscript indicating which sort of index it operates on. In (SpAppSubst) we write $x_{12}[v_2/\underline{0}\{x\}]_x$ to substitute v_2 for the expression index $\underline{0}$ in x_{12}. Likewise we write $x_{12}[t_2/\underline{0}\{x\}]_t$ for type substitution.

Rules (SpSucc) and (SpZero) evaluate our representative pure primitive operators. For the meta-level implementation of these operators, we use the Coq library functions 'S' and 'beq_nat' to take the successor and test for zero respectively.

Rule (SfStep) embeds a pure evaluation rule in one that contains the store and frame stack.

4.1.2 Frame Stacks, let-continuations and Deallocation

Fig. 7 gives the definition of frame stacks. A frame stack is a list of frames, where a frame can either be a let-continuation or region context. A let-continuation $\texttt{let}\ \{z\} : t = \circ\ \texttt{in}\ x_2$ holds the body of a let binding x_2 while the bound expression is being evaluated. We use the $\circ$ to indicate the part of the expression currently under evaluation. A region context frame $\texttt{priv}\ \textit{mode}\ p$ records the fact that region p has been created and can have new bindings allocated into it. The *mode* field says what to do with the store bindings in that region when we leave the scope of the construct that created it. For the `private` construct we use mode `d`, which indicates that all store bindings in p should be deallocated. For `extend` we use a mode like $(\texttt{m}\ p')$ which indicates that all store bindings in the inner region p should be merged into the outer region p'.

Returning to Fig. 6, rule (SfLetPush) enters a let-binding by pushing a continuation holding the body x_2 onto the stack, and then begins evaluation of the bound expression x_1.

Rule (SfLetPop) matches when the expression has reduced to a value and there is a let-continuation on the top of the stack. In this case we substitute the value into the body of the original let-expression x_2.

Rule (SfPrivatePush) allocates a new region. For this we use meta-function 'allocRegion' to examine the current list of store properties and produce a fresh region identifier p. The definition of 'allocRegion' in the Coq script just takes the maximum of all existing region identifiers and adds one to it. In a concrete implementation we could instead base the new identifier on a counter of previously allocated regions, or use the starting address of the new region as a fresh identifier. Having generated a fresh identifier, we then push a $\texttt{priv d}\ p$ frame onto the stack to record that the region has been allocated. As we will see in §4.3, we use the set of `priv` frames currently on the stack to determine what locations in the store are safe to access, and the new frame also indicates that all store bindings in region p are live (not yet deallocated). Finally, we substitute the region handle $\texttt{rgn}\ p$ for the original region variable in the body expression x_1. This substitution performs the *region phase change*, which means that any effects of x_1 had that mentioned the region variable $\underline{0}\{r\}$ now mention $\texttt{rgn}\ p$ instead — so an effect like *Read* r changes to *Read* $(\texttt{rgn}\ p)$. In our proof of Preservation we manage this phase change with the visible subsumption judgment described in §4.5.

$$\boxed{exp \longrightarrow exp} \qquad \text{(StepP)}$$

$$(\lambda\{z\} : t_{11}.\, x_{12})\; v_2 \longrightarrow x_{12}[v_2/\underline{0}\{z\}]_x \qquad \text{(SpAppSubst)}$$

$$(\Lambda\{a\} : k_{11}.\, x_{12})\; t_2 \longrightarrow x_{12}[t_2/\underline{0}\{a\}]_t \qquad \text{(SpAPPSubst)}$$

$$\texttt{succ}\; n \longrightarrow n' \;\text{ where }\; n' = \texttt{S}\; n \qquad \texttt{isZero}\; n \longrightarrow b' \;\text{ where }\; b' = \text{beq\_nat}\; n\; 0 \qquad \text{(SpSucc/Zero)}$$

$$\boxed{store \mid stprops \mid stack \mid exp \longrightarrow store \mid stprops \mid stack \mid exp} \qquad \text{(StepF)}$$

$$\frac{x \longrightarrow x'}{ss \mid sp \mid fs \mid x \longrightarrow ss \mid sp \mid fs \mid x'} \qquad \text{(SfStep)}$$

$$\begin{array}{lll} & ss \mid sp \mid fs & \mid \texttt{let}\; \{z\} : t = x_1\; \texttt{in}\; x_2 \\ \longrightarrow & ss \mid sp \mid fs, \texttt{let}\; \{z\} : t = \circ\; \texttt{in}\; x_2 & \mid x_1 \end{array} \qquad \text{(SfLetPush)}$$

$$\begin{array}{lll} & ss \mid sp \mid fs, \texttt{let}\; \{z\} : t = \circ\; \texttt{in}\; x_2 & \mid v_1 \\ \longrightarrow & ss \mid sp \mid fs & \mid x_2[v_1/\underline{0}\{z\}]_x \end{array} \qquad \text{(SfLetPop)}$$

$$\begin{array}{lll} & ss \mid sp \mid fs & \mid \texttt{private}\; \{r\}\; \texttt{in}\; x_1 \\ \longrightarrow & ss \mid \texttt{region}\; p, sp \mid fs, \texttt{priv}\; \texttt{d}\; p & \mid x_1[\texttt{rgn}\; p/\underline{0}\{r\}]_t \end{array} \;\text{ where }\; p = \text{allocRegion}\; sp \qquad \text{(SfPrivatePush)}$$

$$ss \mid sp \mid fs, \texttt{priv}\; \texttt{d}\; p \mid v_1 \longrightarrow ss' \mid sp \mid fs \mid v_1 \;\text{ where }\; ss' = \text{map}\; (\text{deallocB}\; p)\; ss \qquad \text{(SfPrivatePop)}$$

$$\begin{array}{lll} & ss \mid sp \mid fs & \mid \texttt{extend}\; (\texttt{rgn}\; p_1)\; \texttt{with}\; \{r\}\; \texttt{in}\; x_1 \\ \longrightarrow & ss \mid \texttt{region}\; p_2, sp \mid fs, \texttt{priv}\; (\texttt{m}\; p_1)\; p_2 & \mid x_1[\texttt{rgn}\; p_2/\underline{0}\{r\}]_t \end{array} \;\text{ where }\; p_2 = \text{allocRegion}\; sp \qquad \text{(SfExtendPush)}$$

$$ss \mid sp \mid fs, \texttt{priv}\; (\texttt{m}\; p_1)\; p_2 \mid v_1 \longrightarrow ss' \mid sp \mid fs \mid v_1 \;\text{ where }\; ss' = \text{map}\; (\text{mergeB}\; p_1\; p_2)\; ss \qquad \text{(SfExtendPop)}$$

$$ss \mid sp \mid fs \mid \texttt{alloc}\; (\texttt{rgn}\; p)\; v_1 \longrightarrow p\; \texttt{with}\; v_1, ss \mid sp \mid fs \mid \texttt{loc}\; l \;\text{ where }\; l = \text{length}\; ss \qquad \text{(SfStoreAlloc)}$$

$$ss \mid sp \mid fs \mid \texttt{read}\; (\texttt{rgn}\; p)\; (\texttt{loc}\; l) \longrightarrow ss \mid sp \mid fs \mid v \;\text{ where }\; p\; \texttt{with}\; v = \text{get}\; l\; ss \qquad \text{(SfStoreRead)}$$

$$ss \mid sp \mid fs \mid \texttt{write}\; (\texttt{rgn}\; p)\; (\texttt{loc}\; l)\; v_2 \longrightarrow ss' \mid sp \mid fs \mid \texttt{unit} \;\text{ where }\; p\; \texttt{with}\; v_1 = \text{get}\; l\; ss,\; ss' = \text{update}\; l\; (p\; \texttt{with}\; v_2)\; ss \qquad \text{(SfStoreWrite)}$$

Fig. 6. Small Step Evaluation

$$\begin{array}{lll} stack & ::= & \overline{frame} \qquad \text{(frame stacks)} \\ frame & ::= & \texttt{let}\; \{z\} : type = \circ\; \texttt{in}\; exp \qquad \text{(let-continuation)} \\ & \mid & \texttt{priv}\; mode\; p \qquad \text{(region context)} \\ mode & ::= & \texttt{d} \mid \texttt{m}\; p \qquad \text{(deallocate / merge)} \\ store & ::= & \overline{stbind} \qquad \text{(mutable stores)} \\ stbind & ::= & p\; \texttt{with}\; val \qquad \text{(live store binding)} \\ & \mid & p\; \texttt{with}\; \bullet \qquad \text{(dead store binding)} \end{array}$$

Fig. 7. Stores and Store Bindings

Rule (SfPrivatePop) matches when the expression has reduced to a value v_1. When the top-most frame on the stack is a `priv d` p we deallocate all bindings in region p and pop the frame. Fig. 7 shows that a store is a list of store bindings, which themselves may be live or dead. A live store binding p `with` *val* holds a value *val*, tagged with the region it is in p.

We model deallocation by replacing the value contained in the store binding by the placeholder •, which indicates that the value is no longer available. This is also the approach taken by Calcagno *et al.* (2002). In Fig. 6 the deallocation of a single binding is performed using the meta-function 'deallocB', which takes a region identifier p, and a store binding, and replaces the contained value with • if the binding is tagged with p. Note that in the formal semantics we cannot simply remove dead bindings from the store. If we removed them, then any dangling references to these bindings would no longer be well typed. Dangling references were discussed in §1.5.

Rules (SfExtendPush) and (SfExtendPop) are similar to (SfPrivatePush) and (SfPrivatePop), except that the version for `extend` also records the identifier of the outer region in the stack frame. In (SfExtendPop) when the frame `priv` (`m` p_1) p_2 is popped from the stack we use the meta-function 'mergeB' to merge all objects in region p_2 into region p_1, instead of simply deallocating them as before. The 'mergeB' function rewrites the region annotation on store bindings, so p_2 `with` v becomes p_1 `with` v for any v, and p_2 `with` • becomes p_1 `with` •. In the formal semantics we must also rewrite the region annotations on dead bindings to ensure that dangling references retain their correct types, though in a concrete implementation we would not need to perform this operation at runtime.

Rule (SfStoreAlloc) appends a new store binding in region p to the store, using the meta-function 'length' to get the location of this new binding.

Rule (SfStoreRead) reads the value of the binding at location l. The binding must be live for this to succeed. The Preservation theorem in §4.5.1 ensures that well typed programs never try to read dead bindings.

Rule (SfStoreWrite) first retrieves the binding at location l to ensure that it is live, and then overwrites it with the new value.

4.2 Store Environment and Well Formedness

The store environment contains the types of each location in the store, and was defined in Fig. 3. Well formedness for stores is defined in Fig. 8. The judgment $se \mid sp \vdash ss\ ;\ fs$ **wf** reads: "with store environment *se* and store properties *sp*, store *ss* and frame stack *fs* are well formed". To keep the formalism manageable, this well-formedness judgment is defined in terms of several auxiliary judgments. The first says that all types in the store environment must be closed. We describe the others in turn.

The judgment $se \mid sp \vdash_s ss$ reads: "with store environment *se* and store properties *sp*, store *ss* is well typed." A judgment of this form specifies that all store bindings in the store are closed and well typed with respect to their corresponding entries in the store environment. This judgment form is defined in terms of an auxiliary one $ke \mid te \mid se \mid sp \vdash_b b :: t$ that checks the type of a single store binding b. In the notation used in rule (StoreT), a statement like 'Forall2 b,t in ss,se. $P(b,t)$' asserts that property P

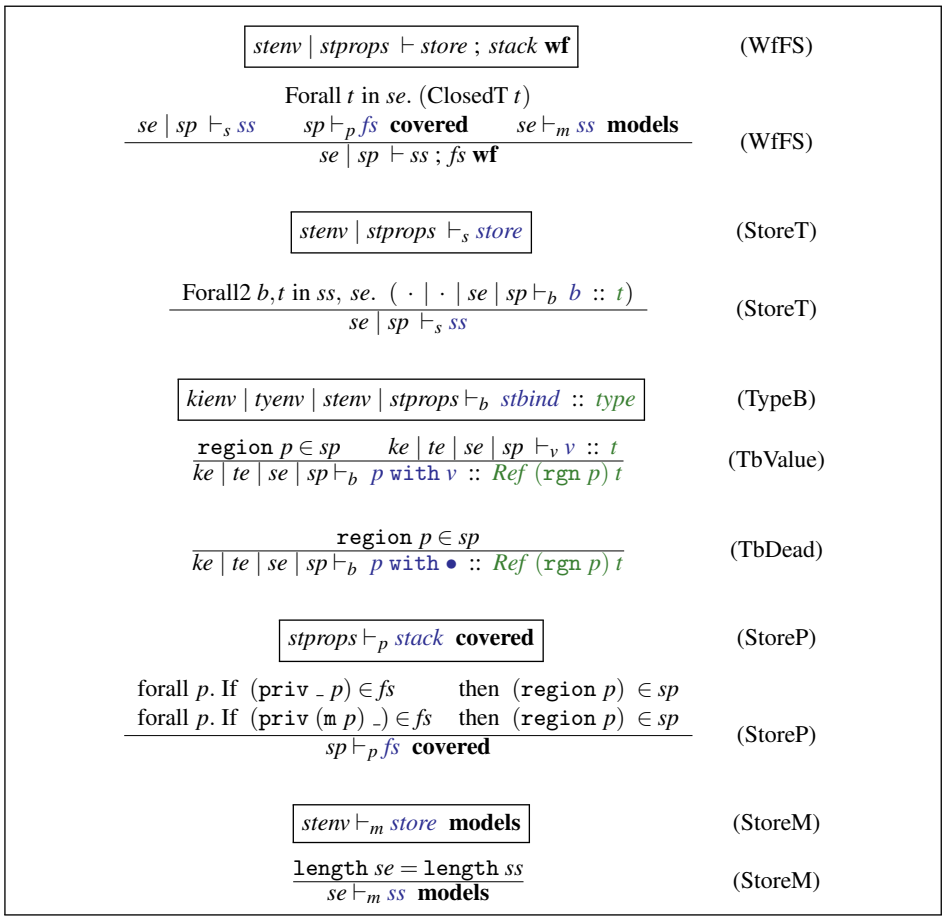

Fig. 8. Store Typing, Coverage, and Model

is true for pairs of elements b and t taken from the lists ss and se. The quantifier 'Forall2' comes as part of the standard Coq libraries, along with many administrative lemmas.

Rule (TbValue) says that a value v in some region p, written p `with` v, is well typed when the value itself is well typed and the region identifier p exists in the store properties. Rule (TbDead) is similar, though a deallocated value can be assigned any type, similarly to the `undefined` value from Haskell.

The judgment $sp \vdash_p fs$ **covered** reads: "with store properties sp, the frame stack fs is covered". The only inference rule (StoreP) ensures that every region identifier p that appears in a `priv` frame on the stack also appears in the store properties.

The judgment $se \vdash_m ss$ **models** reads: "the store environment se models the store ss". The only inference rule (StoreM) requires both se and ss to have the same length, which together with (StoreT) ensures that there are no entries in the store environment that do not have corresponding entries in the store.

4.2.1 Properties of the Store Typing

The following are the key lemmas used to show that the well formedness of the store is preserved during evaluation. We have one lemma for each of the rules of Fig 6 that modify the store.

Lemma 4.1. *Pushing a new* `priv` *frame on the stack, and appending the corresponding entry to the store environment preserves well formedness of the store.*

If $se \mid sp \vdash ss\ ;\ fs$ **wf**
then $se \mid \texttt{region}\ p,\ sp \vdash ss\ ;\ fs, \texttt{priv d}\ p$ **wf**

If $se \mid sp \vdash ss\ ;\ fs$ **wf** and $\texttt{region}\ p_1 \in sp$
then $se \mid \texttt{region}\ p_2,\ sp \vdash ss\ ;\ fs, \texttt{priv}\ (\texttt{m}\ p_1)\ p_2$ **wf**

During evaluation, new region identifiers are created by the (SfPrivatePush) and (SfExtendPush) rules of Fig 6. Note that the well formedness judgment itself does not require that the new region identifiers are fresh with respect to existing identifiers. Freshness is enforced by the (TypeF) judgment of Fig. 10, which we will discuss in §4.4.

□

Lemma 4.2. *Adding a closed store binding to the store, and corresponding entry to the store environment preserves the well formedness of the store.*

If $\cdot \mid \cdot \mid se \mid sp \vdash_v v :: t$ and $\texttt{region}\ p \in sp$
and $se \mid sp \vdash ss\ ;\ fs$ **wf**
then $Ref\ (\texttt{rgn}\ p)\ t,\ se \mid sp \vdash p\ \texttt{with}\ v,\ ss\ ;\ fs$ **wf**

□

Lemma 4.3. *Updating a store location with a closed well typed binding preserves the well formedness of the store.*

If $\text{get}\ l\ se = Ref\ p\ t$
and $\cdot \mid \cdot \mid se \mid sp \vdash_v v :: t$ and $\texttt{region}\ p \in sp$
and $se \mid sp \vdash ss\ ;\ fs$ **wf**
then $se \mid sp \vdash \text{update}\ l\ (p\ \texttt{with}\ v)\ ss\ ;\ fs$ **wf**

□

Lemma 4.4. *Deallocating a region preserves the well formedness of the store.*

If $se \mid sp \vdash ss\ ;\ fs, \texttt{priv d}\ p$ **wf**
then $se \mid sp \vdash \text{map}\ (\text{deallocB}\ p)\ ss\ ;\ fs$ **wf**

The bare fact that deallocating a region preserves the well formedness of the store is straightforward to prove, because the (TbDead) rule of Fig. 8 allows deallocated bindings to have the same types as they did before deallocation. Proving that subsequent reduction does not get stuck requires further machinery, which we discuss in §4.3.

□

Lemma 4.5. *Merging bindings into an existing region preserves the well formedness of the store.*

If $se \mid sp \vdash ss\,;\, fs,\ \texttt{priv}\ (\texttt{m}\ p_1)\ p_2$ **wf**
and $\texttt{region}\ p_1 \in sp$
then map (mergeT $p_1\ p_2$) $se \mid sp \vdash$ map (mergeB $p_1\ p_2$) $ss\,;\, fs$ **wf**

When we merge bindings from one region into another, we must update their corresponding types in the store environment to match. This is achieved with the 'mergeT' meta-function, where the application (mergeT $p_1\ p_2\ t$) rewrites all region identifiers p_2 to p_1 in type t. □

4.3 Liveness

Fig. 9 gives the key liveness invariants that ensure a running program will only accesses store bindings that exist in the store, and have not yet been deallocated. The judgment $fs \vdash_e e$ **live** reads: "frame stack fs is live relative to effect e", and the judgment $ss \vdash_s fs$ **live** reads: "store ss is live relative to frame stack fs". The first says that for every region identifier p in some effect e, there is a corresponding `priv` $m\ p$ frame on the stack fs. The second says that for every region identifier mentioned in a `priv` frame on the stack, all the store bindings in the corresponding regions are live. The fact that every `read` and `write` statement in the program is assigned an appropriate effect by the rules of Fig. 4 then makes it straightforward to reason that the evaluation of these statements only accesses store bindings that currently exist in the store.

In the (LiveE) rule of Fig. 9 the meta-function 'flattenT' takes a compound effect and produces a list of its atomic components. For example, effect $Read\ r_1 + (Write\ r_2 + Write\ r_3)$ flattens to the list $Read\ r_1$, $Write\ r_2$, $Write\ r_3$.

4.3.1 Properties of Liveness

The following lemmas are used to ensure that the liveness invariants between the store, frame stack and effect of an expression are preserved under during evaluation.

Lemma 4.6. *If the store is live relative to the frame stack, and frame stack live relative to an atomic effect on some region p, then all store bindings in region p are live.*

If $ss \vdash_e fs$ **live** and $fs \vdash_s e$ **live**
and $b = \text{get}\ l\ ss$
and $p = \text{regionOfStBind}\ b = \text{handleOfEffect}\ e$
then exists $v.\ b = p\ \texttt{with}\ v$ □

Lemma 4.7. *Masking effect on some variable with index ix preserves the liveness relationship with the frame stack.*

If $fs \vdash_e \text{maskOnVarT}\ ix\{v\}\ e$ **live**
then $fs \vdash_e e$ **live**

The lemma is "obviously true", because the rule (LiveE) rule only mentions region *handles* rather than the region *variables* that are being masked, though its proof requires some boilerplate to deal with the fact that the effect e being flattened in the rule.

□

Lemma 4.8. *If there is a* `priv` *m p frame on the frame stack, for some mode m, then substituting* `rgn` *p for variable* $\underline{0}$ *in some effect e preserves liveness of the stack relative to that effect.*

If fs, `priv` m $p \vdash_e e$ **live**
then fs, `priv` m $p \vdash_e e[\texttt{rgn}\ p/\underline{0}]$ **live**

Suppose the evaluation of the `private` expression creates a new region with identifier p. Given ($fs \vdash_e e$ **live**), it is trivial to show that the weakened version (fs, `priv d` $p \vdash_e e$ **live**) is also true. Using the above lemma, we can then substitute p into the body of the `private` expression e and show that the resulting frame stack is still live relative to the phase changed effect, (fs, `priv d` $p \vdash_e e[\texttt{rgn}\ p/\underline{0}]$ **live**).

□

Lemma 4.9. *If there is a* `priv d` *p frame on the top of the frame stack, then provided p is not used in a* `priv` *frame lower in the stack (NoPrivFs p fs), then popping the top frame while deallocating all store bindings in region p preserves liveness of the store relative to the frame stack.*

If $ss \vdash_s fs$, `priv d` p **live**
and NoPrivFs p fs
then map (deallocB p) $ss \vdash_s fs$ **live**

This key lemma shows that the liveness invariant between the store and the frame stack is preserved when applying rule (SfPrivatePop) of Fig. 6. Assuming the frame on the top of the stack is `priv d` p, once the body of the associated `private` expression reduces to a value, we pop the top frame and deallocate all store bindings in region p. Due to the NoPrivFs premise, we know that all store bindings in regions mentioned by `priv` frames deeper in the stack are still live.

□

Lemma 4.10. *If all store bindings in region p_2 are live, and the store is live relative to the frame stack fs, then merging region p_2 into some other region p_1 preserves liveness of the store relative to the frame stack.*

If (forall b. $b \in ss$ implies $b \vdash_{bp} p_2$ **live**)
and $ss \vdash_s fs$ **live**
then map (mergeB p_1 p_2) $ss \vdash_s fs$ **live**

This lemma shows that the liveness invariant between the store and frame stack is preserved when applying rule (SfExtendPop) of Fig. 6. If all store bindings in region p_2 are live, then when we merge them into region p_1 they are still live.

□

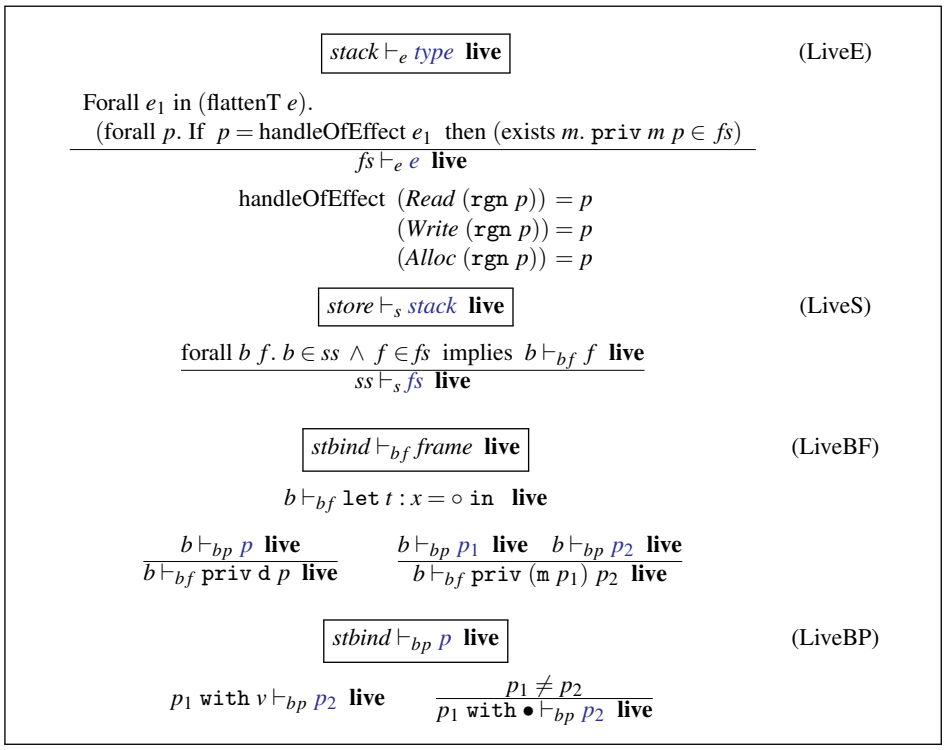

Fig. 9. Liveness of Effects and Frame Stacks

4.4 Types of Configurations and Frame Stacks

A configuration combines a frame stack fs with some expression x and is written fs/x. As discussed earlier, the frame stack describes the context in which the expression evaluates. The typing rules for configurations and frame stacks are given in Fig. 10. At the top of the figure, the judgment $ke \mid te \mid se \mid sp \vdash_c fs/x :: t ; e$ reads: "with kind environment ke, type environment te, store environment se and store properties sp, frame stack fs with expression x has type t and effect e". The second judgment form checks the frame stack itself, where $ke \mid te \mid se \mid sp \vdash_f fs :: t_1 \multimap t_2 ; e$ reads "... frame stack fs takes an expression of type t_1 to a result of type t_2, causing effect e". Here, the effect e is the effect of evaluating all the suspended let-continuations contained on the stack.

In Fig. 10, rule (TcExp) checks the type of an entire configuration. As with the other rules in the same figure, (TcExp) is a "dynamic typing" rule that is only needed when checking the program during evaluation. Before a source program has commenced evaluation it has no associated frame stack, so the static typing rules of Fig. 4 suffice to check it. The premise $ke \mid sp \vdash e_1 + e_2 \equiv e_3 :: Effect$ is needed to normalize the syntactic form of the overall effect e_3. The premise allows us to treat an effect term such as $(e_1 + \bot)$ as simply e_1, and use the usual commutativity and associativity properties of $+$. In hand written proofs these properties are typically used without mention, but in a mechanical proof we must be explicit. The equivalence judgment itself is defined in Fig. 11 and discussed in the next section.

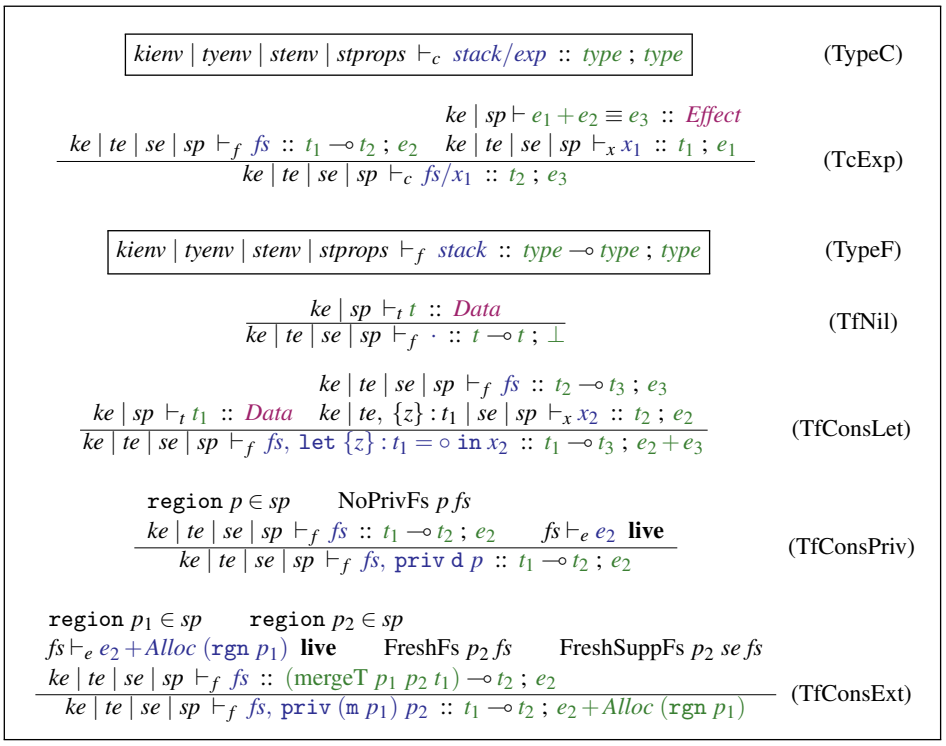

Fig. 10. Types of Configurations and Frame Stacks

Rule (TcNil) shows that an empty frame stack takes an expression of type t to another expression of type t (itself), performing no effects.

Rule (TfConsLet) gives the type of a frame stack where the top most frame is a let-continuation. Recall that a let-continuation holds the body of a let-expression while the binding is being evaluated. In the rule, the binding has type t_1 and the body x_2 has type t_2. As per rule (SfLetPop) from Fig. 6, once the let-binding has reduced to a value it will be substituted into the body x_2 held in the let-continuation. Further evaluation of the body will then produce a value of type t_2 which will cause the next frame on the rest of the stack fs to be popped, and so on. For this reason fs has type $t_2 \multimap t_3$, where t_3 is the type of the overall value that the configuration produces.

Rule (TfConsPriv) gives the type of a frame stack with a `priv d` p frame on top, for some region identifier p. Recall from the discussion of (SfPrivatePop) in §4.1.2 that a frame `priv d` p indicates that when the expression being evaluated has reduced to a value then we should deallocate all store bindings in region p and pop the frame. Unlike the previous case with (SfLetPop), in this case when we pop the frame we keep the original value, so the rest of the stack fs must also accept this value — hence we have $t_1 \multimap t_2$ in both the premise and conclusion. The premise NoPrivFs p fs says that the region identifier p cannot be mentioned in `priv` frames in the rest of the stack fs, which inductively ensures that all region identifiers in `priv` frames on the stack are distinct. Finally, the premise $fs \vdash_s e_2$ **live** ensures that the liveness of the overall stack (fs, `priv d` p) relative to e_2 will be preserved once we come to pop the `priv d` p frame later in the evaluation.

$$\boxed{kienv \mid stprops \vdash type \equiv type \;::\; kind} \qquad \text{(EquivT)}$$

$$\frac{ke \mid sp \vdash_t t \;::\; k}{ke \mid sp \vdash t \equiv t \;::\; k} \qquad \text{(EqRefl)}$$

$$\frac{\{ke \mid sp \vdash_t t_i \;::\; k\}^{i \leftarrow 1..2} \quad ke \mid sp \vdash t_1 \equiv t_2 \;::\; k}{ke \mid sp \vdash t_2 \equiv t_1 \;::\; k} \qquad \text{(EqSym)}$$

$$\frac{ke \mid sp \vdash t_1 \equiv t_2 \;::\; k \qquad ke \mid sp \vdash t_2 \equiv t_3 \;::\; k}{ke \mid sp \vdash t_1 \equiv t_3 \;::\; k} \qquad \text{(EqTrans)}$$

$$\frac{ke \mid sp \vdash t_1 \equiv t_1' \;::\; \mathit{Effect} \qquad ke \mid sp \vdash t_2 \equiv t_2' \;::\; \mathit{Effect}}{ke \mid sp \vdash t_1 + t_2 \equiv t_1' + t_2' \;::\; \mathit{Effect}} \qquad \text{(EqSumCong)}$$

$$\frac{ke \mid sp \vdash_t t \;::\; \mathit{Effect}}{ke \mid sp \vdash t \equiv t + \bot \;::\; \mathit{Effect}} \qquad \text{(EqSumBot)}$$

$$\frac{ke \mid sp \vdash_t t \;::\; \mathit{Effect}}{ke \mid sp \vdash t \equiv t + t \;::\; \mathit{Effect}} \qquad \text{(EqSumIdemp)}$$

$$\frac{\{ke \mid sp \vdash_t t_i \;::\; \mathit{Effect}\}^{i \leftarrow 1..2}}{ke \mid sp \vdash t_1 + t_2 \equiv t_2 + t_1 \;::\; \mathit{Effect}} \qquad \text{(EqSumComm)}$$

$$\frac{\{ke \mid sp \vdash_t t_i \;::\; \mathit{Effect}\}^{i \leftarrow 1..3}}{ke \mid sp \vdash t_1 + (t_2 + t_3) \equiv (t_1 + t_2) + t_3 \;::\; \mathit{Effect}} \qquad \text{(EqSumAssoc)}$$

Fig. 11. Type Equivalence

Rule (TfConsExt) gives the type of a frame stack with a $\texttt{priv}\ (\texttt{m}\ p_1)\ p_2$ frame on top, for outer and inner region identifiers p_1 and p_2. From the discussion of rule (SfExtendPop) from §4.1.2, recall that the frame $\texttt{priv}\ (\texttt{m}\ p_1)\ p_2$ indicates that when the expression being evaluated has reduced to a value, then we should merge region p_2 into region p_1 and pop the frame. The merging process rewrites p_2 into p_1 in both the value and in all store bindings, hence we must also rewrite the type of the value to match. The rest of the stack *fs* thus has type $(\text{mergeT}\ p_1\ p_2\ t_1) \multimap t_2$, as it expresses a continuation that takes the value after merging.

The premise $\text{FreshFs}\ p_2\ fs$ says that none of the frames in stack *fs* may mention the region identifier p_2, which prevents it from appearing in let-continuation frames as well as `priv` frames. This FreshFs predicate is stronger than the NoPrivFs predicate used in rule (TfConsPriv). We need the stronger version here because if an intermediate let-continution in *fs* actually did mention p_2 then its type would also change during the merging process. However, at runtime such a let-continuation cannot be constructed, because from the form of the stack we know that p_2 will have been created after any let-continuations deeper in the stack were pushed. The premise $\text{FreshSuppFs}\ p_2\ se\ fs$ says that for all store locations mentioned in *fs*, their corresponding types from *se* cannot mention region identifier p_2 (where "Supp" is short for "Support"). This premise is needed for the same reason as the previous one: if *fs* mentioned any locations that had types involving p_2 then these would change during merging, but such locations cannot exist because p_2 is guaranteed to be created after the locations.

$$\boxed{kienv \mid stprops \vdash type \sqsupseteq type \;::\; kind} \qquad \text{(SubsT)}$$

$$\frac{ke \mid sp \vdash t_1 \equiv t_2 \;::\; k}{ke \mid sp \vdash t_1 \sqsupseteq t_2 \;::\; k} \qquad \text{(SbEquiv)}$$

$$\frac{ke \mid sp \vdash t_1 \sqsupseteq t_2 \;::\; k \qquad ke \mid sp \vdash t_2 \sqsupseteq t_3 \;::\; k}{ke \mid sp \vdash t_1 \sqsupseteq t_3 \;::\; k} \qquad \text{(SbTrans)}$$

$$\frac{ke \mid sp \vdash_t t \;::\; Effect}{ke \mid sp \vdash t \sqsupseteq \bot \;::\; Effect} \qquad \text{(SbBot)}$$

$$\frac{ke \mid sp \vdash t_1 \sqsupseteq t_2 \;::\; Effect \qquad ke \mid sp \vdash t_1 \sqsupseteq t_3 \;::\; Effect}{ke \mid sp \vdash t_1 \sqsupseteq t_2 + t_3 \;::\; Effect} \qquad \text{(SbSumAbove)}$$

$$\frac{ke \mid sp \vdash t_1 \sqsupseteq t_2 \;::\; Effect \qquad ke \mid sp \vdash_t t_3 \;::\; Effect}{ke \mid sp \vdash t_1 + t_3 \sqsupseteq t_2 \;::\; Effect} \qquad \text{(SbSumBelow)}$$

$$\frac{ke \mid sp \vdash t_1 \sqsupseteq t_2 + t_3 \;::\; Effect}{ke \mid sp \vdash t_1 \sqsupseteq t_2 \;::\; Effect} \qquad \text{(SbSumAboveLeft)}$$

$$\frac{ke \mid sp \vdash t_1 \sqsupseteq t_2 + t_3 \;::\; Effect}{ke \mid sp \vdash t_1 \sqsupseteq t_3 \;::\; Effect} \qquad \text{(SbSumAboveRight)}$$

Fig. 12. Type Subsumption

4.5 Type Equivalence and Subsumption

The rules for type equivalence are given in Fig. 11. As discussed in the previous section, we use type equivalence in rule (TcExp) of Fig. 10 to normalize the effect of the reduction. The rules of Fig. 11 are completely standard, though note that the relation also requires the types mentioned to be well kinded. This property is needed to prove the administrative lemmas that are used in the body of the proof.

Interestingly, the type (and effect) equivalence relation is not sufficient to make a general statement of Preservation. An example trace that highlights the problem is given in Fig. 14. On the left of the figure we have the evaluation state as per the small step evaluation rules of Fig. 6, and on the right we have the effect of the configuration. This is the effect gained by applying the (TcExp) rule Fig. 10, using an empty kind and type environment. Note that the syntactic effect of the configuration is *not preserved* during evaluation. For example, the effect if the initial state is $(Write\ (\texttt{rgn}\ 0))$, but it changes to $\bot$ in the next state and then increases to $(Alloc\ (\texttt{rgn}\ 1) + Write\ (\texttt{rgn}\ 1))$ in the next.

The point about Fig 14 is that a running program dynamically allocates and deallocates new regions, and the effect we assign to intermediate states rightly reflects these changes. As we expose this runtime detail, we must define what it means for such an effect to be "valid", as it would not make sense for it to change to something completely arbitrary. As far as the client programmer is concerned, the observable effect of a closed, well typed program that begins evaluation in an empty store is precisely nothing. Such a program may allocate and deallocate new regions during evaluation, but because it cannot affect any *existing* bindings in the store (because there were none), it must be observationally pure.

We relate the effects of each successive program state with *visible subsumption*, defined in Fig 13. The judgment $ke \mid sp \vdash_{vis} e_1 \sqsupseteq e_2 \dagger sp'$ reads "with kind environment ke and store properties sp, effect e_1 visibly subsumes effect e_2 relative to the new store properties sp'".

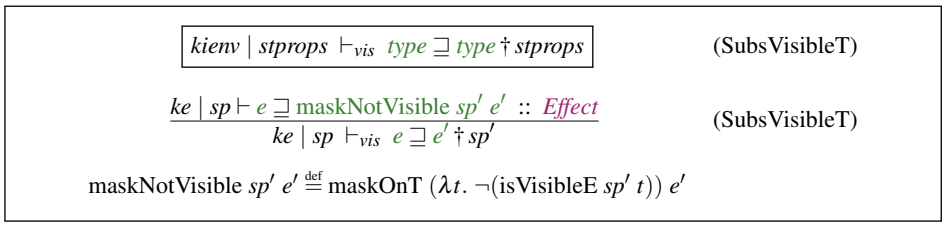

Fig. 13. Visible Subsumption

This judgment is defined in terms of the standard subsumption judgment, which is given in Fig. 12. The effect e_1 visibly subsumes effect e_2 relative to store properties sp', when e_1 subsumes e_2 after masking out all atomic effects in e_2 on regions that are not in sp'. In our statement of Preservation, we will use visible subsumption to mean "subsumption without worrying about regions that haven't been allocated yet".

As an example, the effect of the first state in Fig. 14 (*Write* (`rgn` 0)) visibly subsumes the effect of the third state (*Alloc* (`rgn` 1) + *Write* (`rgn` 1)) relative to the store properties (`region` 0). To write this statement formally, first note that the judgment form in Fig. 13 takes two separate lists of store properties. The properties on the left of the turnstile must mention the region identifiers in *both* effects being related, whereas the properties on the right mention just the identifiers of visible regions that we use to perform the masking. Here is the example statement in full, using an empty kind environment:

$$\cdot \mid \texttt{region}\,0, \texttt{region}\,1 \vdash_{vis} \mathit{Write}\ (\texttt{rgn}\ 0) \sqsupseteq \mathit{Alloc}\ (\texttt{rgn}\ 1) + \mathit{Write}\ (\texttt{rgn}\ 1) \dagger \texttt{region}\ 0$$

In the definition in Fig. 13, the meta-function 'maskNotVisible' takes a list of store properties sp', an effect e', and replaces atomic effect terms in e' that act on regions that are not mentioned in sp' with $\bot$. For example:

$$\begin{aligned}&\text{maskNotVisible}\ [\texttt{region}\ 0, \texttt{region}\ 2]\ (\mathit{Read}\ (\texttt{rgn}\ 0) + \mathit{Write}\ (\texttt{rgn}\ 1) + \mathit{Alloc}\ (\texttt{rgn}\ 2))\\ &\quad = \mathit{Read}\ (\texttt{rgn}\ 0) + \bot + \mathit{Alloc}\ (\texttt{rgn}\ 2)\end{aligned}$$

The meta-function 'maskNotVisible' itself is defined in terms of 'maskOnT', which is a higher order function that masks out atomic effect terms that do match the given predicate. In this case the required predicate is defined in terms of 'isVisibleE'. The expression 'isVisibleE sp' t' returns false when t is an atomic effect on some region that is not listed in sp', and true otherwise. In the Coq script we reuse the meta-function 'maskOnT' to define 'maskOnVarT', which appears in Fig. 4, as well as administrative lemmas about it.

4.5.1 Properties of Visible Subsumption

Lemma 4.11. *If effect e_1 visibly subsumes an effect e_2 that has terms involving region variable $\underline{0}$ masked out, then e_1 also visibly subsumes effect e_2 after substituting region handle `rgn` p for the masked variable, provided p is not a visible region identifier.*

$$\begin{aligned}&\text{If} && \neg(\texttt{region}\ p \in \mathit{spVis})\\ &\text{and} && \cdot \mid sp \vdash_{vis} e_1 \sqsupseteq \text{maskOnVarT}\ \underline{0}\ e_2 \dagger \mathit{spVis}\\ &\text{then} && \cdot \mid sp \vdash_{vis} e_1 \sqsupseteq e_2[(\texttt{rgn}\ p)/\underline{0}] \dagger \mathit{spVis}\end{aligned}$$

In the proof of Preservation, this lemma manages the region phase change which occurs to the overall effect of the program state when when apply rule (SfPrivatePush) from Fig. 6.

Here it is again:

$$\begin{array}{lll} & ss \mid sp \mid fs & \mid \texttt{private}\ \{r\}\ \texttt{in}\ x_2 \\ \longrightarrow & ss \mid \texttt{region}\ p,\ sp \mid fs,\ \texttt{priv d}\ p & \mid x_2[\texttt{rgn}\ p/\underline{0}\{r\}]_t \qquad \text{(SfPrivatePush)} \\ & \text{where} \quad p \ = \text{allocRegion}\ sp & \end{array}$$

Suppose x_2 has effect e_2 and the overall expression ($\texttt{private}\ r\ \texttt{in}\ x_2$) is closed. Using rule (TcExp) from Fig. 10 and rule (TxPrivate) from Fig. 4 with empty environments and frame stack, we assign the overall expression the effect (maskOnVarT $\underline{0}\{r\}\ e_2) \downarrow$. As the environments are empty we know that the masked effect is closed, so the overall effect of the expression is just (maskOnVarT $\underline{0}\{r\}\ e_2$), without the lowering operator. Applying rule (SfPrivatePush) above yields the new expression ($x_2[\texttt{rgn}\ p/\underline{0}\{r\}]_t$), with a fresh region handle substituted for variable $\underline{0}\{r\}$. Using the System-F style type substitution lemma, the effect of the new expression becomes ($e_2[\texttt{rgn}\ p/\underline{0}\{r\}]_t$). The lemma above is then used in the proof of Preservation to show that whatever effect visibly subsumes the effect of a `private` expression before reduction also subsumes it afterwards. The premise $\neg(\texttt{region}\ p \in spVis)$ is satisfied automatically because the region identifier p is freshly allocated by (SfPrivatePush). □

Lemma 4.12. *Strengthening the related store properties preserves visible subsumption.*

$$\begin{array}{ll} \text{If} & spVis' \ \text{extends}\ spVis \\ \text{and} & ke \mid sp \vdash_{vis} e_1 \sqsupseteq e_2 \dagger spVis' \\ \text{then} & ke \mid sp \vdash_{vis} e_1 \sqsupseteq e_2 \dagger spVis \end{array}$$

$$\text{where } (spVis' \ \text{extends}\ spVis) \stackrel{\text{def}}{=} (\text{exists}\ sp.\ spVis' = sp +\!\!+ spVis)$$ □

Theorem 4.1 (Preservation).

$$\begin{array}{ll} \text{If} & se \mid sp \vdash ss : fs\ \textbf{wf} \quad \text{and} \quad ss \vdash_s fs\ \textbf{live} \quad \text{and} \quad fs \vdash_e e\ \textbf{live} \\ \text{and} & \cdot \mid \cdot \mid se \mid sp \vdash_c fs/x :: t ; e \\ \text{and} & ss \mid sp \mid fs \mid x \longrightarrow ss' \mid sp' \mid fs' \mid x' \\ \text{then} & \text{exists}\ se'\ e'. \\ & \quad se' \mid sp' \vdash ss' ; fs'\ \textbf{wf} \quad \text{and} \quad ss' \vdash_s fs'\ \textbf{live} \quad \text{and} \quad fs' \vdash_e e'\ \textbf{live} \\ & \quad \text{and} \ \cdot \mid sp' \vdash_{vis} e \sqsupseteq e' \dagger sp \\ & \quad \text{and} \ \cdot \mid \cdot \mid se' \mid sp' \vdash_c fs'/x' :: t ; e' \end{array}$$

When a well typed configuration transitions to a new state, then the new configuration has the same type as before, and its effect is visibly subsumed by the effect of the previous configuration.

Proof by rule induction over the derivation of ($ss \mid sp \mid fs \mid x \longrightarrow ss' \mid sp' \mid fs' \mid x'$). The interesting cases are for rules (SfPrivatePush) and (SfPrivatePop) of Fig. 6, which allocate and deallocate regions respectively. The proof of the (SfPrivatePush) case uses Lemma 4.11 to manage the region phase change that occurs when the new region handle is substituted for the corresponding region variable in the expression under evaluation. This process is described in §4.5.1. The proof of the (SfPrivatePop) case uses Lemma 4.9 from to show that well formedness of the store and frame stack is preserved when leaving the body of a `private` expression and deleting the corresponding region. □

	0 with tt | region 0 | priv d 0 | let _ = write (rgn 0) (loc 0) ff in private *r* in let *x* : *Ref r Nat* = alloc *r* 5 in write *r x* 42	*Write* (rgn 0)
$\xrightarrow{*}$	0 with ff | region 0 | priv d 0 | private *r* in let *x* : *Ref r Nat* = alloc *r* 5 in write *r x* 42	⊥
⟶	0 with ff | region 0, region 1 | priv d 0, priv d 1 | let *x* : *Ref* (rgn 1) *Nat* = alloc (rgn 1) 5 in write (rgn 1) *x* 42	*Alloc* (rgn 1) + *Write* (rgn 1)
⟶	0 with ff | region 0, region 1 | priv d 0, priv d 1, (let *x* : ... = ○ in write ...) | alloc (rgn 1) 5	*Alloc* (rgn 1) + *Write* (rgn 1)
⟶	0 with ff, 1 with 5 | region 0, region 1 | priv d 0, priv d 1, (let *x* : ... = ○ in write ...) | loc 1	*Write* (rgn 1)
⟶	0 with ff, 1 with 5 | region 0, region 1 | priv d 0, priv d 1 | write (rgn 1) (loc 1) 42	*Write* (rgn 1)
⟶	0 with ff, 1 with 42 | region 0, region 1 | priv d 0, priv d 1 | unit	⊥
⟶	0 with ff, 1 with • | region 0, region 1 | priv d 0 | unit	⊥

Fig. 14. Example Trace

Theorem 4.2 (Progress).

If $se \mid sp \vdash ss\ ;\ fs$ **wf** and $ss \vdash_s fs$ **live** and $fs \vdash_e e$ **live**
and $\cdot \mid \cdot \mid se \mid sp \vdash_c fs/x :: t\ ;\ e$
then done $fs\ x$
or exists $ss'\ sp'\ fs'\ x'.\ ss \mid sp \mid fs \mid x \longrightarrow ss' \mid sp' \mid fs' \mid x'$

where done $fs\ x \stackrel{\text{def}}{=} (fs = \cdot) \wedge (\text{exists } v.\ x = v)$

A well typed configuration is either done (has finished evaluating) or can transition to a new state. In the above statement the clause (exists $v.\ x = v$) means "x is a value", where the forms of values are defined in Fig. 2.

Proof by induction over the form of x. The interesting cases are when x performs a read or write to the store, as we must show that the corresponding binding has been allocated but not yet deallocated. The proofs of these cases make critical use of liveness information via Lemma. 4.6. We sketch the case for `read` below, with the case for `write` being very similar. For full details see the Coq script.

Case $x = \texttt{read}\ t_1\ v_2$

$$\frac{(5)\ \cdot \mid sp \vdash_t t_1 :: \mathit{Region} \qquad (6)\ \cdot \mid \cdot \mid se \mid sp \vdash_v v_2 :: \mathit{Ref}\ t_1\ t_2}{(4)\ \cdot \mid \cdot \mid se \mid sp \vdash_c fs/\texttt{read}\ t_1\ v_2 :: t_3\ ;\ e_1 \quad \text{(Assume)}}$$

(1..3)	$(se \mid sp \vdash ss\ ;\ fs$ **wf**$)$ $(ss \vdash_s fs$ **live**$)$ $(fs \vdash_e e_1$ **live**$)$	(Assume)
(5, 6)	...	(Invert 4)
(7)	$t_1 = \texttt{rgn}\ p_1$	(Lemma 3.1, 5)
(8)	$v_2 = \texttt{loc}\ l$	(Forms of Values 6)
(9)	$\cdot \mid \cdot \mid se \mid sp \vdash_v \texttt{loc}\ l :: \mathit{Ref}\ t_1\ t_2$	(Substitute 6 8)
(10)	$\mathit{Ref}\ t_1\ t_2 = \text{get}\ l\ se$	(Invert/TyLoc 9)
(11)	$b = \text{get}\ l\ ss$	(StoreM 10 1)
(12)	Case $b = (p_3\ \texttt{with}\ v_3)$	
(13)	$\cdot \mid \cdot \mid se \mid sp \vdash_b p_3\ \texttt{with}\ v_3 :: \mathit{Ref}\ (\texttt{rgn}\ p_1)\ t_2$	(StoreT 12 10 1)
(14)	$p_3 = p_1$	(Invert/TbValue 13)
(15)	$ss \mid sp \mid fs \mid \texttt{read}\ (\texttt{rgn}\ p_1)\ (\texttt{loc}\ l) \longrightarrow ss \mid sp \mid fs \mid v_3$	(SfStoreRead 11 12 14)
(16)	Case $b = (p_3\ \texttt{with}\ \bullet)$	
(17)	$\cdot \mid \cdot \mid se \mid sp \vdash_b p_3\ \texttt{with}\ \bullet :: \mathit{Ref}\ (\texttt{rgn}\ p_1)\ t_2$	(StoreT 16 10 1)
(18)	$p_3 = p_1$	(Invert/TbDead 13)
(19)	$\cdot \mid \cdot \mid se \mid sp \vdash_f fs :: t_2 \multimap t_3\ ;\ e_2$	(Invert/TcExp 4)
(20)	$\cdot \mid \cdot \mid se \mid sp \vdash_x \texttt{read}\ (\texttt{rgn}\ p_1)\ l :: t_2\ ;\ e_3$	...
(21)	$\cdot \mid sp \vdash e_2 + e_3 \equiv e_1 :: \mathit{Effect}$	...
(22)	$e_3 = \mathit{Read}\ (\texttt{rgn}\ p_1)$	(Invert/TxRead 20)
(23)	$\cdot \mid sp \vdash e_1 \sqsupseteq \mathit{Read}\ (\texttt{rgn}\ p_1) :: \mathit{Effect}$	(Prop of $\equiv$ 21 22)
(24)	$fs \vdash_e \mathit{Read}\ (\texttt{rgn}\ p_1)$ **live**	(Prop of LiveE 3 23)
(25)	exists v, $b = (p_1\ \texttt{with}\ v)$	(Lemma 4.6, 2 24 11)
(26)	Contradiction	(16 25)

In (7) type t_1 is a region handle because it is closed (via 5), and the only closed types of region kind are region handles (Lemma 3.1). Similarly, in (8) value v_2 is a location because it is closed (6), and the only closed values of type $\mathit{Ref}\ t_1\ t_2$ are locations.

In (11) the length of the store *ss* is the same as the length of the store environment *se* due to the well formedness condition on stores (1). This means we can get the binding *b* associated with location *l*, but we do not yet know whether it is dead (deallocated) or still live (a value binding).

In the case where store binding *b* is live (12), we know it is in region p_1 as indicated by its type (*Ref* (`rgn` p_1) t_1), because the store is well typed relative to the store typing (1). Region p_1 matches the region handle (`rgn` p_1) in the expression, so the whole configuration can reduce via (SfStoreRead) from Fig. 6.

In the case where the store binding *b* is dead (16) we invert the typing judgment (4) to reveal the effect of the overall configuration e_1. This effect includes the effect of just the `read` expression currently being reduced (*Read* (`rgn` p_1)) (22), as well as the effect of the rest of the computation e_2. Based on the read effect, the liveness statement (3) indicates there must be a corresponding (`priv` *m* p_1) frame on the frame stack *fs*, for some mode *m*. Using Lemma 4.6 and (2) this implies that all bindings in region p_1 must be live. This contradicts the original statement that *b* was dead (16), so this case cannot happen. □

5 Related Work

Banerjee, Heintze and Riecke (1999) prove soundness for a fragment of the region calculus (Tofte & Talpin, 1993) by translation to a target language. The target language is given a denotational semantics, and completeness of the translation shows that the source language is sound. The fragment covered is monomorphic, and does not include mutable references.

Calcagno, Helsen and Thiemann (2000; 2002) give hand written syntactic soundness proofs for several versions of the region calculus (Tofte & Talpin, 1993). The version in (Helsen & Thiemann, 2000) models region deallocation by substituting a special dead region identifier for the associated variable when leaving the scope of a `private` construct. Their dynamic semantics does not include an explicit store, and does not make a phase distinction between compile-time region variables and run-time region handles. This store-less semantics supports a straightforward proof of the region deallocation mechanism, but does not support mutable references. Calcagno *et al.* (2002) extend the previous work with mutable references and a store, but remove polymorphism. As discussed in §1.6, the dynamic semantics of this latter language makes polymorphism difficult to add due to the use of an auxiliary expression to hold allocated region names. The version presented in (Calcagno, 2001) is similar.

Walker, Crary and Morisett (2000) define the Capability Language (CL), which supports region based memory management where the allocation and deallocation points for separate regions can be interleaved. To achieve this, the CL requires programs to be written in continuation passing style (CPS) so that the set of live regions can be tracked at each point in the program. In contrast, languages like System-F^{re} require region lifetimes to be nested, following the lexical scoping of the `private` construct. Walker et al give a hand written syntactic soundness proof for the CL, as well as a type directed translation from Tofte and Talpin's language into CL. Compared to lexically scoped languages, the CPS style CL permits more efficient use of memory for objects whose lifetimes do not follow the lexical structure of the code. However, the lexically scoped version directly supports well known program transformations, which is a key motivation for our current work. For a concrete

compiler implementation it would seem reasonable to use both approaches, basing the front-end language on the lexically scoped representation, but optimizing region lifetimes by converting to the CPS representation during code generation.

Henglein, Makholm and Niss (2001; 2004) present a language related to CL that also allows the allocation and deallocation points for separate regions to be interleaved. Their system does not require the program to be CPS converted, but only supports first order programs. A different language presented by Henglein *et al.* (2004) illustrates the core of a type and effect system by simulating the usual mechanisms that access immutable regions (allocate and read) with ones that simply tag and untag a value with a region identifier. As with the system by Calcagno *et al.* (2002) they prove soundness of the region calculus and translation correctness from the source language separately. They use nu-binding (Pitts & Stark, 1993) as per rule (TyPrivate) from §1.6 to introduce new region identifiers, and their evaluation semantics relies on implicit alpha conversion to avoid variable capture.

Fluet and Morisett (2004; 2006) define a monadic intermediate language, F^{RGN} inspired by the ST monad of Launchbury & Peyton Jones (1994). The monadic system provides a stack of regions, as well as constraints that particular regions must outlive others. They give a soundness argument for the region calculi in terms of translation onto F^{RGN}, and the companion technical report (Fluet, 2004) contains a hand written proof. Although the intermediate language F^{RGN} distills the essential mechanism of region deallocation, there is a large semantic gap between it and the original region calculi, with the translation to F^{RGN} proceeding via two other intermediate languages. The later work by Fluet (2006) gives a translation from F^{RGN} to an even lower level language with linearly typed regions. Kiselyov & Shan (2008) provide an embedding in Haskell.

Papakyriakou, Gerakios and Papaspyrou (2007) provide a syntactic soundness proof for a version of System-F with mutable references, but without regions. The proof was mechanized in Isabelle/HOL (Nipkow *et al.*, 2002).

Boudol (2008) defines a monomorphic variant of the region calculus that allows regions to be deallocated earlier than they would be using the stack discipline. The type system includes *negative* deallocation effects, and ensures that the usual *positive* effects such as reads and writes are not performed after negative effects to the same region. This work includes a hand written soundness proof based on "subject reduction up-to-simulation".

Montenegro, Pena, Segura and Dios (2008; 2010) present a first order functional language with nested regions. The language includes a `case` expression that deallocates its argument while it destructing it, and the type system tracks which arguments of functions will be deallocated during evaluation. Later work by de Dios *et al.* (2010) describes a soundness proof in Isabelle/HOL, as well as a method for generating certificates that prove type checked programs do not contain dangling pointers.

Pottier (2013) presents a mechanized syntactic soundness proof for a type and capability system with hidden state. Pottier's system uses affine capabilities and includes region adoption and focusing (Fahndrich & DeLine, 2002), and the anti-frame rule (Pottier, 2008) for hiding local state. Pottier's system assumes garbage collection, and does not include region deallocation. Regions are used to reason about the aliasing and separation properties of data, but not as memory management discipline. In contrast, the regions in our system are intended primarily for memory management, and our type system does not encode the more advanced separation properties.

6 Conclusion

Writing about a proof is like dancing about architecture.[1] We intend this paper to be a summary of the language System-F^{re}, presented in a way that can be mechanically proved sound, while leaving most of the low level details in the proof script where they belong.

This work was carried out at a time of increasing community interest in producing mechanically verified languages and compiler implementations. The influential CompCert project (Leroy, 2009) has yielded a verified compiler for the C language, and we are beginning to see results for higher level functional languages (Chlipala, 2007; Chlipala, 2010). As opposed to C, implementations for functional languages implicitly depend on a runtime system to manage storage allocation, and verifying such a system requires a large investment of effort on its own (Hawblitzel & Petrank, 2010). One advantage that System-F^{re} has in this space is that the language semantics naturally includes storage management, which (we hope) will reduce the overall cost of verifying its implementation.

Acknowledgements Thanks to Peter Gammie, Fritz Henglein and Amos Robinson for helpful feedback on draft versions of this paper. This work was supported in part by the Australian Research Council under grant number LP0989507.

References

Banerjee, Anindya, Heintze, Nevin, & Riecke, Jon G. (1999). Region analysis and the polymorphic lambda calculus. *Logic in computer science*. IEEE.

Boudol, Gérard. (2008). Typing safe deallocation. *ESOP: European Symposium on Programming*.

Calcagno, Cristiano. (2001). Stratified operational semantics for safety and correctness of the region calculus. *POPL: Principles of Programming Languages*. ACM Press.

Calcagno, Cristiano, Helsen, Simon, & Thiemann, Peter. (2002). Syntactic type soundness results for the region calculus. *Information and computation*, **173**(2).

Chlipala, Adam. (2007). A certified type-preserving compiler from lambda calculus to assembly language. *PLDI: Programming language design and implementation*. ACM.

Chlipala, Adam. (2010). A verified compiler for an impure functional language. *POPL: Principles of Programming Languages*. ACM.

de Dios, Javier, Montenegro, Manuel, & Peña, Ricardo. (2010). Certified absence of dangling pointers in a language with explicit deallocation. *IFM: Integrated Formal Methods*.

Fahndrich, Manuel, & DeLine, Robert. (2002). Adoption and focus: practical linear types for imperative programming. *SIGPLAN Notices*.

Fluet, Matthew. (2004). *Monadic regions: Formal type soudness and correctness*. Tech. rept. Cornell University.

Fluet, Matthew, & Morrisett, Greg. (2004). Monadic regions. *ICFP: International Conference on Functional Programming*.

Fluet, Matthew, Morrisett, Greg, & Ahmed, Amal J. (2006). Linear regions are all you need. *ESOP: European Symposium on Programming*.

Garrigue, Jacques. (2002). Relaxing the value restriction. *APLAS: Asian Workshop on Programming Languages and Systems*.

[1] After a quote about music by Frank Zappa, reimagined by Peter Gammie.

Hawblitzel, Chris, & Petrank, Erez. (2010). Automated verification of practical garbage collectors. *Logical methods in computer science*, **6**(3).

Helsen, Simon, & Thiemann, Peter. (2000). Syntactic type soundness for the region calculus. *Electronic notes on theoretical computer science*, **41**(3).

Henglein, Fritz, Makholm, Henning, & Niss, Henning. (2001). A direct approach to control-flow sensitive region-based memory management. *PPDP: Principles and Practice of Declarative Programming*.

Henglein, Fritz, Makholm, Henning, & Niss, Henning. (2004). *Advanced topics in types and programming languages*. The MIT Press. Chap. Effect Types and Region-Based Memory Management.

Kiselyov, Oleg, & chieh Shan, Chung. (2008). Lightweight monadic regions. *Haskell symposium*.

Launchbury, John, & Peyton Jones, Simon L. (1994). Lazy functional state threads. *PLDI: Programing Language Design and Implementation*.

Leroy, Xavier. (1993). Polymorphism by name for references and continuations. *POPL: Principles of Programming Languages*.

Leroy, Xavier. (2009). Formal verification of a realistic compiler. *Communications of the acm*, **52**(7), 107–115.

Lucassen, John M. (1987). *Types and effects: Towards the integration of functional and imperative programming*. Tech. rept. Laboratory for Computer Science, Massachusetts Institute of Technology.

Lucassen, John M., & Gifford, David K. (1988). Polymorphic effect systems. *POPL: Principles of Programming Languages*.

Montenegro, Manuel, Pena, Ricardo, & Segura, Clara. (2008). A type system for safe memory management and its proof of correctness. *PPDP: Principles and Practice of Declarative Programming*.

Nipkow, Tobias, Wenzel, Markus, & Paulson, Lawrence C. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*.

Papakyriakou, Michalis A., Gerakios, Prodromos E., & Papaspyrou, Nikolaos S. (2007). A mechanized proof of type safety for the polymorphic -calculus with references. *Panhellenic logic symposium*.

Peyton Jones, Simon L., & Santos, Andr L.M. (1998). A transformation-based optimiser for haskell. *Science of computer programming*. Elsevier North-Holland, Inc.

Pitts, Andrew M., & Stark, Ian D. B. (1993). Observable properties of higher order functions that dynamically create local names, or what's new? *Mathematical foundations of computer science*. Springer-Verlag.

Pottier, François. (2008). Hiding local state in direct style: A higher-order anti-frame rule. *LICS: Logic in Computer Science*.

Pottier, François. (2013). Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of functional programming*, **23**(1).

Reynolds, John C. (2002). Separation logic: A logic for shared mutable data structures. *LICS: Logic in Computer Science*.

Smith, Frederick, Walker, David, & Morrisett, Greg. (2000). Alias types. *ESOP: European Symposium on Programming*.

Talpin, Jean-Pierre, & Jouvelot, Pierre. (1994). The type and effect discipline. *Information and computation*, **111**(2).

Tofte, Mads, & Talpin, Jean-Pierre. (1993). *A theory of stack allocation in polymorphically typed languages*. Tech. rept. 93/15. Department of Computer Science, Copenhagen University.

Walker, David, Crary, Karl, & Morrisett, Greg. (2000). Typed memory management via static capabilities. *TOPLAS: Transactions on Programing Languages and Systems*, **22**(4).