Monadic Regions[∗]

MATTHEW FLUET
Cornell University, Department of Computer Science, Ithaca, NY 14853, USA
(*e-mail:* `fluet@cs.cornell.edu`)

GREG MORRISETT
Harvard University, Division of Engineering and Applied Science, Cambridge, MA 02138, USA
(*e-mail:* `greg@eecs.harvard.edu`)

Abstract

Region-based type systems provide programmer control over memory management without sacrificing type-safety. However, the type systems for region-based languages, such as the ML-Kit or Cyclone, are relatively complicated, and proving their soundness is non-trivial. This paper shows that the complication is in principle unnecessary. In particular, we show that plain old parametric polymorphism, as found in Haskell, is all that is needed. We substantiate this claim by giving a type- and meaning-preserving translation from a variation of the region calculus of Tofte and Talpin to a monadic variant of System F with region primitives whose types and operations are inspired by (and generalize) the ST monad of Launchbury and Peyton Jones.

Capsule Review

Languages with regions are normally specified using a type-and-effects system to ensure safety. Because these formulations involve sets of regions and subset checks, it is not obvious that types for a region calculus can be faithfully encoded into the familiar polymorphism of System F. In this paper, the authors provide just such a translation.

The key steps are to view Haskell's state transformers as computations on independent regions, to add extra function arguments that witness the "outlives" relation between regions, and to observe that if regions have properly nested lifetimes then any set of coexisting regions contains one that outlives the others.

The ideas are quite interesting, and Sections 1–3 do a very good job of providing intuition. The formal translation from regions to monads that follows is technically very involved, however. Nevertheless, the monadic account of regions is deserving of further study.

1 Background

Tofte and Talpin introduced a new technique for type-safe memory management based on *regions* (Tofte & Talpin, 1994; Tofte & Talpin, 1997). In their calculus,

∗ This is a revised and extended version of the paper that appeared in the *Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP'04)* (Fluet & Morrisett, 2004).

regions are areas of memory holding heap allocated data. Regions are introduced and eliminated with a lexically-scoped construct:

$$\mathsf{letregion}\ \rho\ \mathsf{in}\ e$$

and thus have last-in-first-out (LIFO) lifetimes following the block structure of the program. In the example above, a region corresponding to ρ is created upon entering the expression; for the duration of the expression, data can be allocated into the region; after evaluating e to a value, all of the data allocated within the region are reclaimed and the value is returned. The operations for memory management (create region, reclaim region, and allocate object in region) can be implemented in constant time and thus regions provide a compelling alternative to garbage collection.

The key contribution of Tofte and Talpin's framework (hereafter referred to as TT) was a type-and-effects system that ensures the safety of this allocation and deallocation scheme. The types of allocated data objects are augmented with the region in which they live. For example the type:

$$((\mathsf{int}, \rho_1) \times (\mathsf{int}, \rho_2), \rho_1)$$

describes pairs of integers where the pair and first component live in region ρ_1 and the second component lives in region ρ_2.

Region polymorphism makes it possible to abstract over the regions a computation manipulates. Furthermore, function types include an *effect* which records the set of regions that must still be allocated in order to ensure that the computation is safe to run. In general, any operation that needs to dereference a pointer into a region will require that region to be live. For example, a function `fst` that takes in a pair of integers and returns the first component without examining it could have a type of the form:

$$\mathtt{fst} :: \forall \rho_1, \rho_2, \rho_3.((\mathsf{int}, \rho_1) \times (\mathsf{int}, \rho_2), \rho_3) \xrightarrow{\{\rho_3\}} (\mathsf{int}, \rho_1)$$

Such a function is polymorphic over regions ρ_1, ρ_2, and ρ_3 so the caller can effectively re-use the function regardless of where the data were allocated. However, the effect "$\{\rho_3\}$" on the arrow indicates that whatever region instantiates ρ_3 needs to still be allocated when `fst` is called. In principle, neither of the other regions needs to be live across the call since the function does not examine the integer values. In practice, ρ_1 will be live assuming the caller wishes to use the result.

A unique feature of this scheme is that evaluation can lead to values with *dangling pointers*: a pointer to data in some region that has been reclaimed. This is allowed when the effect of the continuation does not contain the dangling pointer's region, for then we know the computation never dereferences the dangling pointer. For some programs, this allows a region-based memory manager to reclaim strictly more objects than a trace-based garbage collector. Consider, for example, the following program:

```
letregion $\rho_a$ in
  let g = letregion $\rho_b$ in
            let p = $(3$ at $\rho_a, 4$ at $\rho_b)$ at $\rho_a$
            in $\lambda z$:unit. fst $[\rho_a, \rho_b, \rho_a]$ p
  in g ()
```

The pair p and its first component are allocated in the outer (older) region ρ_a whereas p's second component is allocated in an inner (younger) region ρ_b. The closure bound to g is a thunk that calls fst on p. Note that the region ρ_b is deallocated before the thunk is run, and thus g's closure contains a dangling pointer to an object that is never dereferenced. The TT system is strong enough to show that the code is safe.

Variations on the TT typing discipline have been used in a number of projects. The ML-Kit compiler (Tofte *et al.*, 2002) uses automatic region inference to translate Standard ML into a region-based language instead of relying upon traditional garbage collection. In contrast, the Cyclone Safe-C language (Grossman *et al.*, 2002) exposes regions and region allocation to the programmer. Furthermore, the type-and-effects system of Cyclone extends that of TT with a form of region subtyping—pointers into older regions can be safely treated as pointers into younger regions. This extra degree of polymorphism is crucial for minimizing the lifetimes of objects, as they would otherwise be constrained to live in the same region. Region subtyping also simplifies the interfaces for abstract datatypes because the regions of the ADT can be encapsulated by an upper bound.

Unfortunately, the type-and-effect systems of TT and Cyclone are relatively complicated. At the type level, they introduce new kinds for regions and effects. Effects are meant to be treated as sets of regions, so standard term equality no longer suffices for type checking. Finally, the typing rule for letregion is extremely subtle because of the interplay of dangling pointers and effects. Indeed, over the past few years, a number of papers have been published attempting to simplify or at least clarify the soundness of the construct (Crary *et al.*, 1999; Banerjee *et al.*, 1999; Helsen & Thiemann, 2000; Calcagno, 2001; Calcagno *et al.*, 2002; Henglein *et al.*, 2005). All of these problems are amplified in Cyclone because of region subtyping where the meta-theory is considerably more complicated (Grossman *et al.*, 2001).

Overview

The goal of this work is to find a simpler account of region-based type systems. In particular, we wish to explain the type soundness of a Cyclone-like language via translation to a language with only parametric polymorphism, such as System F (Reynolds, 1974; Girard *et al.*, 1989) extended with appropriate region primitives. The essence of the translation is that parametric polymorphism alone provides the power needed to model region effects.

Our work was inspired by the ST monad of Launchbury and Peyton Jones (1995; 1994) which is used to encapsulate a "stateful" computation within a pure functional language such as Haskell. Indeed, the runST primitive turns out to be a good approximation of letregion in that it creates a new store, allows one to allocate values in the store, and upon completion, deallocates the store and returns a value that may have dangling pointers. The runST primitive can be assigned a conventional polymorphic type, which, through the magic of parametricity, ensures that dangling pointers are never dereferenced. Unfortunately, runST is not sufficient to encode region-based languages since there is no support for nested stores. In particular, a nested application of runST cannot allocate or touch data in an outer store. An extension to ST that admits a limited form of nested stores was proposed by Launchbury and Sabry (1997) but, as we discuss in Section 2, it does not provide enough flexibility to encode the region polymorphism of TT or Cyclone.

In this paper, we consider a monad family, called RGN, which does provide the necessary power to encode region calculi and back this claim by giving a translation from a novel region calculus to a monadic version of System F which we call $\mathsf{F}^{\mathsf{RGN}}$. The central element of the translation is the presence of terms that witness region subtyping. These terms provide the evidence needed to safely "shift" computations from one store to another. We believe that this translation sheds new light on both region calculi as well as Haskell's ST monad. In particular, it shows that the notion of region subtyping is in some sense central for supporting nested stores.

The remainder of this paper is structured as follows. In the following section, we examine more closely why the ST monad and its variants are insufficient for encoding region-based languages. This motivates the design for $\mathsf{F}^{\mathsf{RGN}}$, which is presented more formally in Section 3. A key aspect of $\mathsf{F}^{\mathsf{RGN}}$ is that no extension to the type system of System F is required. We give dynamic and static semantics for the language, and prove the soundness of the type system. Encapsulation of region computations in $\mathsf{F}^{\mathsf{RGN}}$ is ensured by the type system, using parametric polymorphism. We feel that Sections 2 and 3 develop sufficient intuition to reasonably establish our goal of finding a simpler account of region-based type systems.

However, the skeptical reader may well wonder if the simplicity of the $\mathsf{F}^{\mathsf{RGN}}$ type system points to some deficiency, failing to capture all of the idioms available in type-and-effect systems for region calculi. In the interest of rigorously demonstrating that we have lost no power in adopting $\mathsf{F}^{\mathsf{RGN}}$, Section 4 returns to region calculi, developing a source language that captures the key aspects of TT and Cyclone-like region calculi. Then, in Section 5, we show how this language can be translated to $\mathsf{F}^{\mathsf{RGN}}$ in a type- and meaning-preserving fashion, thereby establishing our claim that parametric polymorphism is sufficient for encoding the type-and-effects systems of region calculi. Much of the technical complexity stems from our desire to establish the *correctness* of the translation, in addition to the simpler property of type preservation. Section 6 considers the expressiveness of our region calculi. Some readers may prefer to skip Sections 4–6 on a first reading.

In Sections 7 and 8, we consider related work and summarize and note directions for future work. Due to space limitations, the proofs of the theorems stated here can be found in a companion technical report (Fluet, 2004).

2 From ST to RGN

Launchbury and Peyton Jones (1995; 1994) introduced the ST monad to encapsulate stateful computations within the pure functional language Haskell. Three key insights give rise to a safe and efficient implementation of stateful computations. First, a stateful computation is represented as a *store transformer*, a description of commands to be applied to an initial store to yield a final store. Second, the store can not be duplicated, because the state type is opaque and all primitive store transformers use the store in a single-threaded manner; hence, a stateful computation can update the store in place. Third, parametric polymorphism can be used to safely encapsulate and run a stateful computation.

The types and operations associated with the ST monad are the following:

$$\tau \quad ::= \quad \ldots \mid \mathsf{ST}\ \tau_s\ \tau_a \mid \mathsf{STRef}\ \tau_s\ \tau_a$$

$$\begin{array}{rcl}
\mathsf{returnST} & :: & \forall s.\forall a.a \to \mathsf{ST}\ s\ a \\
\mathsf{thenST} & :: & \forall s.\forall a, b.\mathsf{ST}\ s\ a \to (a \to \mathsf{ST}\ s\ b) \to \mathsf{ST}\ s\ b \\
\mathsf{newSTRef} & :: & \forall s.\forall a.a \to \mathsf{ST}\ s\ (\mathsf{STRef}\ s\ a) \\
\mathsf{readSTRef} & :: & \forall s.\forall a.\mathsf{STRef}\ s\ a \to \mathsf{ST}\ s\ a \\
\mathsf{writeSTRef} & :: & \forall s.\forall a.\mathsf{STRef}\ s\ a \to a \to \mathsf{ST}\ s\ () \\
\mathsf{runST} & :: & \forall a.(\forall s.\mathsf{ST}\ s\ a) \to a
\end{array}$$

The type ST s a is the type of computations which operate on a store and deliver a value of type a. The type s behaves like a name (or index) for the store and serves to distinguish computations operating on one store from computations operating on another. The type STRef s a is the type of references allocated from a store indexed by s and containing a value of type a.

The operations returnST and thenST are the *unit* and *bind* operations of the ST monad. The former yields the trivial store transformer that delivers its argument without affecting the store. The latter composes store transformers in sequence, passing the result and final store of the first computation to the second; notice that the two computations must manipulate stores indexed by the same type.

The next three operations are primitive store transformers that operate on the store. newSTRef takes an initial value and yields a store transformer, which, when applied to a store, allocates a fresh reference, and delivers the reference and the store augmented with the reference mapping to the initial value. Similarly, readSTRef and writeSTRef yield computations that respectively query and update the mappings of references to values in the current store. Note that all of these operations require the store index types of ST and STRef to be equal.

In this section, we will write short code examples in pseudo-Haskell syntax[1] using the `do` notation, which provides a more conventional syntax for monadic programming. This notation allows `do` x `<-` e`;` *stmts* as shorthand for `thenST` e (λx`.` `do` *stmts*) and `do` e as shorthand for e.

Here is a function yielding a computation that adds the contents of two references into a new reference:

```
add :: ∀s. STRef s Int → STRef s Int → ST s Int
add v w = do a <- readSTRef v
             b <- readSTRef w
             newSTRef (a + b)
```

Finally, the operation runST encapsulates a stateful computation. To do so, it takes a store transformer as its argument, applies it to an initial empty store, and returns the result while discarding the final store. Note that to apply runST,

[1] We use Haskell as a convenient and familiar notation, but the correspondence is quite weak. In particular, all of the calculi presented in the remainder of this paper will evaluate under a call-by-value semantics.

we instantiate a with the type of the result to be returned, and then supply a store transformer, *which is polymorphic in the store index type.* The effect of this universal quantification is that the store transformer makes no assumptions about the initial store (e.g., the existence of pre-allocated references). Furthermore, the instantiation of the type variable a occurs outside the scope of the type variable s; this prevents the store transformer from delivering a value whose type mentions s. Thus, references or computations depending on the final store cannot escape beyond the encapsulation of runST.

All of these observations can be carried over to the region case, where we interpret stores as regions. We introduce the type RGN r a as the type of computations which transform a region indexed by r and deliver a value of type a. Likewise, the type RGNLoc r a is the type of (immutable) locations allocated in a region indexed by r and containing a value of type a. Each of the operations in the ST monad has an isomorphic analogue in the RGN monad:

$$\begin{aligned}
\mathsf{returnRGN} &:: \forall r.\forall a.a \rightarrow \mathsf{RGN}\ r\ a\\
\mathsf{thenRGN} &:: \forall r.\forall a,b.\mathsf{RGN}\ r\ a \rightarrow (a \rightarrow \mathsf{RGN}\ r\ b) \rightarrow \mathsf{RGN}\ r\ b\\
\mathsf{newRGNLoc} &:: \forall r.\forall a.a \rightarrow \mathsf{RGN}\ r\ (\mathsf{RGNLoc}\ r\ a)\\
\mathsf{readRGNLoc} &:: \forall r.\forall a.\mathsf{RGNLoc}\ r\ a \rightarrow \mathsf{RGN}\ r\ a\\
\mathsf{runRGN} &:: \forall a.(\forall r.\mathsf{RGN}\ r\ a) \rightarrow a
\end{aligned}$$

Does this suffice to encode region-based languages, where runRGN corresponds to letregion? In short, it does not. In a region-based language, it is critical to allocate locations in and read locations from an outer region while in the scope of an inner region. For example, an essential idiom in region-based languages is to enter a letregion in which temporary data is allocated, while reading input from and allocating output in an outer region; upon leaving the letregion, the temporary data is reclaimed, but the input and output data are still available.

Unfortunately, this idiom cannot be accommodated in the framework presented thus far. For example, consider this canonical example of region-based memory management usage:

```
letregion $\rho_1$ in
  let $a = 1$ at $\rho_1$ in
  let $c =$ letregion $\rho_2$ in
              let $b = 7$ at $\rho_2$ in
              let $z = (a \oplus b)$ at $\rho_1$ in
              $z$ in
  $\ldots c \ldots$
```

where we think of a as an input, b as a temporary, and c as an output. A naïve translation fails to type-check:

```
runRGN ($\Lambda r_1$.
  do a <- newRGNLoc [$r_1$] 1
     c <- runRGN ($\Lambda r_2$.
            do b <- newRGNLoc [$r_2$] 7
               z <- a $\oplus$ b
               newRGNLoc [$r_1$] z)
     ... c ...)
```

The error arises from the fact that allocating a temporary in the younger region (newRGNLoc [r_2] 7) yields a computation of type RGN r_2, while allocating the result in the older region (newRGNLoc [r_1] z) yields a computation of type RGN r_1. These computations cannot be sequenced, since their region indices differ.

Launchbury and Sabry (1997) argue that the principle behind runST can be generalized to provide nested scope. They introduce two additional operations

$$\begin{aligned} \mathsf{blockST} &:: \forall s.\forall a.(\forall r.\mathsf{ST}\ (s,r)\ a) \rightarrow \mathsf{ST}\ s\ a \\ \mathsf{importSTRef} &:: \forall s.\forall r.\forall a.\mathsf{STRef}\ s\ a \rightarrow \mathsf{STRef}\ (s,r)\ a \end{aligned}$$

where blockST encapsulates a nested scope and importSTRef explicitly allows references from an enclosing scope to be manipulated by the inner scope. Similarly, Peyton Jones[2] suggests introducing the constant

$$\mathsf{liftST} :: \forall s.\forall r.\forall a.\mathsf{ST}\ s\ a \rightarrow \mathsf{ST}\ (s,r)\ a$$

in lieu of importSTRef, with the same intention of importing computations from an outer scope into the inner scope. In essence, liftST encodes the stack of stores using a tuple type for the index on the ST monad.

Should we adopt blockST and liftST in the RGN monad as letRGN and liftRGN? At first glance, doing so would appear to provide sufficient expressiveness to encode region-based languages. We can "fix" our previous translation as follows:

```
runRGN (Λr₁.
   do a <- newRGNLoc [r₁] 1
      c <- letRGN (Λr₂.
              do b <- newRGNLoc [r₂] 7
                 z <- a ⊕ b
                 liftRGN (newRGNLoc [r₁] z))
   ... c ...)
```

However, another critical aspect of region-based languages is region polymorphism. For example, consider a generalization of the add function above, where each of the two input locations are allocated in different regions, the output location is to be allocated in a third region, and the result computation is to be indexed by a fourth region; such a function would have the type:

```
gadd :: ∀r₁,r₂,r₃,r₄.
         RGNLoc r₁ Int → RGNLoc r₂ Int → RGN r₄ (RGNLoc r₃ Int)
```

However, there is no way to write gadd with liftRGN terms that will result in sufficient polymorphism over regions. For example, if we write

```
gadd v w = liftRGN  (do a <- readRGNLoc v
                        b <- liftRGN (readRGNLoc w)
                        liftRGN (liftRGN (newRGNLoc (a + b))))
```

then we produce a function with the type:

```
gadd :: ∀r₁,r₂,r₃,r₄.
         RGNLoc ((r₁, r₂), r₃) Int → RGNLoc (r₁, r₂) Int →
         RGN (((r₁, r₂), r₃), r₄) (RGNLoc r₁ Int)
```

The problem is that the explicit connection between the outer and inner regions in the product type enforces a total order on regions, which leaks into the types of region allocated values. The function above *only* works when the four regions are consecutive and the output location is allocated in the outermost region, the input

[2] private communication

locations are allocated in the next two regions, and the computation is indexed by the innermost region.

However, the order of the regions should not matter. The only requirement is that if the final computation (indexed by r_4) is ever run, then each of the regions r_1, r_2, and r_3 must be live. To put it another way, the three regions are older than (i.e., subtypes of) region r_4. Hence, we adopt a simple solution, one that enables the translation given in Section 5, whereby we abstract the liftRGN applications and pass evidence that witnesses the region subtyping.

```
gadd :: ∀r1,r2,r3,r4.
          (∀b.RGN r1 b → RGN r4 b) →
          (∀b.RGN r2 b → RGN r4 b) →
          (∀b.RGN r3 b → RGN r4 b) →
          RGNLoc r1 Int → RGNLoc r2 Int →
          RGN r4 (RGNLoc r3 Int)
gadd ev1 ev2 ev3 v w = do a <- ev1 (readRGNLoc v)
                          b <- ev2 (readRGNLoc w)
                          ev3 (newRGNLoc (a + b))
```

While this evidence can be assembled from liftRGN terms, we find that the key notion is subtyping on regions and evidence that witnesses the subtyping. The product type used in blockST is one way of connecting the outer and inner stores, but all the "magic" happens with liftST. Therefore, we adopt an approach that fuses the two operations into letRGN:

$$\begin{array}{lcl} r_1 \preceq r_2 & \equiv & \forall b.\mathsf{RGN}\ r_1\ b \rightarrow \mathsf{RGN}\ r_2\ b \\ \mathsf{letRGN} & :: & \forall r_1.\forall a.(\forall r_2.r_1 \preceq r_2 \rightarrow \mathsf{RGN}\ r_2\ a) \rightarrow \mathsf{RGN}\ r_1\ a \end{array}$$

The argument to letRGN is given the evidence that the outer (older) region is a subtype of the new (younger) region, which it can use in the region computation. The same parametricity argument that applied to runST applies here: locations and computations from the inner region cannot escape beyond the encapsulation of letRGN. We no longer need a product type connecting the outer and inner regions, as this relationship is given by the witness function.

We note that much of the development in this paper could be pursued using letRGN and liftRGN with types analogous to blockST and liftST (i.e., using a product type) and appropriately assembling evidence from liftRGN terms. However, we have adopted the approach given above for a number of reasons. First, the types are smaller than under the alternative scheme. Looking forward to Section 5, we trade the number of terms in scope for the size of the types in scope. Second, one is encouraged to write region polymorphic functions with the fused letRGN, whereas one can write region constrained functions with liftRGN. Third, letRGN makes it clear that the only witness functions are those that arise from entering a new region. Finally, although we have made the type $r_1 \preceq r_2$ a synonym for a witness function, we can imagine a scheme in which this primitive evidence is abstract and we provide additional operations for combining evidence and operations for taking evidence to functions for importing RGN computations or RGNLoc locations. The latter corresponds to pointer subtyping in Cyclone, where a pointer to region r_1 may be coerced to a pointer to region r_2 when r_1 outlives r_2.

3 Target Language: $\mathsf{F}^{\mathsf{RGN}}$

The language $\mathsf{F}^{\mathsf{RGN}}$ is an extension of System F (Reynolds, 1974; Girard *et al.*, 1989) (also referred to as the polymorphic lambda calculus), adding the types and operations from the RGN monad. As described in the previous section, the design of $\mathsf{F}^{\mathsf{RGN}}$ takes inspiration from the work on monadic state (Launchbury & Peyton Jones, 1995; Launchbury & Peyton Jones, 1994; Launchbury & Sabry, 1997; Ariola & Sabry, 1998; Semmelroth & Sabry, 1999; Moggi & Sabry, 2001). Essentially, $\mathsf{F}^{\mathsf{RGN}}$ uses an explicit region monad to enforce the locality of region allocated values.

We present the full formal language $\mathsf{F}^{\mathsf{RGN}}$ and (sketch) a syntactic proof of type soundness. We begin with a presentation of the language (surface syntax, computation syntax, dynamic semantics, and static semantics) and then proceed to the proof.

The dynamic semantics defines a large-step (or natural) semantics, which defines an *evaluation relation* from *towers of stacks of regions* and *expressions* to *values*. Our main reason for adopting a large-step operational semantics is to simplify the theorems and proofs of Section 5; establishing the correctness of the translation would be more difficult using small-step operational semantics, due to differing numbers of intermediate small-steps.

To prove type soundness, we adopt a proof method using *natural transition semantics* (Volpano & Smith, 1997; Smith & Volpano, 1998), which models program execution in terms of transitions between *partial derivations*. Although the language presented here can only describe terminating programs, this proof method can be extended in a straight-forward manner to handle non-terminating executions, as will arise from adding a fixRGNLoc command. Adopting this method allows us to prove type soundness with the familiar progress and preservation theorems, without needing define a small-step operational semantics nor establish its correspondence with the large-step operational semantics we wish to use in Section 5. We remark further on this proof method at the end of Section 3.5.

3.1 Syntax of $\mathsf{F}^{\mathsf{RGN}}$

Figure 1 presents the syntax of "surface programs" (that is, excluding syntax and semantic values that will appear in the operational semantics) of $\mathsf{F}^{\mathsf{RGN}}$. In the following sections, we explain and motivate the main constructs of $\mathsf{F}^{\mathsf{RGN}}$.

3.1.1 Types

Types in $\mathsf{F}^{\mathsf{RGN}}$ include all those found in System F (function and product types and type abstractions) along with the primitive types int and bool. The RGN τ_r τ_a and RGNLoc τ_r τ_a types were introduced in the previous section. We add the type RGNHandle τ_r as the type of handles for the region indexed by the type τ_r. A value of this type is a *region handle* – a run-time value holding the data necessary to allocate values within a region. Region indices (types) and region handles (values) are distinguished in order to maintain a phase distinction between compile-time and run-time expressions and to more accurately reflect implementations of regions. The

$$\begin{array}{rcl}
i & \in & \mathbb{Z} \\
\alpha,\beta,\gamma & \in & \mathit{TVars} \\
f,x & \in & \mathit{Vars}
\end{array}$$

Surface types

$$\begin{array}{rcl}
\tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \alpha \mid \forall\alpha.\tau \mid \\
 & & \mathsf{RGN}\ \tau_r\ \tau_a \mid \mathsf{RGNLoc}\ \tau_r\ \tau_a \mid \mathsf{RGNHandle}\ \tau_r
\end{array}$$

Surface terms

$$\begin{array}{rcl}
e & ::= & i \mid e_1 \oplus e_2 \mid e_1 \circleddash e_3 \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \mid \\
 & & x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid (e_1,\ldots,e_n) \mid \mathsf{sel}_i\ e \mid \Lambda\alpha.e \mid e\ [\tau] \mid \\
 & & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \kappa_r \mid \kappa
\end{array}$$

Surface commands

$$\begin{array}{rcl}
\kappa_r & ::= & \mathsf{runRGN}\ [\tau_a]\ v \\
\kappa & ::= & \mathsf{returnRGN}\ [\tau_r]\ [\tau_a]\ v \mid \mathsf{thenRGN}\ [\tau_r]\ [\tau_a]\ [\tau_b]\ v_1\ v_2 \mid \\
 & & \mathsf{letRGN}\ [\tau_r]\ [\tau_a]\ v \mid \mathsf{newRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \mid \mathsf{readRGNLoc}\ [\tau_r]\ [\tau_a]\ v
\end{array}$$

Surface values

$$\begin{array}{rcl}
v & ::= & i \mid \mathsf{tt} \mid \mathsf{ff} \mid x \mid \lambda x{:}\tau.e \mid (v_1,\ldots,v_n) \mid \Lambda\alpha.e \mid \kappa_r \mid \kappa
\end{array}$$

Fig. 1. Surface syntax of $\mathsf{F}^{\mathsf{RGN}}$

original Tofte-Talpin region calculus (1994; 1997) distinguished between put and get effects; a put entails allocating in a region (and requires a handle), while a get entails reading through a pointer (and does not require a handle). This ensures that region indices, like other types, have no run-time significance and may be erased from compiled code. On the other hand, region handles are necessary at run-time to allocate values within a region.

Region indices could be introduced as a distinct kind, but doing so unnecessarily complicates the type system. Hence, we allow an arbitrary type in the first argument of the RGN monad type constructor. While the type RGN bool int is well-formed, a value of such type cannot arise during the execution of a well-typed program. Furthermore, surface programs will never require a region index to be represented by anything other than a type variable.

3.1.2 Terms

As with types, terms in $\mathsf{F}^{\mathsf{RGN}}$ include all those found in System F; constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and elimination, and type abstraction and instantiation are all completely standard.

We let κ_r and κ range over the syntactic class of monadic commands. (Equivalently, and as suggested by the explicit type annotations and the restriction of subexpressions to values, we can consider the monadic commands as constants with polymorphic types in a call-by-value interpretation of $\mathsf{F}^{\mathsf{RGN}}$. Presenting monadic commands in this fashion avoids intermediate terms in the operational semantics corresponding to partial application.) Each of the commands has been described previously.

	l	$\in$	*Locations*
Region names	r	$\in$	*Rnames*
Region placeholders	ρ	$::=$	$r \mid \bullet$
Stack names	s	$\in$	*Snames*
Stack placeholders	σ	$::=$	$s \mid \circ$
Computation types	τ	$::=$	$\ldots \mid \sigma\sharp\rho$
Computation terms	e	$::=$	$\ldots \mid \langle l \rangle_{\sigma\sharp\rho} \mid \mathsf{handle}(\sigma\sharp\rho)$
Computation commands	κ	$::=$	$\ldots \mid \mathsf{witnessRGN}\ \sigma\sharp\rho_1\ \sigma\sharp\rho_2\ [\tau_a]\ v$
Computation values	v	$::=$	$\ldots \mid \langle l \rangle_{\sigma\sharp\rho} \mid \mathsf{handle}(\sigma\sharp\rho)$
Regions	R	$::=$	$\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$
Stacks	S	$::=$	$\cdot \mid S, r \mapsto R$ (ordered domain)
Towers	T	$::=$	$\cdot \mid T, s \mapsto S$ (ordered domain)

Fig. 2. Computation syntax of $\mathsf{F}^{\mathsf{RGN}}$

3.1.3 Computation Syntax of $\mathsf{F}^{\mathsf{RGN}}$

Figure 2 presents the syntax of "computation programs," which extends the syntax of the previous section with semantic values that appear in the operational semantics.

Stack names, region names, and locations are used to represent pointers to region allocated data. Because runRGN computations can be nested, we need a means to distinguish data allocated in regions that belong to different runRGN computations; stack names serve this purpose. Each runRGN computation is associated with a unique stack, which collects and identifies all regions belonging to that computation. Stack and region placeholders distinguish between live and dead stacks and regions; a dead stack ($\circ$) or region ($\bullet$) corresponds to a deallocated stack or region.

The computation syntax adds one new type form and two new expression forms. The type $\sigma\sharp\rho$ is the instantiated form of a region index (hence, $\circ\sharp\bullet$ corresponds to a dead region in a dead stack). Such a type identifies the stack and region in which a monadic region computation is executing. The expression $\langle l \rangle_{\sigma\sharp\rho}$ is the runtime representation of a RGNLoc $\sigma\sharp\rho\ \tau_a$; that is, it is the pointer associated with a region allocated value. Likewise, the expression handle($\sigma\sharp\rho$) is the runtime representation of a region handle (RGNHandle $\sigma\sharp\rho$).

The computation syntax also adds a new command form. The command witnessRGN $\sigma\sharp\rho_1\ \sigma\sharp\rho_2\ [\tau_a]\ v$ casts a computation from the type RGN $\sigma\sharp\rho_1\ \tau_a$ to the type RGN $\sigma\sharp\rho_2\ \tau_a$. (This command is used to construct terms of the type $\tau_{r_1} \preceq \tau_{r_2} \equiv \forall\beta.\mathsf{RGN}\ \tau_{r_1}\ \beta \to \mathsf{RGN}\ \tau_{r_2}\ \beta$ introduced in Section 2.) Operationally, such a command is the identify function, so long as the cast is valid. The static semantics of the next section ensure that all such casts in a well-typed program are valid.

Thus far, we have talked about region allocated data without discussing where such data is stored. Storable (i.e., closed) values are associated with locations in regions R; regions are ordered into stacks S; finally, stacks are ordered into towers T.

$$\boxed{T; e \hookrightarrow v}$$

$$\frac{}{T; i \hookrightarrow i} \qquad \frac{\begin{array}{cc} T; e_1 \hookrightarrow v_1 & v_1 \equiv i_1 \\ T; e_2 \hookrightarrow v_2 & v_2 \equiv i_2 \\ \multicolumn{2}{c}{i_1 \oplus i_2 = i} \end{array}}{T; e_1 \oplus e_2 \hookrightarrow i} \qquad \frac{\begin{array}{cc} T; e_1 \hookrightarrow v_1 & v_1 \equiv i_1 \\ T; e_2 \hookrightarrow v_2 & v_2 \equiv i_2 \\ \multicolumn{2}{c}{i_1 \olessthan i_2 = b} \end{array}}{T; e_1 \olessthan e_2 \hookrightarrow b} \qquad \frac{}{T; \mathsf{tt} \hookrightarrow \mathsf{tt}}$$

$$\frac{}{T; \mathsf{ff} \hookrightarrow \mathsf{ff}} \qquad \frac{\begin{array}{cc} T; e_b \hookrightarrow v_b & v_b \equiv \mathsf{tt} \\ \multicolumn{2}{c}{T; e_t \hookrightarrow v} \end{array}}{T; \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \hookrightarrow v} \qquad \frac{\begin{array}{cc} T; e_b \hookrightarrow v_b & v_b \equiv \mathsf{ff} \\ \multicolumn{2}{c}{T; e_f \hookrightarrow v} \end{array}}{T; \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \hookrightarrow v}$$

$$\frac{}{T; \lambda x{:}\tau.e \hookrightarrow \lambda x{:}\tau.e} \qquad \frac{\begin{array}{cc} T; e_1 \hookrightarrow v_1 & v_1 \equiv \lambda x{:}\tau_1.e_1' \\ T; e_2 \hookrightarrow v_2 & T; e_1'[v_2/x] \hookrightarrow v_3 \end{array}}{T; e_1\ e_2 \hookrightarrow v_3}$$

$$\frac{T; e_1 \hookrightarrow v_1 \quad \ldots \quad T; e_n \hookrightarrow v_n}{T; (e_1, \ldots, e_n) \hookrightarrow (v_1, \ldots, v_n)} \qquad \frac{\begin{array}{cc} T; e \hookrightarrow v & v \equiv (v_1, \ldots, v_n) \\ \multicolumn{2}{c}{1 \leq i \leq n} \end{array}}{T; \mathsf{sel}_i\ e \hookrightarrow v_i}$$

$$\frac{}{T; \Lambda\alpha.e \hookrightarrow \Lambda\alpha.e} \qquad \frac{\begin{array}{cc} T; e \hookrightarrow v & v \equiv \Lambda\alpha.e' \\ \multicolumn{2}{c}{T; e'[\tau/\alpha] \hookrightarrow v'} \end{array}}{T; e\ [\tau] \hookrightarrow v'} \qquad \frac{\begin{array}{c} T; e_1 \hookrightarrow v_1 \\ T; e_2[v_1/x] \hookrightarrow v_2 \end{array}}{T; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow v_2}$$

$$\frac{\begin{array}{cc} \multicolumn{2}{c}{s \notin \mathit{dom}(T)} \\ T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v' & v' \equiv \kappa' \\ T, s \mapsto (\cdot, r \mapsto \{\}); \kappa' \hookrightarrow_\kappa S''; v'' & S'' \equiv \cdot, r \mapsto R'' \end{array}}{T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow v''[\circ\sharp\bullet/s\sharp r]} \qquad \frac{}{T; \kappa \hookrightarrow \kappa}$$

$$\frac{}{T; \langle l \rangle_{\sigma\sharp\rho} \hookrightarrow \langle l \rangle_{\sigma\sharp\rho}} \qquad \frac{}{T; \mathsf{handle}(\sigma\sharp\rho) \hookrightarrow \mathsf{handle}(\sigma\sharp\rho)}$$

Fig. 3. Dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions)

We use the notation $S(r, l)$ and $T(s, r, l)$ for iterated lookups of values in stacks and towers, repectively. Again, towers are a technical device that serve to distinguish nested runRGN computations from one another. Intuitively, executing a runRGN computation adds a new stack to the top of the tower (the new stack is deallocated upon finishing the computation), while executing a letRGN command adds a new region to the top of the topmost stack (the new region is deallocated upon finishing the command). These intuitions are formalized in the dynamic semantics of the next section.

3.2 Dynamic Semantics of $\mathsf{F}^{\mathsf{RGN}}$

Two mutually inductive judgements (one for pure expressions (Figure 3) and one for monadic commands (Figure 4)) define the dynamic semantics. We state without proof that the dynamic semantics is deterministic; it is syntax-directed, taking $T; e$ configurations modulo α-conversion, including conversion of stack names, region names, and locations, which are (uniquely) bound in the tower T.

$$\boxed{T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v}$$

$$\frac{\tau_r \equiv s\sharp r}{T, s \mapsto S; \mathsf{returnRGN}\ [\tau_r]\ [\tau_a]\ v \hookrightarrow_\kappa S; v}$$

$$\frac{\begin{array}{c}\tau_r \equiv s\sharp r \qquad v_1 \equiv \kappa_1 \qquad T, s \mapsto S; \kappa_1 \hookrightarrow_\kappa S'; v_1' \\ T, s \mapsto S'; v_2\ v_1' \hookrightarrow v'' \qquad v'' \equiv \kappa'' \qquad T, s \mapsto S'; \kappa'' \hookrightarrow_\kappa S'''; v'''\end{array}}{T, s \mapsto S; \mathsf{thenRGN}\ [\tau_r]\ [\tau_a]\ [\tau_b]\ v_1\ v_2 \hookrightarrow_\kappa S'''; v'''}$$

$$\frac{\begin{array}{c}\tau_r \equiv s\sharp r_1 \qquad r_1 \in dom(S) \qquad r_2 \notin dom(S) \\ T, s \mapsto (S, r_2 \mapsto \{\}); v\ [s\sharp r_2]\ w\ \mathsf{handle}(s\sharp r_2) \hookrightarrow v' \qquad v' \equiv \kappa' \\ T, s \mapsto (S, r_2 \mapsto \{\}); \kappa' \hookrightarrow_\kappa S'''; v'' \qquad S''' \equiv S'', r_2 \mapsto R_2''\end{array}}{T, s \mapsto S; \mathsf{letRGN}\ [\tau_r]\ [\tau_a]\ v \hookrightarrow_\kappa S''[s\sharp\bullet/s\sharp r_2]; v''[s\sharp\bullet/s\sharp r_2]}$$
$$\text{where } w = (\Lambda\beta.\lambda k{:}\mathsf{RGN}\ s\sharp r_1\ \beta.\mathsf{witnessRGN}\ s\sharp r_1\ s\sharp r_2\ [\beta]\ k)$$

$$\frac{\begin{array}{c}\sigma\sharp\rho_1 \equiv s\sharp r_1 \qquad \sigma\sharp\rho_2 \equiv s\sharp r_2 \qquad v \equiv \kappa \\ S \equiv S_1, r_1 \mapsto R_1, S_2, r_2 \mapsto R_2, S_3 \qquad T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v'\end{array}}{T, s \mapsto S; \mathsf{witnessRGN}\ \sigma\sharp\rho_1\ \sigma\sharp\rho_2\ [\tau_a]\ v \hookrightarrow_\kappa S'; v'}$$

$$\frac{\tau_r \equiv s\sharp r \qquad v_1 \equiv \mathsf{handle}(s\sharp r) \qquad r \in dom(S) \qquad l \notin dom(S(r))}{T, s \mapsto S; \mathsf{newRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \hookrightarrow_\kappa S\{(r,l) \mapsto v_2\}; \langle l\rangle_{s\sharp r}}$$

$$\frac{\tau_r \equiv s\sharp r \qquad v \equiv \langle l\rangle_{s\sharp r} \qquad r \in dom(S) \qquad l \in dom(S(r)) \qquad S(r,l) = v'}{T, s \mapsto S; \mathsf{readRGNLoc}\ [\tau_r]\ [\tau_a]\ v \hookrightarrow_\kappa S; v'}$$

Fig. 4. Dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands)

The judgement $T; e \hookrightarrow v$ asserts that evaluating the closed expression e in tower T results in a value v. Likewise, the judgement $T, s \mapsto S; \kappa \hookrightarrow S'; v$ asserts that evaluating the closed monadic command κ in a non-empty tower whose top stack is S results in a new top stack S' and a value v.

The rules for $T; e \hookrightarrow v$ for expression forms other than runRGN are completely standard. The tower T is passed unchanged to sub-evaluations. The rule for runRGN $[\tau_a]$ v runs a monadic computation. The rule executes in the following manner. First, fresh stack and region names s and r are chosen. Next, the argument v is applied to the region index $s\sharp r$ and the region handle handle$(s\sharp r)$ and evaluated in the extended tower $T, s \mapsto (\cdot, r \mapsto \{\})$ (that is, the tower T extended with a stack consisting of a single empty region (bound to r) bound to s) to a monadic command κ'. This command is evaluated under the extended tower to a modified region and a value v''. The modified region is discarded, while occurrences of $s\sharp r$ are replaced by $\circ\sharp\bullet$ in v'', because the stack and region have been deallocated and are no longer accessible.

The rules for $T, s \mapsto S; \kappa \hookrightarrow S'; v$ peform monadic operations that side-effect the top stack. The monadic unit and bind operations are standard; in particular, note the manner in which the rule for thenRGN threads the modified top stack through the computation.

The rule for letRGN $[s\sharp r_1]$ τ_a v executes in much the same way as the rule for runRGN. First, a fresh region name r_2 is chosen. Next, the argument v is applied to the region index $s\sharp r_2$, a witness function, and the region handle $\mathsf{handle}(s\sharp r_2)$ and evaluated under an extended tower that adds an empty region bound to r_2 to the top of the stack. This evaluation yields a monadic command κ', which is also evaluated under the extended tower to a modified top stack and value v''. The modified top region is discarded, while occurrences of $s\sharp r_2$ are replaced by $s\sharp\bullet$ in the modified top stack and in v'', because the region has been deallocated and is no longer accessible.

The rule for witnessRGN permits a monadic computation to occur when the region names r_1 and r_2 appear in order in the top stack.

The rules for newRGNLoc and readRGNLoc respectively allocate and read region allocated data. The rule for newRGNLoc requires a region handle for a region in the top stack, chooses a fresh location in the region, and returns a modified top stack (with the value stored at the freshly chosen location) and the location. The rule for readRGNLoc requires a location into a region in the top stack, and returns the value stored in the location.

It is important to note that the execution of a monadic command is predicated upon the command's region index corresponding to a live region in the top stack. While it will be possible to have commands that reference deallocated stacks and regions, it will not be possible to execute them. Furthermore, the restriction to the top stack corresponds to the fact that while runRGN computations can be nested, the inner computation must complete before executing a command in the outer computation. The type system of the next section ensures that these invariants are preserved during the execution of well-typed programs.

3.3 Natural Transition Semantics of $\mathsf{F}^{\mathsf{RGN}}$

While the dynamic semantics presented thus far suffices to describe the complete execution of a program, it cannot describe non-terminating executions or failed executions. To do so, we adopt a *natural transition semantics* (Volpano & Smith, 1997; Smith & Volpano, 1998), which provides a notion of attempted or partial execution. The key idea is to model program execution as a sequence of *partial derivation trees*, which may or may not converge to a complete derivation. The advantage of the natural transition semantics is that it is directly related to the large-step operational semantics of the language, while being capable of describing the evaluation of programs that (a) diverge, (b) terminate with a value, and (c) "get stuck."

Before defining partial derivation trees, we distinguish between *complete judgements* ($T; e \hookrightarrow v$ and $T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v$, introduced in the dynamic semantics) and *pending judgements*, which are judgements of the form $T; e \hookrightarrow\ ?$ or $T, s \mapsto S; \kappa \hookrightarrow_\kappa\ ?$ and represent expressions and commands that need to be evaluated.

A *partial derivation tree* is an inductively defined structure given by the following grammar:

Predicates	P		
Complete derivations ⋆	$\mathfrak{J}$	$::=$	$[T; e \hookrightarrow v] \mid [T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v] \mid P$
Partial derivation trees	$\mathfrak{D}$	$::=$	$\mathfrak{J} \mid [T; e \hookrightarrow ?]() \mid [T, s \mapsto S; \kappa \hookrightarrow_\kappa ?]()$
			$\mid [T; e \hookrightarrow ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_{k-1}, \mathfrak{D}_k)$ †
			$\mid [T, s \mapsto S; \kappa \hookrightarrow_\kappa ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_{k-1}, \mathfrak{D}_k)$ ‡

where

⋆ A complete derivation represents the entire derivation tree (comprised of instances of the evaluation rules) that terminates with the eponymous complete judgement.

† There is an instance of an evaluation rule with the form

$$\frac{J_1 \qquad \cdots \qquad J_n}{T; e \hookrightarrow v}$$

where $1 \leq k \leq n$ and

— for $i < k$, $\mathfrak{J}_i \equiv [J_i]$.
— if $J_k \equiv T; e \hookrightarrow v$, then $\mathfrak{D}_k = [T; e \hookrightarrow v]$ or $\mathfrak{D}_k = [T; e \hookrightarrow ?](\ldots)$.
— if $J_k \equiv T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v$, then $\mathfrak{D}_k = [T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v]$ or $\mathfrak{D}_k = [T, s \mapsto S; \kappa \hookrightarrow_\kappa ?](\ldots)$.
— if $J_k \equiv P$, then $\mathfrak{D}_k = P$.

‡ There is an instance of an evaluation rule with the form

$$\frac{J_1 \qquad \cdots \qquad J_n}{T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v}$$

where $1 \leq k \leq n$ and

— for $i < k$, $\mathfrak{J}_i \equiv [J_i]$.
— if $J_k \equiv T; e \hookrightarrow v$, then $\mathfrak{D}_k = [T; e \hookrightarrow v]$ or $\mathfrak{D}_k = [T; e \hookrightarrow ?](\ldots)$.
— if $J_k \equiv T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v$, then $\mathfrak{D}_k = [T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v]$ or $\mathfrak{D}_k = [T, s \mapsto S; \kappa \hookrightarrow_\kappa ?](\ldots)$.
— if $J_k \equiv P$, then $\mathfrak{D}_k = P$.

Note that the definition of a partial derivation tree requires that a node labeled with a pending judgement must have children that are "compatible" with the corresponding complete judgement. Furthermore, each node of a partial derivation tree can have at most one pending judgement amongst its children; the pending judgement must be the rightmost child and the parent node must also be a pending judgement.

Figure 5 gives (a representative sample of) the rules for the natural transition semantics. The rules are derived systematically from the judgements of Figures 3 and 4. In addition, note the two "congruence" rules. Finally, it should be clear that each transition moves a partial derivation tree "closer" to a complete judgement. Let $\longrightarrow^*$ be the reflexive, transitive closure of the $\longrightarrow$ relation.

The natural transition semantics enjoys soundness and completeness properties demonstrating that it accurately models the dynamic semantics in the case of terminating computations.

Lemma 3.1
If $\mathfrak{D}$ is a partial derivation and $\mathfrak{D} \longrightarrow \mathfrak{D}'$, then $\mathfrak{D}'$ is a partial derivation.

$\boxed{\mathfrak{D} \longrightarrow \mathfrak{D}'}$

$$[T; \lambda x{:}\tau.e \hookrightarrow ?]() \longrightarrow \left[\frac{}{T; \lambda x{:}\tau.e \hookrightarrow \lambda x{:}\tau.e}\right]$$

$$[T; e_1\ e_2 \hookrightarrow ?]() \longrightarrow [T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow ?]())$$

$$\frac{v_1 \equiv \lambda x{:}\tau_1.e_1'}{[T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow v_1]) \longrightarrow [T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow v_1], v_1 \equiv \lambda x{:}\tau_1.e_1')}$$

$$\begin{array}{l}[T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow v_1], v_1 \equiv \lambda x{:}\tau_1.e_1') \longrightarrow \\ \quad [T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow v_1], v_1 \equiv \lambda x{:}\tau_1.e_1', [T; e_2 \hookrightarrow ?]())\end{array}$$

$$\begin{array}{l}[T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow v_1], v_1 \equiv \lambda x{:}\tau_1.e_1', [T; e_2 \hookrightarrow v_2]) \longrightarrow \\ \quad [T; e_1\ e_2 \hookrightarrow ?]([T; e_1 \hookrightarrow v_1], v_1 \equiv \lambda x{:}\tau_1.e_1', [T; e_2 \hookrightarrow v_2], [T; e_1'[v_2/x] \hookrightarrow ?]())\end{array}$$

$$[T; e_1\ e_2 \hookrightarrow ?]\begin{pmatrix}[T; e_1 \hookrightarrow v_1], v_1 \equiv \lambda x{:}\tau_1.e_1', \\ [T; e_2 \hookrightarrow v_2], [T; e_1'[v_2/x] \hookrightarrow v_3]\end{pmatrix} \longrightarrow \left[\frac{\begin{array}{cc}T; e_1 \hookrightarrow v_1 & v_1 \equiv \lambda x{:}\tau_1.e_1' \\ T; e_2 \hookrightarrow v_2 & T; e_1'[v_2/x] \hookrightarrow v_3\end{array}}{T; e_1\ e_2 \hookrightarrow v_3}\right]$$

$$\frac{s \notin \mathit{dom}(T)}{[T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]() \longrightarrow [T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?](s \notin \mathit{dom}(T))}$$

$$\begin{array}{l}[T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?](s \notin \mathit{dom}(T)) \longrightarrow \\ \quad [T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?](s \notin \mathit{dom}(T), [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow ?]())\end{array}$$

$$\frac{v' \equiv \kappa'}{\begin{array}{l}[T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v']\end{pmatrix} \longrightarrow \\ \quad [T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v'], v' \equiv \kappa'\end{pmatrix}\end{array}}$$

$$\begin{array}{l}[T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v'], v' \equiv \kappa'\end{pmatrix} \longrightarrow \\ \quad [T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v'], v' \equiv \kappa', \\ [T, s \mapsto (\cdot, r \mapsto \{\}); \kappa' \hookrightarrow_\kappa ?]()\end{pmatrix}\end{array}$$

$$\frac{S'' \equiv \cdot, r \mapsto R''}{\begin{array}{l}[T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v'], v' \equiv \kappa', \\ [T, s \mapsto (\cdot, r \mapsto \{\}); \kappa' \hookrightarrow_\kappa S''; v'']\end{pmatrix} \longrightarrow \\ \quad [T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v'], v' \equiv \kappa', \\ [T, s \mapsto (\cdot, r \mapsto \{\}); \kappa' \hookrightarrow_\kappa S''; v''], [S'' \equiv \cdot, r \mapsto R'']\end{pmatrix}\end{array}}$$

$$\begin{array}{l}[T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow ?]\begin{pmatrix}s \notin \mathit{dom}(T), \\ [T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v'], v' \equiv \kappa', \\ [T, s \mapsto (\cdot, r \mapsto \{\}); \kappa' \hookrightarrow_\kappa S''; v''], S'' \equiv \cdot, r \mapsto R''\end{pmatrix} \longrightarrow \\ \quad \left[\frac{\begin{array}{c}s \notin \mathit{dom}(T) \\ T, s \mapsto (\cdot, r \mapsto \{\}); v\ [s\sharp r]\ \mathsf{handle}(s\sharp r) \hookrightarrow v' \qquad v' \equiv \kappa' \\ T, s \mapsto (\cdot, r \mapsto \{\}); \kappa' \hookrightarrow_\kappa S''; v'' \qquad S'' \equiv \cdot, r \mapsto R''\end{array}}{T; \mathsf{runRGN}\ [\tau_a]\ v \hookrightarrow v''[\circ\sharp\bullet / s\sharp r]}\right]\end{array}$$

$$\frac{\mathfrak{D} \longrightarrow \mathfrak{D}'}{\begin{array}{l}[T; e \hookrightarrow ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}) \longrightarrow \\ \quad [T; e \hookrightarrow ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}')\end{array}} \qquad \frac{\mathfrak{D} \longrightarrow \mathfrak{D}'}{\begin{array}{l}[T, s \mapsto S; \kappa \hookrightarrow_\kappa ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}) \longrightarrow \\ \quad [T, s \mapsto S; \kappa \hookrightarrow_\kappa ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}')\end{array}}$$

Fig. 5. Natural transition semantics of $\mathsf{F}^{\mathsf{RGN}}$ (abbreviated)

Lemma 3.2 (*NTS Soundness*)

(1) If $[T; e \hookrightarrow ?]() \longrightarrow^* \mathfrak{D}'$ and $\mathfrak{D}'$ contains no pending judgements, then $\mathfrak{D}'$ is a complete derivation for a judgement of the form $T; e \hookrightarrow v$ (i.e., $\mathfrak{D}' \equiv \mathfrak{J}' \equiv [T; e \hookrightarrow v]$).

(2) If $[T, s \mapsto S; \kappa \hookrightarrow_\kappa ?]() \longrightarrow^* \mathfrak{D}'$ and $\mathfrak{D}'$ contains no pending judgements, then $\mathfrak{D}'$ is a complete derivation for a judgement of the form $T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v$ (i.e., $\mathfrak{D}' \equiv \mathfrak{J}' \equiv [T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v]$).

Lemma 3.3 (*NTS Completeness*)

(1) If $T; e \hookrightarrow v$ and $\mathfrak{D}$ is a complete derivation for $T; e \hookrightarrow v$, then $[T; e \hookrightarrow ?]() \longrightarrow^* \mathfrak{D}$.

(2) If $T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v$ and $\mathfrak{D}_\kappa$ is a complete derivation for $T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v$, then $[T, s \mapsto S; \kappa \hookrightarrow_\kappa ?]() \longrightarrow^* \mathfrak{D}_\kappa$.

For each tower T and expression e, we define an execution of e in T as a sequence

$$[T; e \hookrightarrow ?]() \longrightarrow \mathfrak{D}_1 \longrightarrow \mathfrak{D}_2 \longrightarrow \cdots$$

Thus, an execution has three possibilities:

(1) Suppose that for all $\mathfrak{D}_n$ such that $[T; e \hookrightarrow ?]() \longrightarrow^* \mathfrak{D}_n$, there exists $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$. Then, we say that e in T diverges.

(2) Suppose that there exists $\mathfrak{D}_n$ such that $[T; e \hookrightarrow ?]() \longrightarrow^* \mathfrak{D}_n$, such that there does not exist $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$.
   (a) Suppose $\mathfrak{D}_n$ contains no pending judgements. By Lemma 3.2, $\mathfrak{D}_n \equiv [T; e \hookrightarrow v]$. Then, we say that e in T terminates with the value v.
   (b) Suppose $\mathfrak{D}_n$ contains pending judgements. Then, we say that e in T gets stuck.

By inspection of the rules in Figure 5, it is clear that the stuck partial derivation trees correspond to trees in which predicates cannot be satisfied; all other transitions are unrestricted. Predicates like $v \equiv \lambda x{:}\tau.e$ and $v \equiv \kappa$ are traditional type errors, where expressions evaluate to values of the wrong form. Predicates like $s \in \mathit{dom}(T)$ also correspond to type errors, where towers have the wrong form. The static semantics given in the next section and the definitions given in Section 3.5 ensure that stuck partial derivation trees are not well-typed.

3.4 Static Semantics of $\mathsf{F}^{\mathsf{RGN}}$

Well-typed programs obey several invariants, which are enforced with typing judgements. In addition to the traditional "type-checking" judgements for expressions, we have judgements that enforce the consistency of towers, and various well-formedness judgements that serve as a technical convenience.

3.4.1 Definitions

Figure 6 presents additional definitions for syntactic classes that appear in the static semantics. Type contexts Δ are ordered lists of type variables and value contexts Γ are ordered lists of variables and types. Tower, stack, and region types mimic towers, stacks, and regions, recording the type of the value stored at each location. Tower, stack, and region domains are a technical device that records the locations

Type contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha$	
Value contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$	
Region domains	$\overline{\mathcal{R}}$	$::=$	$\{l_1, \ldots, l_n\}$	
Region types	$\mathcal{R}$	$::=$	$\{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\}$	
Stack domains	$\overline{\mathcal{S}}$	$::=$	$\cdot \mid \overline{\mathcal{S}}, r \mapsto \overline{\mathcal{R}}$	(ordered domain)
Stack types	$\mathcal{S}$	$::=$	$\cdot \mid \mathcal{S}, r \mapsto \mathcal{R}$	(ordered domain)
Tower domains	$\overline{\mathcal{T}}$	$::=$	$\cdot \mid \overline{\mathcal{T}}, s \mapsto \overline{\mathcal{S}}$	(ordered domain)
Tower types	$\mathcal{T}$	$::=$	$\cdot \mid \mathcal{T}, s \mapsto \mathcal{S}$	(ordered domain)

$$\tau_{r_1} \preceq \tau_{r_2} \equiv \forall \beta.\mathsf{RGN}\ \tau_{r_1}\ \beta \to \mathsf{RGN}\ \tau_{r_2}\ \beta$$

$$\begin{aligned} \overline{\mathcal{S}} \sqsupseteq \overline{\mathcal{S}}' \quad \equiv \quad & \mathit{dom}(\overline{\mathcal{S}}) = \mathit{dom}(\overline{\mathcal{S}}') \wedge \\ & \forall r \in \mathit{dom}(\overline{\mathcal{S}}').\mathit{dom}(\overline{\mathcal{S}}(r)) \supseteq \mathit{dom}(\overline{\mathcal{S}}'(r)) \end{aligned}$$

$$\overline{\mathcal{T}}|_s \equiv \overline{\mathcal{T}}', s \mapsto \overline{\mathcal{S}}' \quad \text{such that } \overline{\mathcal{T}} \equiv \overline{\mathcal{T}}', s \mapsto \overline{\mathcal{S}}', \overline{\mathcal{T}}''$$

$$\begin{aligned} \mathcal{S} \sqsupseteq \mathcal{S}' \quad \equiv \quad & \mathit{dom}(\mathcal{S}) = \mathit{dom}(\mathcal{S}') \wedge \\ & \forall r \in \mathit{dom}(\mathcal{S}').\mathit{dom}(\mathcal{S}(r)) \supseteq \mathit{dom}(\mathcal{S}'(r)) \wedge \\ & \qquad \forall l \in \mathit{dom}(\mathcal{S}'(r)).\mathcal{S}(r, l) = \mathcal{S}'(r, l) \end{aligned}$$

$$\mathcal{T}|_s \equiv \mathcal{T}', s \mapsto \mathcal{S}' \quad \text{such that } \mathcal{T} \equiv \mathcal{T}', s \mapsto \mathcal{S}', \mathcal{T}''$$

Fig. 6. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (definitions)

in scope. Because proving the well-formedness of tower types requires proving the well-formedness of types, which requires verifying that stack and region names are in scope, one cannot easily have tower types serve the dual purpose of recording locations in scope. We tacitly assume that all domains are well-formed – containing distinct stack names, region names, and locations.

We recall the abbreviation $\tau_{r_1} \preceq \tau_{r_2}$ for the type of a function that coerces any computation taking place in the region indexed by τ_{r_1} into a computation taking place in the region indexed by τ_{r_2}.

We define the relation $\sqsupseteq$ to describe extensions of stack domains and types. Note that we consider tower and stack domains and types to have ordered domains. Hence, $\mathit{dom}(S) \supseteq \mathit{dom}(S')$ indicates that the ordered domain of $\mathit{dom}(S')$ is a prefix of the ordered domain of $\mathit{dom}(S)$. Finally, we define restriction operators, which return a prefix of tower domains and types.

3.4.2 Expressions

Figures 7, 8, and 10 present the typing rules for the judgement $\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau$, which asserts that under type context Δ, value context Γ, and tower type $\mathcal{T}$ with tower domain $\overline{\mathcal{T}}$, the expression e has type τ.

The rules for constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and elimination, and type abstraction and instantiation are all completely standard. As expected in a monadic language, each command expression is given the monadic type $\mathsf{RGN}\ \tau_r\ \tau_a$ for appropriate region

$$\boxed{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e : \tau}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} i : \mathsf{int}} \qquad \frac{\begin{array}{c}\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1 : \mathsf{int}\\ \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_2 : \mathsf{int}\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1 \oplus e_2 : \mathsf{int}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1 : \mathsf{int}\\ \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_2 : \mathsf{int}\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1 \circledcirc e_2 : \mathsf{bool}} \qquad \frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{tt} : \mathsf{bool}} \qquad \frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{ff} : \mathsf{bool}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_b : \mathsf{bool}\\ \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_t : \tau \qquad \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_f : \tau\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \qquad x \in \mathit{dom}(\Gamma) \qquad \tau = \Gamma(x)}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} x : \tau} \qquad \frac{\Delta;\Gamma,x{:}\tau_1;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e : \tau_2}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1 : \tau_1 \to \tau_2\\ \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_2 : \tau_1\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1\ e_2 : \tau_2} \qquad \frac{\begin{array}{c}\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}}\\ \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_i : \tau_i\ ^{i\in 1\ldots n}\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} (e_1,\ldots,e_n) : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e : \tau_1 \times \cdots \times \tau_n\\ 1 \le i \le n\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{sel}_i\ e : \tau_i} \qquad \frac{\begin{array}{c}\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}}\\ \Delta,\alpha;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e : \tau\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \Lambda\alpha.e : \forall\alpha.\tau}$$

$$\frac{\begin{array}{c}\Delta;\overline{\mathcal{T}} \vdash_{\mathrm{type}} \tau\\ \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e : \forall\alpha.\tau'\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e\ [\tau] : \tau'[\tau/\alpha]} \qquad \frac{\begin{array}{c}\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_1 : \tau_1\\ \Delta;\Gamma,x{:}\tau_1;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e_2 : \tau_2\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{\Delta;\overline{\mathcal{T}} \vdash_{\mathrm{type}} \tau_a \qquad \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} v : \forall\gamma_r.\mathsf{RGNHandle}\ \gamma_r \to \mathsf{RGN}\ \gamma_r\ \tau_a}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{runRGN}\ [\tau_a]\ v : \tau_a}$$

Fig. 7. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions)

index and return type. The typing rules for returnRGN and thenRGN correspond to the standard typing rules for monadic unit and bind operations. The typing rules for newRGNLoc and readRGNLoc are straight-forward.

The key judgements are those relating to the creation of new regions. Recall that we would like to consider a value of type RGN τ_r τ_a as a region-transformer – that is, it accepts a region (indexed by the type τ_r), performs some operations (such as allocating into the region), and returns a value and the modified region. However, this is slightly inaccurate, owing to the fact that a stack of regions admits a region *outlives* relationship. Rather, we should consider a value of type RGN τ_r τ_a as a region-stack-transformer – that is, it accepts a stack of regions (indexed by the type τ_r, corresponding to a particular member of the region stack), performs some operations (such as allocating into the regions), and returns a value and the

$\boxed{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_r \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \tau_a}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{returnRGN}\ [\tau_r]\ [\tau_a]\ v : \mathsf{RGN}\ \tau_r\ \tau_a}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_1 : \mathsf{RGN}\ \tau_r\ \tau_a \\ \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_2 : \tau_a \to \mathsf{RGN}\ \tau_r\ \tau_b\end{array}}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{thenRGN}\ [\tau_r]\ [\tau_a]\ [\tau_b]\ v_1\ v_2 : \mathsf{RGN}\ \tau_r\ \tau_b}$$

$$\frac{\begin{array}{c}\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_{r_1} \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_a \\ \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \forall \gamma_{r_2} . \tau_{r_1} \preceq \gamma_{r_2} \to \mathsf{RGNHandle}\ \gamma_{r_2} \to \mathsf{RGN}\ \gamma_{r_2}\ \tau_a\end{array}}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{letRGN}\ [\tau_{r_1}]\ [\tau_a]\ v : \mathsf{RGN}\ \tau_{r_1}\ \tau_a}$$

$$\frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \mathsf{RGN}\ \sigma\sharp\rho_1\ \tau_a \qquad \overline{\mathcal{T}} \vdash_{\text{cast}} \sigma\sharp\rho_1 \rightsquigarrow \sigma\sharp\rho_2}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{witnessRGN}\ \sigma\sharp\rho_1\ \sigma\sharp\rho_2\ [\tau_a]\ v : \mathsf{RGN}\ \sigma\sharp\rho_2\ \tau_a}$$

$$\frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v_1 : \mathsf{RGNHandle}\ \tau_r \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v_2 : \tau_a}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{newRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 : \mathsf{RGN}\ \tau_r\ (\mathsf{RGNLoc}\ \tau_r\ \tau_a)}$$

$$\frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \mathsf{RGNLoc}\ \tau_r\ \tau_a}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{readRGNLoc}\ [\tau_r]\ [\tau_a]\ v : \mathsf{RGN}\ \tau_r\ \tau_a}$$

Fig. 8. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands)

modified stack of regions. Note that the actual stack of regions passed at runtime may include regions younger than the region corresponding to τ_r; τ_r simply ensures the liveness of a particular region (and all regions older than it), without excluding the liveness of younger regions.

We first examine the typing rule for the runRGN expression:

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_a \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \forall \gamma_r . \mathsf{RGNHandle}\ \gamma_r \to \mathsf{RGN}\ \gamma_r\ \tau_a}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathsf{runRGN}\ [\tau_a]\ v : \tau_a}$$

As stated above, the argument to runRGN should describe a region computation. In fact, we require v to be a polymorphic function that yields a region computation after being applied to a (fresh) region handle. The effect of universally quantifying the region index in the type of v is to require v to make no assumptions about the input stack of regions (e.g., the existence of pre-allocated values). Furthermore, all region-transformer operations are "infected" with the region index: when combining operations, the rule for thenRGN requires the region index type to be the same; locations allocated and read using newRGNLoc and readRGNLoc require the region index of the RGNLoc to be the same as the computation in which the operation occurs. While witness functions (discussed in more detail below) may coerce a region computation indexed by τ_r to a region computation indexed by τ_r' for a younger region index τ_r', this coercion simply "infects" the computation with a younger region index that implies the older region index. Thus, if a region computation $\mathsf{RGN}\ \gamma_r\ \tau_a$ were to return a value that depended upon the region indexed by γ_r,

$\boxed{\overline{\mathfrak{T}} \vdash_{\text{cast}} \sigma\sharp\rho \rightsquigarrow \sigma\sharp\rho'}$

$$\frac{}{\overline{\mathfrak{T}} \vdash_{\text{cast}} \circ\sharp\bullet \rightsquigarrow \circ\sharp\bullet} \qquad \frac{s \in \mathit{dom}(\overline{\mathfrak{T}})}{\overline{\mathfrak{T}} \vdash_{\text{cast}} s\sharp\bullet \rightsquigarrow s\sharp\bullet} \qquad \frac{s \in \mathit{dom}(\overline{\mathfrak{T}}) \qquad \overline{\mathfrak{T}}(s) \equiv \overline{\mathfrak{S}}_1, r_1 \mapsto \overline{\mathfrak{R}}_1, \overline{\mathfrak{S}}_2}{\overline{\mathfrak{T}} \vdash_{\text{cast}} s\sharp r_1 \rightsquigarrow s\sharp\bullet}$$

$$\frac{s \in \mathit{dom}(\overline{\mathfrak{T}}) \qquad \overline{\mathfrak{T}}(s) \equiv \overline{\mathfrak{S}}_1, r_1 \mapsto \overline{\mathfrak{R}}_1, \overline{\mathfrak{S}}_2, r_2 \mapsto \overline{\mathfrak{R}}_2, \overline{\mathfrak{S}}_3}{\overline{\mathfrak{T}} \vdash_{\text{cast}} s\sharp r_1 \rightsquigarrow s\sharp r_2}$$

Fig. 9. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (casts)

then γ_r (or some younger, as of yet unintroduced, region index γ_r') would appear in the type τ_a. Since the type τ_a appears outside the scope of the type variable γ_r in the typing rule for runRGN, it follows that γ_r cannot appear in the type τ_a. Therefore, it must be the case that the value returned by the computation described by v does not depend upon the region index which will instantiate γ_r. Taken together, these facts ensure that an arbitrary new region can be supplied to the computation and that the value returned will not leak any means of accessing the region or values allocated within it; hence, the region can be destroyed at the end of the computation. Finally, because we require region handles for allocating within regions, we provide the region handle for the newly created region as the argument to a function that yields the computation we wish to execute.

The typing rule for letRGN is very similar:

$$\frac{\begin{array}{c}\Delta; \overline{\mathfrak{T}} \vdash_{\text{type}} \tau_{r_1} \qquad \Delta; \overline{\mathfrak{T}} \vdash_{\text{type}} \tau_a \\ \Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} v : \forall \gamma_{r_2}. \tau_{r_1} \preceq \gamma_{r_2} \to \mathsf{RGNHandle}\ \gamma_{r_2} \to \mathsf{RGN}\ \gamma_{r_2}\ \tau_a\end{array}}{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} \mathsf{letRGN}\ [\tau_{r_1}]\ [\tau_a]\ v : \mathsf{RGN}\ \tau_{r_1}\ \tau_a}$$

Exactly the same argument as above applies, except that we additionally have an witness argument of type $\tau_{r_1} \preceq \gamma_{r_2}$. The operational behavior of letRGN is ensure that the newly allocated region is related to previously allocated regions according to the stack discipline. The witness argument is provided to the computation taking place in the stack with the inner/younger region allocated in order to coerce computations (such as allocating a new value in some outer/older region) from a computation indexed by the outer region to a computation indexed by the the inner region. This coersion is safe because every region in the stack denoted by τ_{r_1} outlives every region in the stack denoted by γ_{r_2}. Operationally, such a witness function acts as the identity function.

The typing rule for witnessRGN formalizes this outlives argument: a witnessRGN term is well-typed whenever $\sigma\sharp\rho_1$ can be cast to $\sigma\sharp\rho_2$. The judgement $\overline{\mathfrak{T}} \vdash_{\text{cast}} \sigma\sharp\rho_1 \rightsquigarrow \sigma\sharp\rho_2$ (Figure 9) verifies the casts witnessed by witnessRGN terms. Note that the judgement $\overline{\mathfrak{T}} \vdash_{\text{cast}} s\sharp r_1 \rightsquigarrow s\sharp r_2$ enforces the requirement that r_1 outlives r_2 in the stack s. The other $\vdash_{\text{cast}}$ judgements allow casts to deallocated regions, which can be introduced when deallocating a region at the end of a runRGN or letRGN computation. This is a technicality needed to ensure that programs remain closed and well-typed during their execution.

Figure 10 gives typing rules for location and handle expressions. The judgements ensure that stack and region names that appear in locations and handles are in

$$\boxed{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} e : \tau}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \qquad \Delta;\overline{\mathcal{T}} \vdash_{\mathrm{type}} \tau_a}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \langle l\rangle_{\circ\sharp\bullet} : \mathsf{RGNLoc}\ \circ\sharp\bullet\ \tau_a}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \qquad s \in \mathit{dom}(\overline{\mathcal{T}}) \qquad \Delta;\overline{\mathcal{T}} \vdash_{\mathrm{type}} \tau_a}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \langle l\rangle_{s\sharp\bullet} : \mathsf{RGNLoc}\ s\sharp\bullet\ \tau_a}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \\ s \in \mathit{dom}(\overline{\mathcal{T}}) \qquad r \in \mathit{dom}(\overline{\mathcal{T}}(s)) \qquad l \in \mathit{dom}(\overline{\mathcal{T}}(s,r)) \qquad \mathcal{T}(s,r,l) = \tau_a\end{array}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \langle l\rangle_{s\sharp r} : \mathsf{RGNLoc}\ s\sharp r\ \tau_a}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}}}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{handle}(\circ\sharp\bullet) : \mathsf{RGNHandle}\ \circ\sharp\bullet}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \qquad s \in \mathit{dom}(\overline{\mathcal{T}})}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{handle}(s\sharp\bullet) : \mathsf{RGNHandle}\ s\sharp\bullet}$$

$$\frac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \qquad s \in \mathit{dom}(\overline{\mathcal{T}}) \qquad r \in \mathit{dom}(\overline{\mathcal{T}}(s))}{\Delta;\Gamma;\mathcal{T}:\overline{\mathcal{T}} \vdash_{\mathrm{exp}} \mathsf{handle}(s\sharp r) : \mathsf{RGNHandle}\ s\sharp r}$$

Fig. 10. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (locations and handles)

scope; furthermore, a location in a live stack and region points to a value with the type assigned by the tower type.

3.4.3 Towers

Figure 11 presents typing rules that enforce the well-formedness and consistency of towers. The judgement $\vdash_{\mathrm{ttype}} \mathcal{T} : \overline{\mathcal{T}}$ asserts that tower type $\mathcal{T}$ is well-formed with tower domain $\overline{\mathcal{T}}$. In particular, the judgement asserts that $\mathcal{T}$ has the domain specified by $\overline{\mathcal{T}}$ and each type in the range of $\mathcal{T}$ is well-formed. Note the use of the restriction operator; this ensures that types "lower" in the tower cannot reference stack and region names that appear "higher" in the tower. This corresponds to the fact that while runRGN computations can be nested, the inner computation must complete before executing a command in the outer computation. Hence, while an inner computation may have references to the outer computation, there can be no references from the outer computation to the inner computation. Finally, the judgement $\vdash_{\mathrm{tower}} T : \mathcal{T} : \overline{\mathcal{T}}$ asserts that the tower T is well-formed with tower type $\mathcal{T}$ and tower domain $\overline{\mathcal{T}}$. Like the judgement $\vdash_{\mathrm{ttype}}$, it asserts that T has the domain specified by $\overline{\mathcal{T}}$ and each stored value in the range of T has the type specified by $\mathcal{T}$. Again, restriction operators are used to assert that storable values "lower" in the tower cannot contain references to storable values "higher" in the tower.

$\boxed{\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}}}$

$$\frac{\begin{array}{c} dom(\overline{\mathcal{T}}) = dom(\mathcal{T}) \\ \forall s \in dom(\overline{\mathcal{T}}).\ dom(\overline{\mathcal{T}}(s)) = dom(\mathcal{T}(s)) \\ \forall s \in dom(\overline{\mathcal{T}}).\forall r \in dom(\overline{\mathcal{T}}(s))\ dom(\overline{\mathcal{T}}(s,r)) = dom(\mathcal{T}(s,r)) \\ \forall s \in dom(\overline{\mathcal{T}}).\forall r \in dom(\overline{\mathcal{T}}(s)).\forall l \in dom(\overline{\mathcal{T}}(s,r)).\ \cdot; \overline{\mathcal{T}}|_s \vdash_{\text{type}} \mathcal{T}(s,r,l) \end{array}}{\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}}}$$

$\boxed{\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}}$

$$\frac{\begin{array}{c} \vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}} \\ dom(\overline{\mathcal{T}}) = dom(\mathcal{T}) = dom(T) \\ \forall s \in dom(\overline{\mathcal{T}}).\ dom(\overline{\mathcal{T}}(s)) = dom(\mathcal{T}(s)) = dom(T(s)) \\ \forall s \in dom(\overline{\mathcal{T}}).\forall r \in dom(\overline{\mathcal{T}}(s))\ dom(\overline{\mathcal{T}}(s,r)) = dom(\mathcal{T}(s,r)) = dom(T(s,r)) \\ \forall s \in dom(\overline{\mathcal{T}}).\forall r \in dom(\overline{\mathcal{T}}(s)).\forall l \in dom(\overline{\mathcal{T}}(s,r)).\ \cdot; \cdot; \mathcal{T}|_s : \overline{\mathcal{T}}|_s \vdash_{\text{exp}} T(s,r,l) : \mathcal{T}(s,r,l) \end{array}}{\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}}$$

Fig. 11. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (towers, stacks, and regions)

3.4.4 Technical details

Figures A 1 and A 2 in Appendix A contain additional (completely standard) judgements for ensuring that types τ, type contexts Δ, and value contexts Γ are well-formed. Because types may contain stack and region names, the judgements $\vdash_{\text{type}}$ and $\vdash_{\text{vctxt}}$ require a tower domain $\overline{\mathcal{T}}$.

3.4.5 Surface programs

It is useful to note that the static semantics can be greatly simplified for the surface syntax presented in Figure 1. Tower types and tower domains are purely technical devices that are used to prove type soundness. In the static semantics, they simply collect the names of stacks and regions in scope and assign types to locations. Note that in every rule, tower types and tower domains are passed unmodified to sub-judgements. Since surface programs do not admit syntax for naming stacks and regions, we can type any closed, surface expression with the judgement $\cdot; \cdot; \cdot : \cdot \vdash_{\text{exp}} e : \tau$. Pushing these empty tower types and tower domains through the rules leads to the following simplifications:

$\Delta; \Gamma; \cdot : \cdot \vdash_{\text{exp}} e : \tau \quad \Longrightarrow \quad \Delta; \Gamma \vdash_{\text{exp}} e : \tau$	$\Delta; \cdot \vdash_{\text{type}} \tau \quad \Longrightarrow \quad \Delta \vdash_{\text{type}} \tau$
$\Delta; \cdot \vdash_{\text{vctxt}} \Gamma \quad \Longrightarrow \quad \Delta \vdash_{\text{vctxt}} \Gamma$	$\vdash_{\text{ctxt}} \Delta; \Gamma; \cdot : \cdot \quad \Longrightarrow \quad \vdash_{\text{ctxt}} \Delta; \Gamma$

Hence, we recover a type system equivalent to that of System F, which is sufficient for type-checking surface programs.

Further simplifications can be made by interpreting the monadic commands as constants with polymorphic types. For example, the typing judgements for each of the monadic commands are equivalent to the following typings:

$\mathsf{runRGN} :: \forall\alpha.(\forall\gamma_r.\mathsf{RGNHandle}\ \gamma_r \to \mathsf{RGN}\ \gamma_r\ \alpha) \to \alpha$
$\mathsf{returnRGN} :: \forall\gamma_r.\forall\alpha.\alpha \to \mathsf{RGN}\ \gamma_r\ \alpha$
$\mathsf{thenRGN} :: \forall\gamma_r.\forall\alpha,\beta.\mathsf{RGN}\ \gamma_r\ \alpha \to (\alpha \to \mathsf{RGN}\ \gamma_r\ \beta) \to \mathsf{RGN}\ \gamma_r\ \beta$
$\mathsf{letRGN} :: \forall\gamma_{r_1}.\forall\alpha.(\forall\gamma_{r_2}.\gamma_{r_1} \preceq \gamma_{r_2} \to \mathsf{RGNHandle}\ \gamma_{r_2} \to \mathsf{RGN}\ \gamma_{r_2}\ \alpha) \to \mathsf{RGN}\ \gamma_{r_1}\ \alpha$
$\mathsf{newRGNLoc} :: \forall\gamma_r.\forall\alpha.\mathsf{RGNHandle}\ \gamma_r \to \alpha \to \mathsf{RGN}\ \gamma_r\ (\mathsf{RGNLoc}\ \gamma_r\ \alpha)$
$\mathsf{readRGNLoc} :: \forall\gamma_r.\forall\alpha.\mathsf{RGNLoc}\ \gamma_r\ \alpha \to \mathsf{RGN}\ \gamma_r\ \alpha$

Treating the monadic commands as syntactic forms simplifies the proofs, as there is no need to consider partially applied forms.

3.5 Type Soundness of $\mathsf{F}^{\mathsf{RGN}}$

In this section, we sketch a proof of type soundness. We wish to prove that a well-typed, closed initial program either succeeds (returning a value of the correct type) or diverges. A preservation theorem and a progress theorem make this theorem an easy corollary.

The Preservation Theorem states that the terminating computation of a well-typed expression yields a value of the same type. Because the dynamic semantics are defined by two mutually inductive judgements, the Preservation Theorem also states that the terminating computation of a well-typed command yields a well-typed extension of the top stack and a value of the same type. Various substitution lemmas for dead stacks and regions are required to prove the cases where stacks and regions are deallocated.

Theorem 3.1 (*Preservation*)

(1) If

(a) $\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}$,
(b) $\cdot;\cdot;\mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau$, and
(c) $T; e \hookrightarrow v'$,

then $\cdot;\cdot;\mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v' : \tau$.

(2) If

(a) $\vdash_{\text{tower}} T, s \mapsto S : \mathcal{T}, s \mapsto \mathcal{S} : \overline{\mathcal{T}}, s \mapsto \overline{\mathcal{S}}$,
(b) $\cdot;\cdot;\mathcal{T}, s \mapsto \mathcal{S} : \overline{\mathcal{T}}, s \mapsto \overline{\mathcal{S}} \vdash_{\text{exp}} \kappa^v : \mathsf{RGN}\ s\sharp r\ \tau_a$, and
(c) $T, s \mapsto S; \kappa^v \hookrightarrow_\kappa S'; v'$,

then there exists $\overline{\mathcal{S}}' \sqsupseteq \overline{\mathcal{S}}$ and $\mathcal{S}' \sqsupseteq \mathcal{S}$ such that $\vdash_{\text{tower}} T, s \mapsto S' : \mathcal{T}, s \mapsto \mathcal{S}' : \overline{\mathcal{T}}, s \mapsto \overline{\mathcal{S}}'$ and $\cdot;\cdot;\mathcal{T}, s \mapsto \mathcal{S}' : \overline{\mathcal{T}}, s \mapsto \overline{\mathcal{S}}' \vdash_{\text{exp}} v' : \tau_a$.

Proof
Proceed by mutual induction on the derivations (1c) $T; e \hookrightarrow v'$ and (2c) $T, s \mapsto S; \kappa \hookrightarrow S'; v'$. $\square$

The Progress Theorem states that a partially evaluated expression can always move forward towards complete evaluation. Progress Theorems are notoriously difficult in a large-step operational semantics. Our approach adopts a *natural transition semantics*, introduced in Section 3.3. The Progress Theorem states that any well-typed partial derivation that contains a pending judgement can transition to another well-typed partial derivation. As usual, the proof of the Progress Theorem depends on a Canonical Forms Lemma, which describes the forms of values of particular types.

Definition 3.1

(1) A pending judgement $T; e \hookrightarrow\ ?$ is well typed iff there exists $\overline{\mathfrak{T}}$, $\mathfrak{T}$, and τ such that $\vdash_{\text{tower}} T : \mathfrak{T} : \overline{\mathfrak{T}}$ and $\cdot; \cdot; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} e : \tau$.
(2) A pending judgement $T, s \mapsto S; \kappa \hookrightarrow\ ?$ is well typed iff there exists $\overline{\mathfrak{T}}$, $\mathfrak{T}$, $\overline{\mathfrak{S}}$, $\mathfrak{S}$, $r \in \mathit{dom}(\overline{\mathfrak{S}})$, and τ_a such that $\vdash_{\text{tower}} T, s \mapsto S : \mathfrak{T}, s \mapsto \mathfrak{S} : \overline{\mathfrak{T}}, s \mapsto \overline{\mathfrak{S}}$ and $\cdot; \cdot; \mathfrak{T}, s \mapsto \mathfrak{S} : \overline{\mathfrak{T}}, s \mapsto \overline{\mathfrak{S}} \vdash_{\text{exp}} \kappa : \mathsf{RGN}\ s\sharp r\ \tau_a$.
(3) A partial derivation $\mathfrak{D}$ is well typed iff every pending judgement in it is well typed.

Theorem 3.2 (*Progress*)

If $\mathfrak{D}$ is a well-typed partial derivation with pending judgements, then there exists $\mathfrak{D}'$ such that $\mathfrak{D} \longrightarrow \mathfrak{D}'$ and $\mathfrak{D}'$ is well typed.

Proof

Let $\mathfrak{N}$ be the uppermost node of $\mathfrak{D}$ that is labeled with a pending judgement, either $T; e \hookrightarrow\ ?$ or $T, s \mapsto S; \kappa \hookrightarrow_{\kappa}\ ?$. Any transition on $\mathfrak{D}$ must occur at this node. Proceed by considering all possible forms of pending judgements. $\square$

Theorem 3.3 (*Soundness*)

If $\cdot; \cdot; \cdot : \cdot \vdash_{\text{exp}} e : \tau$, then any execution of e (in $\cdot$) either terminates with a value v (such that $\cdot; \cdot; \cdot : \cdot \vdash_{\text{exp}} v : \tau$) or diverges.

Proof

Let $[\cdot; e \hookrightarrow\ ?]() \longrightarrow \mathfrak{D}_1 \longrightarrow \mathfrak{D}_2 \longrightarrow \cdots$ be an execution of e. Note that $[\cdot; e \hookrightarrow\ ?]()$ is well-typed by $\vdash_{\text{tower}} \cdot : \cdot : \cdot$ and $\cdot; \cdot; \cdot : \cdot \vdash_{\text{exp}} e : \tau$. By Progress, every $\mathfrak{D}_i$ is well typed.

(1) Suppose that for all $\mathfrak{D}_n$ such that $[\cdot; e \hookrightarrow\ ?]() \longrightarrow^* \mathfrak{D}_n$, there exists $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$. Then, e diverges.
(2) Suppose that there exists $\mathfrak{D}_n$ such that $[\cdot; e \hookrightarrow\ ?]() \longrightarrow^* \mathfrak{D}_n$, such that there does not exist $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$.
    (a) Suppose $\mathfrak{D}_n$ contains no pending judgements. By Lemma 3.2, $\mathfrak{D}_n \equiv [\cdot; e \hookrightarrow v]$. Then, e terminates with the value v. By Preservation, $\cdot; \cdot; \cdot : \cdot \vdash_{\text{exp}} v : \tau$.
    (b) Suppose $\mathfrak{D}_n$ contains pending judgements. By Progress, there exists $\mathfrak{D}'$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}'$, contradicting the assumption that there does not exist $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$. Thus, e cannot get stuck. $\square$

Remarks

As stated previously, our main reason for adopting a large-step operational semantics is to simplify the theorems and proofs of Section 5. However, we believe that the technique of proving type soundness for languages described by natural transition semantics shows great promise, particularly for the monadic treatment of effects. In many ways, natural transition semantics attempts to bridge the gap between large-step operational semantics and small-step operational semantics. Natural transition semantics incorporates the advantages of large-step operational semantics (namely, a concise semantics) and ameliorates some of the disadvantages of small-step operational semantics. First, there is no need to introduce intermediate terms to "mark" points of interest in an evaluating program. For example, Semmelroth and Sabry's account of monadic state in ML requires a term $\mathsf{sto}\ \Delta\ e$, to distinguish nested runST evaluations.

Second, there is no need to introduce evaluation contexts. While this may appear to be a minor point (by recognizing that one has effectively defined the evaluation context by the path through a partial derivation tree to a pending judgement), it has broader implications, particularly in the monadic setting. For example, Semmelroth and Sabry's evaluation contexts are quite complex, requiring four separate contexts. This complexity is required to express the relative sequencing of pure and monadic operations; essentially, the contexts must find the $\mathsf{sto}\ \Delta\ []$ that corresponds to the "active" monadic evaluation, then follow commands down to either the "active" monadic command or "active" pure expression. In the natural transition semantics, this is accomplished "automatically" by jumping to the pending judgement of the partial derivation tree. In our case, the fact that this pending judgement can take one of two forms (either a pure call-by-value System F judgement or an imperative monadic judgement), effectively eliminates the need to interleave contexts.

We also believe that the complete soundness proof using natural transition semantics is easier than the corresponding proof using small-step operational semantics (for example, the soundness proof for Cyclone's region system undertaken by the second author (Grossman *et al.*, 2001)). Eliminating intermediate terms and evaluation contexts are obvious savings. The proof flavor is also slightly different: where one was doing case analysis on the form of the active position of an evaluation context, now one is doing case analysis on the pending jugdement's children.

3.6 Extensions

We consider two easy extensions to the language $\mathsf{F}^{\mathsf{RGN}}$: mutable locations and fixed-point locations.

3.6.1 Mutable locations

Figure 12 presents the extensions necessary to support mutable locations. The command $\mathsf{writeRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2$ overwrites the value stored at the location v_1 with the value v_2. All lemmas and theorems can be extended to support mutable locations in a straight-forward manner. In the special case where programs do not contain letRGN, we obtain an alternative proof for the soundness of strict monadic state (Semmelroth & Sabry, 1999).

$$\begin{array}{llll} \text{Surface commands} & \kappa & ::= & \ldots \mid \mathsf{writeRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \\ \text{Computation commands} & \kappa & ::= & \ldots \mid \mathsf{writeRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \end{array}$$

$$\boxed{T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v}$$

$$\frac{\begin{array}{cc} \tau_r \equiv s\sharp r & v_1 \equiv \langle l \rangle_{s\sharp r} \\ r \in dom(S) & l \in dom(S(r)) \end{array}}{T, s \mapsto S; \mathsf{writeRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \hookrightarrow_\kappa S\{(r,l) \mapsto v_2\}; ()}$$

$$\boxed{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} e : \tau}$$

$$\frac{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} v_1 : \mathsf{RGNLoc}\ \tau_r\ \tau_a \qquad \Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} v_2 : \tau_a}{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} \mathsf{writeRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 : 1}$$

Fig. 12. Extensions to $\mathsf{F}^{\mathsf{RGN}}$ for writeRGNLoc

$$\begin{array}{llll} \text{Surface commands} & \kappa & ::= & \ldots \mid \mathsf{fixRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \\ \text{Computation commands} & \kappa & ::= & \ldots \mid \mathsf{fixRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \\ \text{Computation values} & v & ::= & \ldots \mid \Diamond \end{array}$$

$$\boxed{T, s \mapsto S; \kappa \hookrightarrow_\kappa S'; v}$$

$$\frac{\begin{array}{c} \tau_r \equiv s\sharp r \qquad v_1 \equiv \mathsf{handle}(s\sharp r) \\ r \in dom(S) \qquad l \notin T, s \mapsto S \\ T, s \mapsto S\{(r,l) \mapsto \Diamond\}; v_2\ \langle l \rangle_{s\sharp r} \hookrightarrow v' \end{array}}{T, s \mapsto S; \mathsf{fixRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 \hookrightarrow S\{(r,l) \mapsto v'\}, \langle l \rangle_{s\sharp r}}$$

$$\boxed{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} e : \tau}$$

$$\frac{\vdash_{\text{ctxt}} \cdot; \cdot; \mathfrak{T} : \overline{\mathfrak{T}} \qquad \cdot; \overline{\mathfrak{T}} \vdash_{\text{type}} \tau}{\cdot; \cdot; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} \Diamond : \tau}$$

$$\frac{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} v_1 : \mathsf{RGNHandle}\ \tau_r \qquad \Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} v_2 : \mathsf{RGNLoc}\ \tau_r\ \tau_a \to \tau_a}{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} \mathsf{fixRGNLoc}\ [\tau_r]\ [\tau_a]\ v_1\ v_2 : \mathsf{RGN}\ \tau_r\ (\mathsf{RGNLoc}\ \tau_r\ \tau_a)}$$

Fig. 13. Extensions to $\mathsf{F}^{\mathsf{RGN}}$ for fixRGNLoc

3.6.2 Fixed-point locations

Figure 13 presents the extensions necessary to support fixed-point locations. The command fixRGNLoc $[\tau_r]$ $[\tau_a]$ v_1 v_2 allocates a value of type τ_a in the region indexed by τ_r; the value is produced by the function v_2 which is applied to the location where the allocated value is to be stored. This provides a means of allocating recursive functions. (Recursive structures could be accomodated with the addition of recursive types.)

The dynamic semantics for fixRGNLoc make use of a dummy storable value $\Diamond$. The typing rule for $\Diamond$ admits arbitrary well-formed types. (We slightly abuse notation here; $\Diamond$ is not technically an expression form. A $\Diamond$ can only appear in the range of a region.) However, $\Diamond$ is not a value and cannot be the result of any computation. It serves as a place holder in the store, marking the location where the recursive knot will be tied. It also ensures that the tower is well-formed with respect to the extended tower type necessary to prove that $v_2\ \langle l \rangle_{s\sharp r}$ is well-typed.

We note that the typing rule for fixRGNLoc requires that the function v_2 has the type RGNLoc $\tau_r\ \tau_a \to \tau_a$. This is a pure function, not a monadic computation. Hence, it is safe to evaluate with the location bound to $\Diamond$ (where the allocated value is to be stored), because no computation (and, hence, no reading of region allocated values) can occur during the evaluation of the application of v_2 to the location. On the other hand, v_2 can return a (suspended) computation that reads the allocated value, since this computation cannot occur until after the knot has been tied. For example, if h is a variable of type RGNHandle τ_r, then the following expression returns a location of type RGNLoc τ_r (int $\to$ RGN τ_r int), which points to a (monadic) function that evaluates factorials:

```
fixRGNLoc [$\tau_r$] [int → RGN $\tau_r$ int] $h$
            ($\lambda f$:RGNLoc $\tau_r$ (int → RGN $\tau_r$ int).$\lambda n$:int.
             if $n = 0$ then returnRGN $\tau_r$ 1
                         else thenRGN [$\tau_r$] [int → RGN $\tau_r$ int] [int]
                                        (readRGNLoc [$\tau_r$] [int → RGN $\tau_r$ int] $f$)
                                        ($\lambda g$:int → RGN $\tau_r$ int.
                                         thenRGN [$\tau_r$] [int] [int]
                                                   ($g$ ($n - 1$))
                                                   ($\lambda m$:int.returnRGN [$\tau_r$] [int] ($n * m$))))
```

4 Region Calculi

The previous sections have presented $\mathsf{F}^{\mathsf{RGN}}$ and have (hopefully) developed sufficient intuition to reasonably establish our goal of finding a simpler account of region-based type systems. However, while $\mathsf{F}^{\mathsf{RGN}}$ shares some similarities with region calculi (e.g., evaluation with a stack of regions), its type system appears to be quite different from the type-and-effect systems associated with region-based languages. Hence, the skeptical reader may well wonder if the simplicity of the $\mathsf{F}^{\mathsf{RGN}}$ type system points to some deficiency, failing to capture all of the idioms available in type-and-effect systems for region calculi. Our present task is to rigorously demonstrate that we have lost no power in adopting $\mathsf{F}^{\mathsf{RGN}}$; our method for accomplishing this task is to give a type- and meaning-preserving translation from a source language that captures the key aspects of Tofte-Talpin and Cyclone-like region calculi into $\mathsf{F}^{\mathsf{RGN}}$.

Much of the technical complexity in previous and subsequent sections stems from our desire to establish the formal *correctness* of the translation, not simply an intuitive account of the correspondence. Furthermore, as should be clear, there is a large "semantic gap" between the type-and-effect system for a traditional region calculus and the type system for $\mathsf{F}^{\mathsf{RGN}}$. The next subsection sketches the major obstacles to be overcome in the translation. Our conclusion is that the gap is too large to be bridged by a single translation. Instead, we present an Untyped Region Calculus (URC) in Section 4.2, which provides a core syntax and dynamic semantics for a typical region calculus. Sections 4.3–4.5 present a succession of type-systems for URC. The first is a Traditional Region Calculus (TRC), which corresponds directly to type-and-effect systems given in the literature (Helsen & Thiemann, 2000; Calcagno, 2001; Calcagno *et al.*, 2002). The second is the Bounded Region Calculus (BRC), which augments TRC with a form of bounded region polymorphism. The Bounded Region Calculus can be seen as a core model of early versions of Cyclone (Grossman *et al.*, 2001; Grossman *et al.*, 2002). Finally, the Single Effect Calculus (SEC) restricts BRC by admitting only a single region as the latent effect of an expression. Hence, our roadmap is as follows:

Untyped Region Calculus (URC)
$\rightsquigarrow$ Traditional Region Calculus (TRC)
$\rightsquigarrow$ Bounded Region Calculus (BRC)
$\rightsquigarrow$ Single Effect Calculus (SEC)

Type- and meaning-preserving translations down this chain are relatively straightforward (meaning-preservation following directly from the shared dynamic semantics), and will presented as succinctly as possible. After presenting the Single Effect Calculus, it should be clear that there is a much greater correspondence between the type system for SEC and $\mathsf{F}^{\mathsf{RGN}}$ than between the type system for TRC and $\mathsf{F}^{\mathsf{RGN}}$. However, the translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ contains some subtleties, and will be covered in full detail in Section 5.

4.1 From TRC to SEC to $\mathsf{F}^{\mathsf{RGN}}$: Translation Sketch

The form of the function type and the form of the expression typing judgement are defining characteristics of "traditional" type-and-effect systems, such as that to be given for the Traditional Region Calculus:

$$\tau_1 \xrightarrow{\varphi} \tau_2 \qquad\qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$$

In a region-based language, an effect denotes a set of regions. In the function type, the effect φ is a *latent effect*: a (super)set of those regions read from or written to when the function is called. In the expression typing judgement, the effect φ describes the regions affected when the expression evaluates.

A typed call-by-value monad translation (Wadler, 1995; Sabry & Wadler, 1997; Moggi & Sabry, 2001; Wadler & Thiemann, 2003) suggests translating the type $\tau_1 \xrightarrow{\varphi} \tau_2$ to $\mathbb{T}[\![\tau_1]\!] \to \mathsf{RGN}\ \mathbb{T}[\![\varphi]\!]\ \mathbb{T}[\![\tau_2]\!]$ and translating the judgement $\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$ to an expression of type $\mathsf{RGN}\ \mathbb{T}[\![\varphi]\!]\ \mathbb{T}[\![\tau]\!]$. The difficulty with this translation is that φ naturally denotes a set of regions, while τ_r in $\mathsf{RGN}\ \tau_r\ \tau_a$ only names a single region.

Hence, we would be better served by representing effects by a single region; but, which region? The key insight is that a LIFO stack of regions imposes a partial order on live (allocated) regions. Older regions (lower on the stack) outlive younger regions (higher on the stack). Hence, the liveness of a region implies the liveness of all regions below it on the stack. Alternatively, we consider a region to be a subtype of all the region that it outlives. Thus, it is the case that a single region can serve as a witness for a set of effects: the region appears as a *single effect* in place of the set. This will be the defining characteristic of the Single Effect Calculus.

To bridge the gap between the type-and-effect systems of the Traditional Region Calculus and the Single Effect Calculus, we take inspiration from Cyclone (Grossman *et al.*, 2001). One key difference (among many) between Cyclone and the Tofte-Talpin region calculus is that the type-and-effects system of Cyclone extends that of Tofte-Talpin's with a form of bounded region polymorphism. The abstraction of a region variable ϱ may be bounded by a set of regions φ. At the instantiation of ϱ by a region ρ, we must show that the liveness of ρ implies the liveness of all the regions in φ. Within the body of the abstraction, we may assume that ϱ is an upper bound on the set of regions φ. However, like the Tofte-Talpin region calculus, Cyclone treats effects as sets of regions affected by the evaluation of an expression. The Bounded Region Calculus will combine these traits by admitting both latent effects as sets of regions *and* bounded region polymorphism.

Hence, a translation from a source region calculus to $\mathsf{F}^{\mathsf{RGN}}$ must accomplish a number of objectives: (1) eliminate region subtyping (through explicit coercions), (2) sequence computations using the monadic constructs, and (3) encode effects using a single region for the index of the RGN monad. In order to simplify the translation to $\mathsf{F}^{\mathsf{RGN}}$ and its proof of correctness, we factor out this third objective in the remainder of this section by first sketching a translation to the Single Effect Calculus. In terms of translating types and typing judgements, our path is as follows:

$$\begin{array}{ccccc}
\text{TRC/BRC} & \rightsquigarrow^{(3)} & \text{SEC} & \rightsquigarrow^{(1,2)} & \mathsf{F}^{\mathsf{RGN}} \\[1em]
\tau_1 \xrightarrow{\varphi} \tau_2 & \rightsquigarrow^{(3)} & \Pi\varrho \succeq \varphi.\tau_1 \xrightarrow{\varrho} \tau_2 & \rightsquigarrow^{(1,2)} & \Lambda\varrho.(\mathbb{T}[\![\varrho \succeq \varphi]\!] \rightarrow \mathbb{T}[\![\tau_1]\!] \rightarrow \mathsf{RGN}\ \mathbb{T}[\![\varrho]\!]\ \ \mathbb{T}[\![\tau_1]\!]) \\[1em]
\Delta;\Gamma \vdash_{\text{exp}} e : \tau, \varphi & \rightsquigarrow^{(3)} & \begin{array}{c}\Delta;\Gamma \vdash_{\text{exp}} e : \tau, \theta \\ \text{where } \theta \text{ bounds } \varphi\end{array} & \rightsquigarrow^{(1,2)} & \mathbb{D}[\![\Delta]\!]\,;\mathbb{G}[\![\Gamma]\!] \vdash_{\text{exp}} \mathbb{E}[\![e]\!] : \mathsf{RGN}\ \mathbb{T}[\![\theta]\!]\ \ \mathbb{T}[\![\tau]\!]
\end{array}$$

4.2 An Untyped Region Calculus

This section presents an Untyped Region Calculus (URC), which is a variation of the region calculus of Tofte and Talpin (1994; 1997), in the spirit of more recent direct presentations of region calculi (Helsen & Thiemann, 2000; Calcagno, 2001; Calcagno *et al.*, 2002; Henglein *et al.*, 2005). This calculus will provide core syntax and dynamic semantics for the subsequent type systems.

4.2.1 Syntax of URC

Figure 14 presents the syntax of programs in the Untyped Region Calculus.

As in $\mathsf{F}^{\mathsf{RGN}}$ we distinguish between region variables, region names, and a deallocated region $\bullet$. For an external language, it suffices to allows region placeholders to range over region variables (*RVars*), which include a distinguished member $\mathcal{H}$, corresponding to a global region that remains live (allocated) throughout the execution of the program. Region names and locations are used to represent pointers to region allocated data.

Terms are similar to those found in the λ-calculus. One major difference is that terms yielding heap allocated values carry a region annotation $\mathsf{at}\ \rho$, which indicates the region in which the value is to be allocated. We assume that integers, pairs, and function closures require heap allocated storage, while booleans do not. New

$$
\begin{array}{rcl}
i & \in & \mathbb{Z} \\
l & \in & \mathit{Locations} \\
r & \in & \mathit{RNames} \qquad \text{where } \mathsf{H} \in \mathit{RNames} \\
\vartheta, \varrho & \in & \mathit{RVars} \qquad \text{where } \mathcal{H} \in \mathit{RVars} \\
f, x & \in & \mathit{Vars}
\end{array}
$$

Region placeholders

$$\theta, \rho \ ::= \ \varrho \mid r \mid \bullet$$

Effects

$$\varphi \ ::= \ \{\rho_1, \ldots, \rho_n\}$$

Terms

$$
\begin{array}{rcl}
e & ::= & i\ \mathsf{at}\ \rho \mid e_1 \oplus e_2\ \mathsf{at}\ \rho \mid e_1 \otimes e_2 \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \mid \\
 & & x \mid \lambda x.e\ \mathsf{at}\ \rho \mid e_1\ e_2 \mid (e_1, e_2)\ \mathsf{at}\ \rho \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \\
 & & \mathsf{letregion}\ \varrho\ \mathsf{in}\ e \mid \lambda\varrho.u\ \mathsf{at}\ \rho \mid e\ [\rho] \mid \\
 & & \langle l \rangle_r \mid \langle l \rangle_\bullet
\end{array}
$$

Abstractions

$$u \ ::= \ \lambda x.e\ \mathsf{at}\ \rho \mid \lambda\varrho.u\ \mathsf{at}\ \rho$$

Values

$$v \ ::= \ \mathsf{tt} \mid \mathsf{ff} \mid x \mid \langle l \rangle_r \mid \langle l \rangle_\bullet$$

Storable values

$$w \ ::= \ i \mid \lambda x.e \mid (v_1, v_2) \mid \lambda\varrho.u$$

Regions

$$R \ ::= \ \{l_1 \mapsto w_1, \ldots, l_n \mapsto w_n\}$$

Region stacks / Stacks

$$S \ ::= \ \cdot \mid S, r \mapsto R \quad \text{(ordered domain)}$$

Fig. 14. Syntax of URC

regions are introduced (and implicitly created and destroyed) by the $\mathsf{letregion}\ \varrho\ \mathsf{in}\ e$ term. The region variable ϱ is bound within e, demarcating the scope of the region. Within e, values may be read from or allocated in the region ϱ.

The term $\lambda\varrho.u\ \mathsf{at}\ \rho$ introduces a region abstraction (allocated in the region ρ), where the term u is polymorphic in the region ϱ.[3] Such region polymorphism is particularly useful in the definition of functions, in which we parameterize over the regions necessary for the evaluation of the function. The term $e\ [\rho]$ eliminates a region abstraction; operationally, it substitutes the place ρ for the region variable ϱ in u and evaluates the resulting term.

The expression $\langle l \rangle_r$ is the (live) pointer associated with a region allocated value. Likewise, the expression $\langle l \rangle_\bullet$ is the is the (dangling) pointer associated with a region deallocated value.

Because the introduction forms for region allocated values are not themselves values, we formalize the syntactic class of storable values. Storable values are associated with locations in regions R and regions are ordered into stacks S. Intuitively,

[3] Limiting the body of a region abstraction to abstractions ensures that an erasure function that removes region annotations and produces a λ-calculus term is meaning preserving.

$$\boxed{S; e \hookrightarrow v}$$

$$\frac{r \in dom(S) \qquad l \notin S}{S; i \text{ at } r \hookrightarrow S\{(r,l) \mapsto i\}; \langle l \rangle_r}
\qquad
\frac{\begin{array}{cc} S; e_1 \hookrightarrow S_1; \langle l_1 \rangle_{r_1} & S_1(r_1, l_1) = i_1 \\ S_1; e_2 \hookrightarrow S_2; \langle l_2 \rangle_{r_2} & S_2(r_2, l_2) = i_2 \\ r \in dom(S_2) \quad l \notin S_2 & i_1 \oplus i_2 = i \end{array}}{S; e_1 \oplus e_2 \text{ at } r \hookrightarrow S_2\{(r,l) \mapsto i\}; \langle l \rangle_r}$$

$$\frac{\begin{array}{cc} S; e_1 \hookrightarrow S_1; \langle l_1 \rangle_{r_1} & S_1(r_1, l_1) = i_1 \\ S_1; e_2 \hookrightarrow S_2; \langle l_2 \rangle_{r_2} & S_2(r_2, l_2) = i_2 \\ \multicolumn{2}{c}{i_1 \olessthan i_2 = b} \end{array}}{S; e_1 \olessthan e_2 \text{ at } \varrho \hookrightarrow S_2; b}
\qquad
\frac{}{S; \mathsf{tt} \hookrightarrow S; \mathsf{tt}}
\qquad
\frac{}{S; \mathsf{ff} \hookrightarrow S; \mathsf{ff}}$$

$$\frac{S; e_b \hookrightarrow S'; \mathsf{tt} \qquad S'; e_t \hookrightarrow S''; v''}{S; \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \hookrightarrow S''; v''}
\qquad
\frac{S; e_b \hookrightarrow S'; \mathsf{ff} \qquad S'; e_f \hookrightarrow S''; v''}{S; \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \hookrightarrow S''; v''}$$

$$\frac{r \in dom(S) \qquad l \notin S}{S; \lambda x.e' \text{ at } r \hookrightarrow S\{(r,l) \mapsto \lambda x.e'\}; \langle l \rangle_r}
\qquad
\frac{\begin{array}{cc} S; e_1 \hookrightarrow S_1; \langle l_1 \rangle_{r_1} & S_1(r_1, l_1) = \lambda x.e' \\ S_1; e_2 \hookrightarrow S_2; v_2 & S_2; e'[v_2/x] \hookrightarrow S_3; v_3 \end{array}}{S; e_1\ e_2 \hookrightarrow S_3; v_3}$$

$$\frac{\begin{array}{cc} S; e_1 \hookrightarrow S_1; v_1 & S_1; e_2 \hookrightarrow S_2; v_2 \\ r \in dom(S_2) & l \notin S_2 \end{array}}{S; (e_1, e_2) \text{ at } r \hookrightarrow S_2\{(r,l) \mapsto (v_1, v_2)\}; \langle l \rangle_r}
\qquad
\frac{\begin{array}{c} S; e \hookrightarrow S'; \langle l \rangle_r \\ S'(r,l) = (v_1, v_2) \end{array}}{S; \mathsf{fst}\ e \hookrightarrow S'; v_1}
\qquad
\frac{\begin{array}{c} S; e \hookrightarrow S'; \langle l \rangle_r \\ S'(r,l) = (v_1, v_2) \end{array}}{S; \mathsf{snd}\ e \hookrightarrow S'; v_2}$$

$$\frac{r \in dom(S) \qquad l \notin S}{S; \lambda \varrho.u' \text{ at } r \hookrightarrow S\{(r,l) \mapsto \lambda \varrho.u'\}; \langle l \rangle_r}
\qquad
\frac{\begin{array}{cc} S; e_1 \hookrightarrow S_1; \langle l_1 \rangle_{r_1} & S_1(r_1, l_1) = \lambda \varrho.u' \\ \multicolumn{2}{c}{S_1; u'[\rho_2/\varrho] \hookrightarrow S_2; v_2} \end{array}}{S; e_1\ [\rho_2] \hookrightarrow S_2; v_2}$$

$$\frac{r \notin S \qquad S, r \mapsto \{\}; e[r/\varrho] \hookrightarrow S', r \mapsto R'; v'}{S; \mathsf{letregion}\ \varrho\ \mathsf{in}\ e \hookrightarrow S'[\bullet/r]; v'[\bullet/r]}$$

$$\boxed{e \hookrightarrow_{\text{prog}} v}$$

$$\frac{\cdot, \mathsf{H} \mapsto \{\}; e[\mathsf{H}/\mathcal{H}] \hookrightarrow \cdot; \mathsf{H} \mapsto R'; v'}{e \hookrightarrow_{\text{prog}} v'[\bullet/\mathsf{H}]}$$

Fig. 15. Dynamic semantics of URC

evaluating a letregion expression adds a new region to the top of the stack (which is deallocated upon finishing the expression).

4.2.2 Dynamic semantics of URC

An inductive judgement (Figure 15) defines the dynamic semantics of the Untyped Region Calculus. We state without proof that the dynamic semantics is deterministic.

The judgement $S; e \hookrightarrow S'; v'$ asserts that evaluating the closed expression e in stack S results in a new stack S' and a value v'. Note that the rules for $S; e \hookrightarrow S'; v'$ thread the modified stack through each expression evaluation, imposing a left-to-right evaluation order. Consider, for example, the following rule:

$$\frac{\begin{array}{cc} S; e_1 \hookrightarrow S_1; \langle l_1 \rangle_{r_1} & S_1(r_1, l_1) = i_1 \\ S_1; e_2 \hookrightarrow S_2; \langle l_2 \rangle_{r_2} & S_2(r_2, l_2) = i_2 \\ r \in dom(S_2) \quad l \notin S_2 & i_1 \oplus i_2 = i \end{array}}{S; e_1 \oplus e_2 \text{ at } r \hookrightarrow S_2\{(r,l) \mapsto i\}; \langle l \rangle_r}$$

The first line evaluates e_1 to a live location and reads out the integer stored at

$$\text{Region contexts} \quad \Delta \quad ::= \quad \cdot \mid \Delta, \varrho$$

$$\boxed{\Delta;\Gamma \vdash_{\mathrm{exp}} e : \tau, \varphi}$$

$$\frac{\Delta;\Gamma, x : \tau_1 \vdash_{\mathrm{exp}} e' : \tau_2, \varphi' \qquad \Delta \vdash_{\mathrm{place}} \rho}{\Delta;\Gamma \vdash_{\mathrm{exp}} \lambda x : \tau_1.^{\varphi'} e' \ \mathsf{at}\ \rho : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho), \{\rho\}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_1'), \varphi_1 \qquad \Delta;\Gamma \vdash_{\mathrm{exp}} e_2 : \tau_1, \varphi_2}{\Delta;\Gamma \vdash_{\mathrm{exp}} e_1\ e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \{\rho_1'\} \cup \varphi'}$$

$$\frac{\Delta \vdash_{\mathrm{type}} \tau \qquad \vdash_{\mathrm{ctxt}} \Delta;\Gamma;(\varphi \setminus \varrho) \qquad \Delta, \varrho;\Gamma \vdash_{\mathrm{exp}} e : \tau, \varphi}{\Delta;\Gamma \vdash_{\mathrm{exp}} \mathsf{letregion}\ \varrho\ \mathsf{in}\ e : \tau, \varphi \setminus \varrho}$$

$$\frac{\Delta, \varrho;\Gamma \vdash_{\mathrm{exp}} u' : \tau, \varphi' \qquad \Delta \vdash_{\mathrm{place}} \rho}{\Delta;\Gamma \vdash_{\mathrm{exp}} \lambda \varrho.^{\varphi'} u' \ \mathsf{at}\ \rho : (\Pi \varrho.^{\varphi'} \tau, \rho), \{\rho\}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{exp}} e : (\Pi \varrho.^{\varphi'} \tau, \rho_1'), \varphi_1 \qquad \Delta \vdash_{\mathrm{place}} \rho_2}{\Delta;\Gamma \vdash_{\mathrm{exp}} e\ [\rho_2] : \tau[\rho_2/\varrho], \varphi \cup \{\rho_1'\} \cup \varphi'[\rho_2/\varrho]}$$

Fig. 16. Static semantics of TRC (abbreviated)

$\langle l_1 \rangle_{r_1}$. Likewise, the second line evaluates e_2 to a live location and reads out the integer stored at $\langle l_2 \rangle_{r_2}$. Finally, a fresh location in the region r is chosen, and the final stack with the computed integer stored at the freshly chosen location and the location are returned. The other rules work in much the same manner.

The rule for letregion introduces (and subsequently eliminates) a new region. Its execution is similar to that of letRGN.

Finally, there is a special rule for the evaluation of surface programs. Programs in the Untyped Region Calculus are simply terms. We distinguish programs because the type systems presented in the next sections have a special judgement for top-level programs. Essentially, this judgement establishes reasonable "boundary conditions" for a program's execution, an aspect that is often overlooked in other descriptions of region calculi. Programs are evaluated under a stack with a distinguished region H, which is substituted for the region variable $\mathcal{H}$ during the evaluation of the program. Essentially, one can consider the evaluation of a program e as being equivalent to the evaluation of the expression letregion $\mathcal{H}$ in e, where the final stack is discarded.

4.3 A Traditional Region Calculus

Figure 16 gives an abbreviated static semantics for a traditional region calculus (Helsen & Thiemann, 2000; Calcagno, 2001; Calcagno *et al.*, 2002) applied to the syntax of the Untyped Region Calculus. The type structure is as follows:

Types			Boxed types		
τ	$::=$	$\mathsf{bool} \mid (\mu, \rho)$	μ	$::=$	$\mathsf{int} \mid \tau_1 \xrightarrow{\varphi'} \tau_2 \mid \tau_1 \times \tau_2 \mid \Pi \varrho.^{\varphi'} \tau$

A region is associated with every type that requires heap allocated storage. The type (μ, ρ) pairs together a boxed type (a type requiring heap allocated storage) and a region placeholder; we interpret (μ, ρ) as the type of values of boxed type μ allocated in region ρ. The judgement $\Delta \vdash_{\text{place}} \rho$ checks that ρ is well-formed in the region context Δ.

Note that the typing rules rely upon set theoretic operations ($\in$, $\cup$, and $\setminus$) to check and synthesize effects. As our translation to $\mathsf{F}^{\mathsf{RGN}}$ will require witnessing effect subsumption by explicit coercions, the Bounded Region Calculus and Single Effect Calculus will formalize these relations in separate judgements.

4.4 The Bounded Region Calculus

In this section, we sketch the Bounded Region Calculus (BRC), which can be seen as a core model of Cyclone. The type structure for BRC is as follows:

$$\begin{array}{lll}\text{Types} & & \text{Boxed types} \\ \tau ::= \mathsf{bool} \mid (\mu, \rho) & & \mu ::= \mathsf{int} \mid \tau_1 \xrightarrow{\varphi'} \tau_2 \mid \tau_1 \times \tau_2 \mid \Pi\varrho \succeq \varphi.^{\varphi'}\tau\end{array}$$

In a region-abstraction type $\Pi\varrho \succeq \varphi.^{\varphi'}\tau$, the effect φ serves as a lower bound on the lifetime of the region ϱ. (Note that the region variable ϱ is bound within φ' and τ, but not φ.) The abstraction can only be instantiated by a region ρ that has been pushed on the stack more recently than those regions in φ. Within the body of the abstraction, we may safely assume that ϱ is outlived by all of the regions in φ. Put another way, if ϱ is live, then all of the regions in φ must be live.

Figure 17 gives an abbreviated static semantics for the Bounded Region Calculus. Region contexts Δ are ordered lists of region variables bounded by effect sets. We summarize the main typing judgements in the following table:

Judgement	Meaning
$\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1$	If region ρ_2 is live, then region ρ_1 is live. (Alt.: region ρ_1 outlives region ρ_2.)
$\Delta \vdash_{\text{re}} \rho \succeq \varphi$	If region ρ is live, then all regions in φ are live. (Alt.: all regions in φ outlive region ρ.)
$\Delta \vdash_{\text{er}} \varphi \ni \rho$	Region ρ is a region in φ.
$\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi'$	All region in φ' are regions in φ.
$\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \varphi$	Term e has type τ and effect φ.

We note that the typing rules for the judgements $\vdash_{\text{rr}}$ and $\vdash_{\text{re}}$ simply formalize the reflexive, transitive closure of the syntactic constraints in Δ, each of which asserts a particular "outlived by" relation between a region variable and an effect set. Likewise, the judgements $\vdash_{\text{er}}$ and $\vdash_{\text{rr}}$ formalize the set theoretic operations used by the traditional region calculus. The $\Delta \vdash_{\text{place}} \rho$ and $\Delta \vdash_{\text{eff}} \varphi$ judgements check that ρ and φ, respecitvely, are well-formed in the region context Δ.

The typing rule for $\mathsf{letregion}\ \varrho\ \mathsf{in}\ e$ relates the new region to the currently live regions by introducing ϱ into the region context with an appropriate bound: while ϱ is live, all regions in $\{\rho_1, \ldots, \rho_n\}$ are live. The typing rule for region instantiation requires that we be able to show that the formal region parameter ρ_2 is outlived by all of the regions in the region abstraction bound φ''.

$$\text{Region contexts}\quad \Delta \quad ::= \quad \cdot \mid \Delta, \varrho \succeq \varphi$$

$$\boxed{\Delta \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1}$$

$$\frac{(\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta \vdash_{\mathrm{rr}} \varrho \succeq \rho_i} \qquad \frac{\Delta \vdash_{\mathrm{place}} \rho}{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho} \qquad \frac{\Delta \vdash_{\mathrm{rr}} \rho_2 \succeq \rho' \qquad \Delta \vdash_{\mathrm{rr}} \rho' \succeq \rho_1}{\Delta \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1}$$

$$\boxed{\Delta \vdash_{\mathrm{re}} \rho \succeq \varphi} \qquad \boxed{\Delta \vdash_{\mathrm{er}} \varphi \ni \rho} \qquad \boxed{\Delta \vdash_{\mathrm{ee}} \varphi \supseteq \varphi'}$$

$$\frac{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\mathrm{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}} \qquad \frac{\Delta \vdash_{\mathrm{eff}} \{\rho_1, \ldots, \rho_n\}}{\Delta \vdash_{\mathrm{er}} \{\rho_1, \ldots, \rho_n\} \ni \rho_i} \qquad \frac{\begin{array}{c}\Delta \vdash_{\mathrm{eff}} \varphi \\ \Delta \vdash_{\mathrm{er}} \varphi \ni \rho_i \quad {}^{i \in 1 \ldots n}\end{array}}{\Delta \vdash_{\mathrm{ee}} \varphi \supseteq \{\rho_1, \ldots, \rho_n\}}$$

$$\boxed{\Delta; \Gamma \vdash_{\mathrm{exp}} e : \tau, \varphi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma, x : \tau_1 \vdash_{\mathrm{exp}} e' : \tau_2, \varphi' \\ \Delta \vdash_{\mathrm{place}} \rho \qquad \Delta \vdash_{\mathrm{er}} \varphi \ni \rho\end{array}}{\Delta; \Gamma \vdash_{\mathrm{exp}} \lambda x : \tau_1 .^{\varphi'} e' \;\mathsf{at}\; \rho : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho), \varphi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_1'), \varphi \qquad \Delta \vdash_{\mathrm{er}} \varphi \ni \rho_1' \\ \Delta; \Gamma \vdash_{\mathrm{exp}} e_2 : \tau_1, \varphi \qquad \Delta \vdash_{\mathrm{ee}} \varphi \supseteq \varphi'\end{array}}{\Delta; \Gamma \vdash_{\mathrm{exp}} e_1 \; e_2 : \tau_2, \varphi}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{type}} \tau \qquad \vdash_{\mathrm{ctxt}} \Delta; \Gamma; \{\rho_1, \ldots, \rho_n\} \\ \Delta, \varrho \succeq \{\rho_1, \ldots, \rho_n\}; \Gamma \vdash_{\mathrm{exp}} e : \tau, \{\rho_1, \ldots, \rho_n, \varrho\}\end{array}}{\Delta; \Gamma \vdash_{\mathrm{exp}} \mathsf{letregion}\; \varrho \;\mathsf{in}\; e : \tau, \{\rho_1, \ldots, \rho_n\}}$$

$$\frac{\begin{array}{c}\Delta, \varrho \succeq \varphi''; \Gamma \vdash_{\mathrm{exp}} u' : \tau, \varphi' \\ \Delta \vdash_{\mathrm{place}} \rho \qquad \Delta \vdash_{\mathrm{er}} \varphi \ni \rho\end{array}}{\Delta; \Gamma \vdash_{\mathrm{exp}} \lambda \varrho \succeq \varphi'' .^{\varphi'} u' \;\mathsf{at}\; \rho : (\Pi \varrho \succeq \varphi'' .^{\varphi'} \tau, \rho), \varphi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{exp}} e : (\Pi \varrho \succeq \varphi'' .^{\varphi'} \tau, \rho_1'), \varphi \qquad \Delta \vdash_{\mathrm{er}} \varphi \ni \rho_1' \\ \Delta \vdash_{\mathrm{place}} \rho_2 \qquad \Delta \vdash_{\mathrm{re}} \rho_2 \succeq \varphi'' \qquad \Delta \vdash_{\mathrm{ee}} \varphi \supseteq \varphi'[\rho_2/\varrho]\end{array}}{\Delta; \Gamma \vdash_{\mathrm{exp}} e \; [\rho_2] : \tau[\rho_2/\varrho], \varphi}$$

Fig. 17. Static semantics of BRC (abbreviated)

Translation of TRC to BRC

There is a trivial translation from the traditional region calculus into the Bounded Region Calculus, whereby every region abstraction becomes a region abstraction with an empty bound. Meaning preservation is trivial, as the languages share the same dynamic semantics. Type preservation corresponds to the validity of effect enlargement.

Lemma 4.1 (*Translation Preserves Types*)
(1) If $\Delta; \Gamma \vdash^{\mathrm{TRC}}_{\mathrm{exp}} e : \tau, \varphi$, then forall and φ',
if $\vdash^{\mathrm{BRC}}_{\mathrm{ctxt}} \tilde{\mathbb{D}}[\![\Delta]\!] \,; \tilde{\mathbb{G}}[\![\Gamma]\!] \,; \varphi'$ and $\tilde{\mathbb{D}}[\![\Delta]\!] \vdash^{\mathrm{BRC}}_{\mathrm{ee}} \varphi' \supseteq \varphi$,
then $\tilde{\mathbb{D}}[\![\Delta]\!] \,; \tilde{\mathbb{G}}[\![\Gamma]\!] \vdash^{\mathrm{BRC}}_{\mathrm{exp}} \tilde{\mathbb{E}}[\![e]\!] : \tilde{\mathbb{T}}[\![\tau]\!] \,, \varphi'$.
(2) If $\vdash^{\mathrm{TRC}}_{\mathrm{prog}} p \;\mathsf{ok}$, then $\vdash^{\mathrm{BRC}}_{\mathrm{prog}} \tilde{\mathbb{E}}[\![p]\!] \;\mathsf{ok}$.

Region contexts	Δ	$::=$	$\cdot \mid \Delta, \varrho \succeq \varphi$	
Value contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$	
Region domains	$\overline{\mathcal{R}}$	$::=$	$\{l_1, \ldots, l_n\}$	
Region types	$\mathcal{R}$	$::=$	$\{l_1 \mapsto \mu_1, \ldots, l_n \mapsto \mu_n\}$	
Stack domains	$\overline{\mathcal{S}}$	$::=$	$\cdot \mid \overline{\mathcal{S}}, r \mapsto \overline{\mathcal{R}}$	(ordered domain)
Stack types	$\mathcal{S}$	$::=$	$\cdot \mid \mathcal{S}, r \mapsto \mathcal{R}$	(ordered domain)

Fig. 18. Static semantics of SEC (definitions)

4.5 The Single Effect Calculus

The Single Effect Calculus (SEC) can be seen as a restricted form of the Bounded Region Calculus, where latent effects consist of a *single region* instead of a set of regions. As a convention, we will use θ to represent regions that correspond to such effects. Hence, the type structure is as follows:

Types	Boxed types
$\tau \;::=\; \mathsf{bool} \mid (\mu, \rho)$	$\mu \;::=\; \mathsf{int} \mid \tau_1 \xrightarrow{\theta} \tau_2 \mid \tau_1 \times \tau_2 \mid \Pi\varrho \succeq \varphi.^{\theta}\tau$

Because the Single Effect Calculus will be the source of our translation into $\mathsf{F}^{\mathsf{RGN}}$, we present the static semantics in more detail than the previous region calculi.

4.5.1 Definitions

Figure 18 present additional definitions for syntactic classes that appear in the static semantics. As in BRC, region contexts Δ are ordered lists of region variables bounded by effect sets. Stack and region types and domains, as in $\mathsf{F}^{\mathsf{RGN}}$, serve a technical purpose in the proof of type-preservation under evaluation.

The typing rules for the Single Effect Calculus appear in the following figures.

4.5.2 Expressions

Figure 19 present the typing rules for the judgement $\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e : \tau, \theta$, which asserts that under region context Δ, value context Γ, and stack type $\mathcal{S}$ with stack domain $\overline{\mathcal{S}}$, the expression e has type τ and effects bounded by the region θ. In practice, and as suggested by the typing rules, θ usually corresponds to the most recently allocated region (also referred to as the top or current region).

Figure 20 reproduces the $\vdash_{\mathrm{rr}}$ and $\vdash_{\mathrm{re}}$ judgements of the Bounded Region Calculus, adding constraints implied by $\overline{\mathcal{S}}$, which asserts "outlived by" relations by explicit ordering of region names.

As in BRC, the typing rule for $\mathsf{letregion}\ \varrho\ \mathsf{in}\ e$ relates the new region to the currently live regions by introducing ϱ into the region context with an appropriate bound. In particular, the new region is outlived by the "old" current region and becomes the "new" current region for the evaluation of the body of the $\mathsf{letregion}$.

The judgements for locations ensure that region names that appear in locations are in scope; furthermore, a location in a live region points to a value with the type assigned by the stack type.

$$\boxed{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e:\tau,\theta}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} i\ \mathsf{at}\ \rho:(\mathsf{int},\rho),\theta}
\qquad
\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1:(\mathsf{int},\rho_1),\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_1\\ \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_2:(\mathsf{int},\rho_2),\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_2\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1\oplus e_2\ \mathsf{at}\ \rho:(\mathsf{int},\rho),\theta}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1:(\mathsf{int},\rho_1),\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_1\\ \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_2:(\mathsf{int},\rho_2),\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_2\end{array}}{\Delta;\Gamma\vdash_{\mathrm{exp}} e_1\circledcirc e_2:\mathsf{bool},\theta}
\qquad
\frac{\vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\mathsf{tt}:\mathsf{bool},\theta}$$

$$\frac{\vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\mathsf{ff}:\mathsf{bool},\theta}
\qquad
\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_b:\mathsf{bool},\theta\\ \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_t:\tau,\theta\qquad \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_f:\tau,\theta\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f:\tau,\theta}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta\\ x\in dom(\Gamma)\qquad \Gamma(x)=\tau\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} x:\tau,\theta}
\qquad
\frac{\begin{array}{c}\Delta;\Gamma,x:\tau_1;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e':\tau_2,\theta'\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\lambda x:\tau_1.^{\theta'}e'\ \mathsf{at}\ \rho:(\tau_1\xrightarrow{\theta'}\tau_2,\rho),\theta}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1:(\tau_1\xrightarrow{\theta'}\tau_2,\rho_1'),\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_1'\\ \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_2:\tau_1,\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\theta'\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1\ e_2:\tau_2,\theta}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1:\tau_1,\theta\\ \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_2:\tau_2,\theta\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}(e_1,e_2)\ \mathsf{at}\ \rho:(\tau_1\times\tau_2,\rho),\theta}
\qquad
\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e:(\tau_1\times\tau_2,\rho),\theta\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\mathsf{fst}\ \ e:\tau_1,\theta}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e:(\tau_1\times\tau_2,\rho),\theta\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\mathsf{snd}\ \ e:\tau_2,\theta}
\qquad
\frac{\begin{array}{c}\Delta;\overline{\mathcal{S}}\vdash_{\mathsf{type}}\tau\qquad \vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta\\ \Delta,\varrho\succeq\{\theta\};\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e:\tau,\varrho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\mathsf{letregion}\ \varrho\ \mathsf{in}\ e:\tau,\theta}$$

$$\frac{\begin{array}{c}\Delta,\varrho\succeq\varphi;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} u':\tau,\theta'\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\lambda\varrho\succeq\varphi.^{\theta'}u'\ \mathsf{at}\ \rho:(\Pi\varrho\succeq\varphi.^{\theta'}\tau,\rho),\theta}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1:(\Pi\varrho\succeq\varphi.^{\theta'}\tau,\rho_1'),\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_1'\\ \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho_2\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{re}}\rho_2\succeq\varphi\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\theta'[\rho_2/\varrho]\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1\ [\rho_2]:\tau[\rho_2/\varrho],\theta}$$

$$\frac{\vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta\qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{btype}}\mu}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\langle l\rangle_\bullet:(\mu,\bullet),\theta}$$

$$\frac{\vdash_{\mathrm{ctxt}}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta\qquad r\in dom(\overline{\mathcal{S}})\qquad l\in dom(\overline{\mathcal{S}}(r))\qquad \mu=\mathcal{S}(r,l)}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}}\langle l\rangle_r:(\mu,r),\theta}$$

Fig. 19. Static semantics of SEC (expressions)

$$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1}$$

$$\frac{\begin{array}{c}\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \\ (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta\end{array}}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \varrho \succeq \rho_i} \qquad \frac{\begin{array}{c}\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \\ \overline{\mathcal{S}} = \overline{\mathcal{S}}_1, r_1 \mapsto \overline{\mathcal{R}}_1, \overline{\mathcal{S}}_2, r_2 \mapsto \overline{\mathcal{R}}_2, \overline{\mathcal{S}}_3\end{array}}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} r_2 \succeq r_1}$$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} r \succeq \{r_1, \ldots, r_n\}}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} r \succeq r_i} \qquad \frac{\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \qquad \overline{\mathcal{S}} = \overline{\mathcal{S}}_1, r_1 \mapsto R_1, \overline{\mathcal{S}}_2}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \bullet \succeq r_1} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \rho}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho \succeq \rho}$$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho' \qquad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho' \succeq \rho_1}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1}$$

$$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \varphi}$$

$$\frac{\begin{array}{c}\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \\ \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho \succeq \rho_i \quad ^{i \in 1 \ldots n}\end{array}}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

Fig. 20. Static semantics of SEC (casts)

It is worth comparing the treatment of latent effects in the Single Effect Calculus with their treatment in the other two type systems:

$$\frac{\Delta; \Gamma \vdash_{\mathrm{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_1'), \varphi_1 \qquad \Delta; \Gamma \vdash_{\mathrm{exp}} e_2 : \tau_1, \varphi_2}{\Delta; \Gamma \vdash_{\mathrm{exp}} e_1\ e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \{\rho_1'\} \cup \varphi'} \ \text{TRC}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{exp}} e_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2, \rho_1'), \varphi \qquad \Delta \vdash_{\mathrm{er}} \varphi \ni \rho_1' \\ \Delta; \Gamma \vdash_{\mathrm{exp}} e_2 : \tau_1, \varphi \qquad \Delta \vdash_{\mathrm{ee}} \varphi \supseteq \varphi'\end{array}}{\Delta; \Gamma \vdash_{\mathrm{exp}} e_1\ e_2 : \tau_2, \varphi} \ \text{BRC}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e_1 : (\tau_1 \xrightarrow{\theta'} \tau_2, \rho_1'), \theta \qquad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \theta \succeq \rho_1' \\ \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e_2 : \tau_1, \theta \qquad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \theta \succeq \theta'\end{array}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e_1\ e_2 : \tau_2, \theta} \ \text{SEC}$$

In the Single Effect Calculus, the composite effect $\varphi_1 \cup \varphi_2 \cup \{\rho_1'\}$ is witnessed by a single region θ that subsumes the effect of the entire expression. We interpret θ as an upper bound on the composite effect; hence, θ is an upper bound on each of the effect sets φ_1 and φ_2, which explains why θ is used in the antecedents that type-check the sub-expressions e_1 and e_2. We require ρ_1' to outlive the current region θ by the antecedent $\Delta \vdash_{\mathrm{rr}} \theta \succeq \rho_1'$. Finally, we require the latent single effect θ', which is an upper bound on the set of regions affected by executing the function, to outlive the current region, which ensures that θ is also an upper bound on the set of regions affected by executing the function.

In the Bounded Region Calculus, the effect for which θ is an upper bound is made manifest as φ. The antecedents $\Delta \vdash_{\mathrm{er}} \varphi \ni \rho_1'$ and $\Delta \vdash_{\mathrm{ee}} \varphi \supseteq \varphi'$ serve the same purpose as the $\Delta \vdash_{\mathrm{rr}} \theta \succeq \rho_1'$ and $\Delta \vdash_{\mathrm{rr}} \theta \succeq \theta'$, namely to ensure that the region of the function closure and the latent effect are subsumed by the effect of the application.

$\boxed{\vdash_{\mathrm{prog}} p \ \mathsf{ok}}$

$$\frac{\cdot, \mathcal{H} \succeq \{\}; \cdot; \cdot : \cdot \vdash_{\mathrm{exp}} p : \mathsf{bool}, \mathcal{H}}{\vdash_{\mathrm{prog}} p \ \mathsf{ok}}$$

Fig. 21. Static semantics of SEC (surface programs)

4.5.3 Technical details

Figures B 1–B 4 in Appendix B contain additional judgements for ensuring that places ρ, effects φ, boxed types μ, types τ, region contexts Δ, value contexts Γ, closed values v, storable values w, stack types $\overline{\mathbb{S}}$ and stacks S are well-formed.

4.5.4 Surface programs

Since surface programs should not admit syntax for naming regions, we adopt the judgement $\vdash_{\mathrm{prog}} p \ \mathsf{ok}$ given in Figure 21. The rule for top-level surface programs requires that an expression evaluate to a boolean value in the context of distinguished region $\mathcal{H}$ that remains live throughout the execution of the program. It also serves as the single effect that bounds the effects of the entire program. Alternative formulations of these "boundary conditions" exist; we have adopted these to simplify the translation in Section 5.

It is useful to note that the static semantics can be greatly simplified given this rule for surface programs. Pushing these empty stack types and stack domains through the rules leads to the following simplifications:

$\Delta; \Gamma; \cdot : \cdot \vdash_{\mathrm{exp}} e : \tau, \theta \implies \Delta; \Gamma \vdash_{\mathrm{exp}} e : \tau, \theta$	
$\Delta; \cdot \vdash_{\mathrm{btype}} \mu \implies \Delta \vdash_{\mathrm{btype}} \mu$	$\Delta; \cdot \vdash_{\mathrm{type}} \tau \implies \Delta \vdash_{\mathrm{type}} \tau$
$\Delta; \cdot \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1 \implies \Delta \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1$	$\Delta; \cdot \vdash_{\mathrm{re}} \rho \succeq \varphi \implies \Delta \vdash_{\mathrm{re}} \rho \succeq \varphi$
$\Delta; \cdot \vdash_{\mathrm{place}} \rho \implies \Delta \vdash_{\mathrm{place}} \rho$	$\Delta; \cdot \vdash_{\mathrm{eff}} \varphi \implies \Delta \vdash_{\mathrm{eff}} \varphi$
$\cdot \vdash_{\mathrm{rctxt}} \Delta \implies \vdash_{\mathrm{rctxt}} \Delta$	$\Delta; \cdot \vdash_{\mathrm{vctxt}} \Gamma \implies \Delta \vdash_{\mathrm{vctxt}} \Gamma$
$\vdash_{\mathrm{ctxt}} \Delta; \Gamma; \cdot : \cdot; \theta \implies \vdash_{\mathrm{ctxt}} \Delta; \Gamma; \theta$	

4.5.5 Translation of BRC to SEC

We can give a straightforward translation from the Bounded Region Calculus into the Single Effect Calculus.

At the type level, this transation expands every function type into a region abstraction and function type:

$$\hat{\mathbb{T}}\left[\!\!\left[(\tau_1 \xrightarrow{\varphi} \tau_2, \rho)\right]\!\!\right] = (\Pi\vartheta \succeq \varphi.^{\rho}(\hat{\mathbb{T}}[\![\tau_1]\!] \xrightarrow{\vartheta} \hat{\mathbb{T}}[\![\tau_2]\!], \rho), \rho)$$

At the term level, source functions become region abstractions and functions, and applications become region instantiations and applications. A similar approach deals with region abstractions in the source language. Essentially, this translation

Boxed types

$$\begin{aligned}
\hat{\mathbb{T}}\left[\!\!\left[\tau_1 \xrightarrow{\varphi'} \tau_2\right]\!\!\right]_\rho &= \Pi\vartheta \succeq \varphi'.^\rho(\hat{\mathbb{T}}[\![\tau_1]\!] \xrightarrow{\vartheta} \hat{\mathbb{T}}[\![\tau_2]\!], \rho) \\
\hat{\mathbb{T}}\left[\!\!\left[\Pi\varrho \succeq \varphi.^{\varphi'}\tau\right]\!\!\right]_\rho &= \Pi\varrho \succeq \varphi.^\rho(\Pi\vartheta \succeq \varphi'.^\vartheta\hat{\mathbb{T}}[\![\tau]\!], \rho)
\end{aligned}$$

Types

$$\hat{\mathbb{T}}[\![(\mu,\rho)]\!] = (\hat{\mathbb{T}}[\![\mu]\!]_\rho, \rho)$$

Expressions

$$\begin{aligned}
\hat{\mathbb{E}}\left[\!\!\left[\lambda x:\tau.^{\varphi'}e \text{ at } \rho\right]\!\!\right]_\theta &= \lambda\vartheta \succeq \varphi'.^\rho(\lambda x:\hat{\mathbb{T}}[\![\tau]\!].^\vartheta\hat{\mathbb{E}}[\![e]\!]_\vartheta \text{ at } \rho) \text{ at } \rho \\
\hat{\mathbb{E}}[\![e_1\ e_2]\!]_\theta &= \hat{\mathbb{E}}[\![e_1]\!]_\theta\ [\theta]\ \hat{\mathbb{E}}[\![e_2]\!]_\theta \\
\hat{\mathbb{E}}[\![\text{letregion } \varrho \text{ in } e]\!]_\theta &= \text{letregion } \varrho \text{ in } \hat{\mathbb{E}}[\![e]\!]_\varrho \\
\hat{\mathbb{E}}\left[\!\!\left[\lambda\varrho \succeq \varphi.^{\varphi'}u \text{ at } \rho\right]\!\!\right]_\theta &= \lambda\varrho \succeq \varphi.^\rho(\lambda\vartheta \succeq \varphi'.^\vartheta\hat{\mathbb{E}}[\![u]\!]_\vartheta \text{ at } \rho) \text{ at } \rho \\
\hat{\mathbb{E}}[\![e\ [\rho]]\!]_\theta &= \hat{\mathbb{E}}[\![e]\!]_\theta\ [\rho]\ [\theta]
\end{aligned}$$

Programs

$$\hat{\mathbb{E}}[\![e]\!] = \hat{\mathbb{E}}[\![e]\!]_{\mathcal{H}}$$

Fig. 22. Translation from BRC to SEC (abbreviated)

works by looking for the places where region sets are used in BRC and simply replacing them by an abstraction bounded by that set. Clearly, this is not the most efficient translation. For example, in places where we could statically identify an upper bound on the region set (e.g., a singleton region set), we could elide the abstraction and simply use the upper bound.

Figure 22 gives an abbreviated translation from the Bounded Effect Calculus to the Single Effect Calculus (the translation is homomorphic on the other syntactic forms). The translation witnesses each introduced bounded abstraction with the current region, which is threaded through the translation by the θ component of $\hat{\mathbb{E}}[\![e]\!]_\theta$. We can prove that the translation is type- and meaning-preserving. (Note that we adopt the simplified type-system for the Single Effect Calculus (see Section 4.5.4), where we assume empty stack types and domains.)

Lemma 4.2 (*Translation Preserves Types*)

(1) If $\Delta;\Gamma \vdash^{\text{BRC}}_{\text{exp}} e : \tau, \varphi$, then forall Δ' and θ,
if $\vdash^{\text{SEC}}_{\text{ctxt}} \hat{\mathbb{D}}[\![\Delta]\!], \Delta'; \hat{\mathbb{G}}[\![\Gamma]\!]; \theta$ and $\hat{\mathbb{D}}[\![\Delta]\!], \Delta' \vdash^{\text{SEC}}_{\text{re}} \theta \succeq \varphi$,
then $\hat{\mathbb{D}}[\![\Delta]\!], \Delta'; \hat{\mathbb{G}}[\![\Gamma]\!] \vdash^{\text{SEC}}_{\text{exp}} \hat{\mathbb{E}}[\![e]\!]_\theta : \hat{\mathbb{T}}[\![\tau]\!], \theta$.

(2) If $\vdash^{\text{BRC}}_{\text{prog}} p$ ok, then $\vdash^{\text{SEC}}_{\text{prog}} \hat{\mathbb{E}}[\![p]\!]$ ok.

5 The Translation

In this section we present a type- and semantics-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. Many of the key components of the translation should be obvious from the suggestive naming of the previous sections. We clearly intend letregion to be translated (in some fashion) to letRGN. Likewise, we can expect types of the form (μ, ρ) to be translated to types of the form RGNLoc τ_r τ_a.

It further seems likely that the outlives relation $\rho_2 \succeq \rho_1$ should be related to the witness functions $\tau_{r_1} \preceq \tau_{r_2}$. We present the translation in stages, as there are some subtleties that require explanation.

We start with a few preliminaries. We assume injections from the sets $RVars^{SEC}$ and $Vars^{SEC}$ to the sets $TVars^{FRGN}$ and $Vars^{FRGN}$ respectively. In the translation, applications of such injections will be clear from context and we freely use variables from source objects in target objects. We further assume two additional injections from the set $RVars^{SEC}$ to the set $Vars^{FRGN}$; the first, written h_ϱ will denote the handle for the region ϱ, while the second, written w_ϱ will denote the witnesses which coerce the region ϱ to its bounding regions. Finally, as a SEC program requires exactly one region stack for evaluation, we assume that the corresponding stack in the translated $\mathsf{F}^{\mathsf{RGN}}$ program is labelled by the stack name s.

The translation is a typed call-by-value monad translation, similar to the standard translation given by Sabry and Wadler (1997). We have not attempted to optimize the translation to avoid the introduction of "administrative" redexes. We feel that this simplifies the translation, and it does not significantly complicate the proof that the translation preserves the semantics, owing to the fact that only three expression forms in the source calculus are value forms. The translation is given by a number of functions: $\mathbb{T}[\![\cdot]\!]$ translates into types, $\mathbb{D}[\![\cdot]\!]$ translates into type contexts, $\mathbb{G}[\![\cdot]\!]$ translates into value contexts, $\mathbb{E}[\![\cdot]\!]$ translates into expressions, and $\mathbb{X}[\![\cdot]\!]$ translates into towers, tower types, and tower domains. Technically, there are separate functions for each syntactic class in the source calculus, but we elide this detail as it is always clear from context. Additionally, to reduce notational clutter, translations from judgements are often written in an abbreviated form giving only the main component; the corresponding judgement should be clear from context.

Figure 23 shows the translation of types and Figure 24 gives the extension of the translation to contexts. As expected, the type (μ, ρ) is translated to $\mathsf{RGNLoc}\ \mathbb{T}[\![\rho]\!]\ \mathbb{T}[\![\mu]\!]$, whereby region allocated values in the source are also region allocated in the target. The translations of primitive types and product types are trivial. More interesting are the translations of function types and region abstraction types. Functions with effects bounded by the region θ are translated into pure functions that yield computations in the RGN monad indexed by θ, whereas region abstractions are translated into type abstractions. Because the target calculus requires explicit region handles for allocation, each time a region is in scope in the source calculus, the region handle must be in scope in the target calculus. This explains the appearance of the $\mathsf{RGNHandle}\ \varrho$ type in the translation. Likewise, the target calculus makes witness functions explicit, whereas in the source calculus such coercions are implied by $\succeq$ related regions. Hence, we interpret $\varrho \succeq \{\rho_1, \ldots, \rho_n\}$ as an n-tuple of functions, each witnessing a coercion from region ρ_i to ϱ. This interpretation is formalized by the $\mathbb{T}[\![\varrho \succeq \{\rho_1, \ldots \rho_n\}]\!]$ translation.[4]

We extend the type translation to contexts in the obvious way. In addition to translating region variables to type variables and translating the types of variables

[4] Note that we treat $\{\rho_1, \ldots, \rho_n\}$ as a list with fixed order and not as a set, so we can realize the witness with an ordered tuple.

Translations yielding types

Places

$$\mathbb{T}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\mathrm{rctxt}} \Delta \qquad \varrho \in \mathit{dom}(\Delta)}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{place}} \varrho} \right\rrbracket = \varrho$$

$$\mathbb{T}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\mathrm{rctxt}} \Delta \qquad r \in \mathit{dom}(\overline{\mathbb{S}})}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{place}} r} \right\rrbracket = s\sharp r$$

$$\mathbb{T}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\mathrm{rctxt}} \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{place}} \bullet} \right\rrbracket = \begin{cases} \circ\sharp\bullet & \text{if } \overline{\mathbb{S}} = \cdot \\ s\sharp\bullet & \text{otherwise} \end{cases}$$

Types

$$\mathbb{T}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\mathrm{rctxt}} \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{type}} \mathsf{bool}} \right\rrbracket = \mathsf{bool}$$

$$\mathbb{T}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{btype}} \mu \qquad \Delta; \overline{\mathbb{S}} \vdash_{\mathrm{place}} \rho}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{type}} (\mu, \rho)} \right\rrbracket = \mathsf{RGNLoc}\ \mathbb{T}\llbracket \rho \rrbracket\ \mathbb{T}\llbracket \mu \rrbracket$$

Boxed types

$$\mathbb{T}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\mathrm{rctxt}} \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{btype}} \mathsf{int}} \right\rrbracket = \mathsf{int}$$

$$\mathbb{T}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{type}} \tau_1 \qquad \Delta; \overline{\mathbb{S}} \vdash_{\mathrm{place}} \theta \qquad \Delta; \overline{\mathbb{S}} \vdash_{\mathrm{type}} \tau_2}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{btype}} \tau_1 \xrightarrow{\theta} \tau_2} \right\rrbracket = \mathbb{T}\llbracket \tau_1 \rrbracket \to \mathsf{RGN}\ \mathbb{T}\llbracket \theta \rrbracket\ \mathbb{T}\llbracket \tau_2 \rrbracket$$

$$\mathbb{T}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{type}} \tau_1 \qquad \Delta; \overline{\mathbb{S}} \vdash_{\mathrm{type}} \tau_2}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{btype}} \tau_1 \times \tau_2} \right\rrbracket = \mathbb{T}\llbracket \tau_1 \rrbracket \times \mathbb{T}\llbracket \tau_2 \rrbracket$$

$$\mathbb{T}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{eff}} \varphi \qquad \Delta, \varrho \succeq \varphi; \overline{\mathbb{S}} \vdash_{\mathrm{place}} \theta \qquad \Delta, \varrho \succeq \varphi; \overline{\mathbb{S}} \vdash_{\mathrm{type}} \tau}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{btype}} \Pi \varrho \succeq \varphi.^{\theta} \tau} \right\rrbracket = \\ \forall \varrho. \mathbb{T}\llbracket \varrho \succeq \varphi \rrbracket \to \mathsf{RGNHandle}\ \varrho \to \mathsf{RGN}\ \mathbb{T}\llbracket \theta \rrbracket\ \mathbb{T}\llbracket \tau \rrbracket$$

Witnesses

$$\mathbb{T}\llbracket \Delta; \overline{\mathbb{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1 \rrbracket = \mathbb{T}\llbracket \rho_1 \rrbracket \preceq \mathbb{T}\llbracket \rho_2 \rrbracket = \forall \beta. \mathsf{RGN}\ \mathbb{T}\llbracket \rho_1 \rrbracket\ \beta \to \mathsf{RGN}\ \mathbb{T}\llbracket \rho_2 \rrbracket\ \beta$$

$$\mathbb{T}\left\llbracket \frac{\begin{array}{c} \overline{\mathbb{S}} \vdash_{\mathrm{rctxt}} \Delta \\ \overline{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{rr}} \rho \succeq \rho_i}^{\,i \in 1 \ldots n} \end{array}}{\Delta; \overline{\mathbb{S}} \vdash_{\mathrm{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}} \right\rrbracket = (\mathbb{T}\llbracket \rho \succeq \rho_1 \rrbracket \times \cdots \times \mathbb{T}\llbracket \rho \succeq \rho_n \rrbracket)$$

Fig. 23. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Types)

in value contexts, we have additional translations from region contexts to value contexts. As explained above, region handles and witness functions are explicit values in the target calculus. Hence, our translation maintains the invariant that whenever a region variable $\varrho \succeq \{\rho_1, \ldots, \rho_n\}$ is in scope in the source calculus, the variables h_ϱ and w_ϱ are in scope in the target calculus. The variable h_ϱ (of type $\mathsf{RGNHandle}\ \varrho$) is the handle for the region ϱ and the variable w_ϱ (of type $\mathbb{T}\llbracket \varrho \succeq \varphi \rrbracket$) is the tuple holding the witness functions that coerce to region ϱ.

Figure 25 shows the translation of witness terms. The first six translations map the reflexive, transitive closure of the syntactic constraints in the source Δ and $\overline{\mathbb{S}}$ into an appropriate coercion function. The final translation collects a set of coercion functions into a tuple; such a term is suitable as an argument to the translation of a

Translations yielding type contexts

Region contexts

$$\mathbb{D}\left\llbracket \frac{}{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \cdot} \right\rrbracket \;=\; \cdot$$

$$\mathbb{D}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \notin \mathit{dom}(\Delta) \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{eff}} \varphi}{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta, \varrho \succeq \varphi} \right\rrbracket \;=\; \mathbb{D}\llbracket \Delta \rrbracket, \varrho$$

Translations yielding value contexts

Region contexts

$$\mathbb{G}\left\llbracket \frac{}{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \cdot} \right\rrbracket \;=\; \cdot$$

$$\mathbb{G}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \notin \mathit{dom}(\Delta) \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{eff}} \varphi}{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta, \varrho \succeq \varphi} \right\rrbracket \;=\; \mathbb{G}\llbracket \Delta \rrbracket, h_\varrho : \mathsf{RGNHandle}\ \varrho, w_\varrho : \mathbb{T}\llbracket \varrho \succeq \varphi \rrbracket$$

Value contexts

$$\mathbb{G}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\text{vctxt}} \cdot} \right\rrbracket \;=\; \cdot$$

$$\mathbb{G}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{vctxt}} \Gamma \qquad x \notin \mathit{dom}(\Gamma) \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathbb{S}} \vdash_{\text{vctxt}} \Gamma, x : \tau} \right\rrbracket \;=\; \mathbb{G}\llbracket \Gamma \rrbracket, x{:}\mathbb{T}\llbracket \tau \rrbracket$$

Fig. 24. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Contexts)

Translations yielding expressions

Witnesses

$$\mathbb{E}\left\llbracket \frac{\begin{array}{c}\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \\ (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta\end{array}}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \varrho \succeq \rho_i} \right\rrbracket \;=\; \Lambda\beta.\lambda k : \mathsf{RGN}\ \mathbb{T}\llbracket \rho_i \rrbracket\ \beta.\mathsf{let}\ w = \mathsf{sel}_i\ w_\varrho\ \mathsf{in}\ w\ [\beta]\ k$$

$$\mathbb{E}\left\llbracket \frac{\begin{array}{c}\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \\ \overline{\mathbb{S}} = \overline{\mathbb{S}}_1, r_1 \mapsto \overline{\mathbb{R}}_1, \overline{\mathbb{S}}_2, r_2 \mapsto \overline{\mathbb{R}}_2, \overline{\mathbb{S}}_3\end{array}}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} r_2 \succeq r_1} \right\rrbracket \;=\; \Lambda\beta.\lambda k : \mathsf{RGN}\ s\sharp r_1\ \beta.\mathsf{witnessRGN}\ s\sharp r_1\ s\sharp r_2\ [\beta]\ k$$

$$\mathbb{E}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{re}} r \succeq \{r_1, \ldots, r_n\}}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} r \succeq r_i} \right\rrbracket \;=\; \Lambda\beta.\lambda k : \mathsf{RGN}\ \mathbb{T}\llbracket r_i \rrbracket\ \beta.\mathsf{let}\ w = \mathsf{sel}_i\ \mathbb{E}\llbracket r \succeq \{r_1, \ldots, r_n\} \rrbracket\ \mathsf{in}\ w\ [\beta]\ k$$

$$\mathbb{E}\left\llbracket \frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \overline{\mathbb{S}} = \overline{\mathbb{S}}_1, r_1 \mapsto R_1, \overline{\mathbb{S}}_2}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \bullet \succeq r_1} \right\rrbracket \;=\; \Lambda\beta.\lambda k : \mathsf{RGN}\ s\sharp r_1\ \beta.\mathsf{witnessRGN}\ s\sharp r_1\ s\sharp\bullet\ [\beta]\ k$$

$$\mathbb{E}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{place}} \rho}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho \succeq \rho} \right\rrbracket \;=\; \Lambda\beta.\lambda k : \mathsf{RGN}\ \mathbb{T}\llbracket \rho \rrbracket\ \beta.k$$

$$\mathbb{E}\left\llbracket \frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho_1} \right\rrbracket \;=\; \Lambda\beta.\lambda k : \mathsf{RGN}\ \mathbb{T}\llbracket \rho_1 \rrbracket\ \beta.\mathsf{let}\ k' = \mathbb{E}\llbracket \rho' \succeq \rho_1 \rrbracket\ [\beta]\ k\ \mathsf{in}\ \mathbb{E}\llbracket \rho_2 \succeq \rho' \rrbracket\ [\beta]\ k'$$

$$\mathbb{E}\left\llbracket \frac{\begin{array}{c}\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \\ \Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho \succeq \rho_i \quad {}^{i \in 1 \ldots n}\end{array}}{\Delta; \overline{\mathbb{S}} \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}} \right\rrbracket \;=\; (\mathbb{E}\llbracket \rho \succeq \rho_1 \rrbracket, \ldots, \mathbb{E}\llbracket \rho \succeq \rho_n \rrbracket)$$

Fig. 25. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Witnesses)

Translations yielding expressions
Places

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \in dom(\Delta)}{\Delta; \overline{\mathbb{S}} \vdash_{\text{place}} \varrho}\right]\!\!\right] = h_\varrho$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad r \in dom(\overline{\mathbb{S}})}{\Delta; \overline{\mathbb{S}} \vdash_{\text{place}} r}\right]\!\!\right] = \mathsf{handle}(s\sharp r)$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\text{place}} \bullet}\right]\!\!\right] = \begin{cases} \mathsf{handle}(\circ\sharp\bullet) & \text{if } \overline{\mathbb{S}} = \cdot \\ \mathsf{handle}(s\sharp\bullet) & \text{otherwise} \end{cases}$$

Fig. 26. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Places)

region abstraction. Figure 26 translates a single region place into its corresponding region handle.

Figures 27 and 28 and Figures C 1 and C 2 in Appendix C give the translation of terms. In order to make the translation easier to read, we introduce the following notation, reminiscent of Haskell's `do` notation:

$$\begin{aligned} \mathsf{bind}\ f{:}\tau_a \Leftarrow e_1; e_2 \equiv\ & \mathsf{let}\ k = e_1\ \mathsf{in} \\ & \mathsf{thenRGN}\ [\tau_r]\ [\tau_a]\ [\tau_b]\ k\ (\lambda f{:}\tau_a.e_2) \\ & \text{where } k \text{ fresh} \end{aligned}$$

where τ_r and τ_b are inferred from context. Note that this induces the following derived rules:

$$\frac{T; e_1 \hookrightarrow v}{T; \mathsf{bind}\ f{:}\tau_a \Leftarrow e_1; e_2 \hookrightarrow \mathsf{thenRGN}\ [\tau_r]\ [\tau_a]\ [\tau_b]\ v\ (\lambda f{:}\tau_a.e_2)}$$

$$\frac{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} e_1 : \mathsf{RGN}\ \tau_r\ \tau_a \qquad \Delta; \Gamma, f{:}\tau_a; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} e_2 : \mathsf{RGN}\ \tau_r\ \tau_b}{\Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}} \vdash_{\text{exp}} \mathsf{bind}\ f{:}\tau_a \Leftarrow e_1; e_2 : \mathsf{RGN}\ \tau_r\ \tau_b}$$

The translation of an integer constant is a canonical example of allocation in the target calculus. The allocation is accomplished by the newRGNLoc command, applied to the appropriate region handle and value. However, the resulting command has type $\mathsf{RGN}\ \mathbb{T}[\![\rho]\!]\ (\mathsf{RGNLoc}\ \mathbb{T}[\![\rho]\!]\ \mathsf{int})$, whereas the source typing judgement requires the computation to be expressed relative to the region θ. We coerce the computation using a witness function, whose existence is implied by the judgement $\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \theta \succeq \rho$. Allocation of a function proceeds in exactly the same manner. Function application, while notationally heavy, is simple. The thenRGN commands (implicit in the bind expressions) sequence evaluating the function to a location, reading the location, evaluating the argument, and applying the function to the argument.

The translation of letregion ϱ in e is pleasantly direct. As described above, we introduce ϱ, h_ϱ, and w_ϱ through Λ- and λ-abstractions. The region handle and coercion function are supplied by the letRGN command when the computation is executed.

The translation of region abstraction is similar to the translation of functions. Once again, region handles and witness functions are λ-bound in accordance to the invariants described above. During the translation of region applications, the

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\dfrac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta \quad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho \qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} i \ \mathsf{at}\ \rho : (\mathsf{int},\rho),\theta}\right]\!\!\right] = \mathbb{E}[\![\theta\succeq\rho]\!]\ [\mathbb{T}[\![(\mathsf{int},\rho)]\!]]\ (\mathsf{newRGNLoc}\ [\mathbb{T}[\![\rho]\!]]\ [\mathbb{T}[\![\mathsf{int}]\!]]\ \mathbb{E}[\![\rho]\!]\ \ i)$$

$$\mathbb{E}\left[\!\!\left[\dfrac{\vdash_{\mathrm{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta \quad x\in \mathit{dom}(\Gamma) \qquad \Gamma(x)=\tau}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} x:\tau,\theta}\right]\!\!\right] = \mathsf{returnRGN}\ [\mathbb{T}[\![\theta]\!]]\ [\mathbb{T}[\![\tau]\!]]\ x$$

$$\mathbb{E}\left[\!\!\left[\dfrac{\Delta;\Gamma,x:\tau_1;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e':\tau_2,\theta' \quad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{place}}\rho \qquad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} \lambda x:\tau_1.^{\theta'} e' \ \mathsf{at}\ \rho : (\tau_1 \xrightarrow{\theta'} \tau_2,\rho),\theta}\right]\!\!\right] =$$
$$\mathbb{E}[\![\theta\succeq\rho']\!]\ [\mathbb{T}\left[\!\!\left[(\tau_1 \xrightarrow{\theta'} \tau_2,\rho')\right]\!\!\right]]$$
$$(\mathsf{newRGNLoc}\ [\mathbb{T}[\![\rho']\!]]\ [\mathbb{T}\left[\!\!\left[\tau_1 \xrightarrow{\theta'} \tau_2\right]\!\!\right]]\ \mathbb{E}[\![\rho']\!]\ \ (\lambda x{:}\mathbb{T}[\![\tau_1]\!].\mathbb{E}[\![e]\!]))$$

$$\mathbb{E}\left[\!\!\left[\dfrac{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1:(\tau_1 \xrightarrow{\theta'} \tau_2,\rho_1'),\theta \quad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\rho_1' \quad \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_2:\tau_1,\theta \quad \Delta;\overline{\mathcal{S}}\vdash_{\mathrm{rr}}\theta\succeq\theta'}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\mathrm{exp}} e_1\ e_2:\tau_2,\theta}\right]\!\!\right] =$$

$$\begin{array}{l}
\mathsf{bind}\ f{:}\mathbb{T}\left[\!\!\left[(\tau_1 \xrightarrow{\theta'} \tau_2,\rho_1')\right]\!\!\right] \Leftarrow \mathbb{E}[\![e_1]\!]\,;\\
\mathsf{bind}\ g{:}\mathbb{T}\left[\!\!\left[\tau_1 \xrightarrow{\theta'} \tau_2\right]\!\!\right]\\
\quad \Leftarrow \mathbb{E}[\![\theta\succeq\rho_1']\!]\ [\mathbb{T}\left[\!\!\left[\tau_1 \xrightarrow{\theta'} \tau_2\right]\!\!\right]]\ (\mathsf{readRGNLoc}\ [\mathbb{T}[\![\rho_1']\!]]\ [\mathbb{T}\left[\!\!\left[\tau_1 \xrightarrow{\theta'} \tau_2\right]\!\!\right]]\ f);\\
\mathsf{bind}\ a{:}\mathbb{T}[\![\tau_1]\!] \Leftarrow \mathbb{E}[\![e_2]\!]\,;\\
\mathsf{let}\ z = g\ a\ \mathsf{in}\\
\mathbb{E}[\![\theta\succeq\theta']\!]\ [\mathbb{T}[\![\tau_2]\!]]\ z
\end{array}$$

where f, g, a, z fresh

Fig. 27. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Terms (I))

appropriate tuple of witness functions (constructed by $\mathbb{E}[\![\Delta;\overline{\mathcal{S}}\vdash_{\mathrm{re}}\rho_2\succeq\varphi]\!]$) and region handle are supplied as arguments.

Figures C 3 and C 4 in Appendix C give the translations of closed and storable values, which follow directly from the translations of expressions. Figure 29 gives the translation of stacks, where each stored value is translated according to the $\vdash_{\mathrm{sto}}$ derivation implied by the $\vdash_{\mathrm{stack}}$ derivation. There is one minor complication due to the fact that a Single Effect Calculus program has an implicit region stack, while $\mathsf{F}^{\mathsf{RGN}}$ explicitly introduces (and eliminates) a region stack with the runRGN command. Hence, a stack domain, stack type, or stack may be translated to either an empty tower or a tower with a single stack. We make this choice based on whether or not *any* region is in the stack. A similar issue arises with occurences of $\bullet$ in the source program, which may be translated either to $s\sharp\bullet$, within the scope of the runRGN, where s is the name of the stack introduced by the runRGN, or to $\circ\sharp\bullet$, outside the scope of the runRGN. All of the translations given in this section must be given via translations on derivations, essentially to propagate the $\overline{\mathcal{S}}$ stack domain

Translations yielding expressions

Expressions

$$\mathbb{E}\left\llbracket\frac{\begin{array}{c}\Delta;\overline{\mathcal{S}} \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta \\ \Delta,\varrho \succeq \{\theta\};\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} e:\tau,\varrho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} \mathsf{letregion}\ \varrho\ \mathsf{in}\ e:\tau,\theta}\right\rrbracket =$$

$$\mathsf{letRGN}\ [\mathbb{T}\llbracket\theta\rrbracket]\ [\mathbb{T}\llbracket\tau\rrbracket]\ (\Lambda\varrho.\lambda w_\varrho{:}\mathbb{T}\llbracket\varrho \succeq \{\theta\}\rrbracket.\lambda h_\varrho{:}\mathsf{RGNHandle}\ \varrho.\mathbb{E}\llbracket e\rrbracket)$$

$$\mathbb{E}\left\llbracket\frac{\begin{array}{c}\Delta,\varrho \succeq \varphi;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} u':\tau,\theta' \\ \Delta;\overline{\mathcal{S}} \vdash_{\text{place}} \rho \qquad \Delta;\overline{\mathcal{S}} \vdash_{\text{rr}} \theta \succeq \rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} \lambda\varrho \succeq \varphi.^{\theta'} u'\ \mathsf{at}\ \rho : (\Pi\varrho \succeq \varphi.^{\theta'}\tau,\rho),\theta}\right\rrbracket =$$

$$\begin{array}{l}\mathbb{E}\llbracket\theta \succeq \rho\rrbracket\ [\mathbb{T}\llbracket(\Pi\varrho \succeq \varphi.^{\theta'}\tau,\rho)\rrbracket] \\ \qquad (\mathsf{newRGNLoc}\ [\mathbb{T}\llbracket\rho\rrbracket]\ [\mathbb{T}\llbracket\Pi\varrho \succeq \varphi.^{\theta'}\tau\rrbracket] \\ \qquad\qquad \mathbb{E}\llbracket\rho\rrbracket\ (\Lambda\varrho.\lambda w_\varrho{:}\mathbb{T}\llbracket\varrho \succeq \theta\rrbracket.\lambda h_\varrho : \mathsf{RGNHandle}\ \varrho.\mathbb{E}\llbracket u\rrbracket))\end{array}$$

$$\mathbb{E}\left\llbracket\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} e_1 : (\Pi\varrho \succeq \varphi.^{\theta'}\tau,\rho_1'),\theta \qquad \Delta;\overline{\mathcal{S}} \vdash_{\text{rr}} \theta \succeq \rho_1' \\ \Delta;\overline{\mathcal{S}} \vdash_{\text{place}} \rho_2 \qquad \Delta;\overline{\mathcal{S}} \vdash_{\text{re}} \rho_2 \succeq \varphi \qquad \Delta;\overline{\mathcal{S}} \vdash_{\text{rr}} \theta \succeq \theta'[\rho_2/\varrho]\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} e_1\ [\rho_2] : \tau[\rho_2/\varrho],\theta}\right\rrbracket =$$

$$\begin{array}{l}\mathsf{bind}\ f{:}\mathbb{T}\llbracket(\Pi\varrho \succeq \varphi.^{\theta'}\tau,\rho_1')\rrbracket \Leftarrow \mathbb{E}\llbracket e\rrbracket\ ; \\ \mathsf{bind}\ g{:}\mathbb{T}\llbracket\Pi\varrho \succeq \varphi.^{\theta'}\tau\rrbracket \\ \qquad \Leftarrow \mathbb{E}\llbracket\theta \succeq \rho_1'\rrbracket\ [\mathbb{T}\llbracket\Pi\varrho \succeq \varphi.^{\theta'}\tau\rrbracket]\ (\mathsf{readRGNLoc}\ [\mathbb{T}\llbracket\rho_1'\rrbracket]\ [\mathbb{T}\llbracket\Pi\varrho \succeq \varphi.^{\theta'}\tau\rrbracket]\ f); \\ \mathsf{let}\ z = (g\ [\mathbb{T}\llbracket\rho_2\rrbracket]\ \mathbb{E}\llbracket\rho_2 \succeq \varphi\rrbracket\ \mathbb{E}\llbracket\rho_2\rrbracket)\ \mathsf{in} \\ \mathbb{E}\llbracket\theta \succeq \theta'[\rho_2/\varrho]\rrbracket\ [\mathbb{T}\llbracket\tau[\rho_2/\varrho]\rrbracket]\ z\end{array}$$

where f, g, z fresh

$$\mathbb{E}\left\llbracket\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta \qquad r \in \mathit{dom}(\overline{\mathcal{S}}) \qquad l \in \mathit{dom}(\overline{\mathcal{S}}(r)) \qquad \mu = \mathcal{S}(r,l)}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} \langle l\rangle_r : (\mu,r),\theta}\right\rrbracket =$$

$$\mathsf{returnRGN}\ [\mathbb{T}\llbracket\theta\rrbracket]\ [\mathbb{T}\llbracket(\mu,r)\rrbracket]\ \langle l\rangle_{s\sharp r}$$

$$\mathbb{E}\left\llbracket\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\theta \qquad \Delta;\overline{\mathcal{S}} \vdash_{\text{btype}} \mu}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}} \vdash_{\text{exp}} \langle l\rangle_\bullet : (\mu,\bullet),\theta}\right\rrbracket =$$

$$\begin{cases}\mathsf{returnRGN}\ [\mathbb{T}\llbracket\theta\rrbracket]\ [\mathbb{T}\llbracket(\mu,\bullet)\rrbracket]\ \langle l\rangle_{\circ\sharp\bullet} & \text{if } \overline{\mathcal{S}} = \cdot \\ \mathsf{returnRGN}\ [\mathbb{T}\llbracket\theta\rrbracket]\ [\mathbb{T}\llbracket(\mu,\bullet)\rrbracket]\ \langle l\rangle_{s\sharp\bullet} & \text{otherwise}\end{cases}$$

Fig. 28. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Terms (II))

to each point where a $\bullet$ may appear. Again, we make the choice of translation based on whether or not *any* region is in the $\overline{\mathcal{S}}$ stack domain (see Figures 23 and 26).

5.1 Surface Programs

Figure 30 shows the translation of programs. An entire region computation is encapsulated and run by the runRGN expression. We bind $w_{\mathcal{H}}$ to an empty tuple, which corresponds to the absence of any coercion functions to the region $\mathcal{H}$.

When we are only interested in translating surface programs, then neither r nor $\bullet$ can appear in types or expressions. This simplifies many of the translations. We no longer require any translations of stack domains, stack types, or stacks. Furthermore, the translations of region contexts, places (which must be of the form ϱ), types and boxed types, and value contexts can all be given as syntactic translations (rather than translations on derivations). The more complicated translation

Translations yielding tower domains

Stack domains

$$\mathbb{X}[\![\overline{\mathbb{S}}]\!] = \begin{cases} \cdot & \text{if } \overline{\mathbb{S}} = \cdot \\ \cdot, s \mapsto \overline{\mathbb{S}} & \text{otherwise} \end{cases}$$

Translations yielding stack types

Stacks types

$$\mathbb{X}\left[\!\!\left[\frac{\begin{array}{c} dom(\overline{\mathbb{S}}) = dom(\mathbb{S}) \\ \forall r \in dom(\overline{\mathbb{S}}).\ dom(\overline{\mathbb{S}}(r)) = dom(\mathbb{S}(r)) \\ \forall r \in dom(\overline{\mathbb{S}}).\forall l \in dom(\overline{\mathbb{S}}(r)).\ \cdot;\overline{\mathbb{S}} \vdash_{\text{btype}} \mathbb{S}(r,l) \end{array}}{\vdash_{\text{stype}} \mathbb{S} : \overline{\mathbb{S}}}\right]\!\!\right] = \mathbb{S}^*$$

$$\text{where} \quad \begin{array}{r} dom(S) = dom(S^*) \\ \forall r \in dom(S).dom(S(r)) = dom(S^*(r)) \\ \forall r \in dom(S).\forall l \in dom(\overline{\mathbb{S}}(r)).S^*(r,l) = \mathbb{T}[\![\mathbb{S}(r,l)]\!] \end{array}$$

Translations yielding tower types

Stack types

$$\mathbb{X}[\![\vdash_{\text{stype}} \mathbb{S} : \overline{\mathbb{S}}]\!] = \begin{cases} \cdot & \text{if } \mathbb{S} = \cdot \\ \cdot, s \mapsto \mathbb{X}[\![\vdash_{\text{stype}} \mathbb{S} : \overline{\mathbb{S}}]\!] & \text{otherwise} \end{cases}$$

Translations yielding stacks

Stacks

$$\mathbb{X}[\![\vdash_{\text{stack}} S : \mathbb{S} : \overline{\mathbb{S}}]\!] = S^*$$

$$\text{where} \quad \begin{array}{r} dom(S) = dom(S^*) \\ \forall r \in dom(S).dom(S(r)) = dom(S^*(r)) \\ \forall r \in dom(S).\forall l \in dom(\overline{\mathbb{S}}(r)).S^*(r,l) = \mathbb{E}[\![S(r,l)]\!] \end{array}$$

Translations yielding towers

Stacks

$$\mathbb{X}[\![\vdash_{\text{stack}} S : \mathbb{S} : \overline{\mathbb{S}}]\!] = \begin{cases} \cdot & \text{if } S = \cdot \\ \cdot, s \mapsto \mathbb{X}[\![\vdash_{\text{stack}} S : \mathbb{S} : \overline{\mathbb{S}}]\!] & \text{otherwise} \end{cases}$$

Fig. 29. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Stacks)

Translations yielding terms

Programs

$$\mathbb{E}\left[\!\!\left[\frac{\cdot, \mathcal{H} \succeq \{\}; \cdot; \cdot : \cdot \vdash_{\text{exp}} p : \mathsf{bool}, \mathcal{H}}{\vdash_{\text{prog}} p\ \mathsf{ok}}\right]\!\!\right] =$$

$$\begin{array}{l} \mathsf{runRGN}\ [\mathbb{T}[\![\cdot, \mathcal{H} \succeq \{\}; \cdot; \cdot : \cdot \vdash_{\text{type}} \mathsf{bool}]\!]]\ (\Lambda\mathcal{H}.\lambda h_{\mathcal{H}}{:}\mathsf{RGNHandle}\ \mathcal{H}. \\ \qquad \mathsf{let}\ w_{\mathcal{H}} = ()\ \mathsf{in} \\ \qquad \mathbb{E}[\![\cdot, \mathcal{H} \succeq \{\}; \cdot; \cdot : \cdot \vdash_{\text{exp}} p : \mathsf{bool}, \mathcal{H}]\!]) \end{array}$$

Fig. 30. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Programs)

on derivations given in this section is necessary only to establish the correctness of the translation.

5.2 Translation Properties

The translation is type preserving, in the sense formalized by the following lemma. The proof is by (mutual) induction on the structure of the typing judgements, making frequent appeals to various well-formedness lemmas.

Lemma 5.1 (*Translation Preserves Types*)

(1) If $\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta$, then $\vdash_{\mathrm{tctxt}} \mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket$.

(2) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \rho$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \rho \rrbracket$.

(3) If $\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{vctxt}} \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket$.

(4) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{btype}} \mu$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{btype}} \mu \rrbracket$.

(5) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{type}} \tau$,
then $\mathbb{T}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{T}\llbracket \vdash_{\mathrm{sdom}} \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{type}} \tau \rrbracket$.

(6) If $\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}$, then $\vdash_{\mathrm{ttype}} \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$.

(7) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{vctxt}} \mathbb{G}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma \rrbracket$.

(8) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{vctxt}} \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket , \mathbb{G}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma \rrbracket$.

(9) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1 \rrbracket$.

(10) If $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \varphi$, then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \varphi \rrbracket$.

(11) If $\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}$ and $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1$,
then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{exp}} \mathbb{E}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1 \rrbracket :$
$\mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1 \rrbracket$.

(12) If $\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}$ and $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \varphi$,
then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{type}} \mathbb{E}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \varphi \rrbracket :$
$\mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{re}} \rho \succeq \varphi \rrbracket$.

(13) If $\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}$ and $\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \rho$,
then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{exp}} \mathbb{E}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \rho \rrbracket :$
$\mathsf{handle}(\mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \rho \rrbracket)$.

(14) If $\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e : \tau, \theta$,
then $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \rrbracket , \mathbb{G}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma \rrbracket ; \mathbb{X}\llbracket \vdash_{\mathrm{sdom}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{exp}}$
$\mathbb{E}\llbracket \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e : \tau, \theta \rrbracket : \mathsf{RGN}\ \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \theta \rrbracket\ \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{type}} \tau \rrbracket$.

(15) If $\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} v : \tau$,
then $\cdot; \cdot; \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{exp}} \mathbb{E}\llbracket \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} v : \tau \rrbracket : \mathbb{T}\llbracket \cdot; \overline{\mathcal{S}} \vdash_{\mathrm{type}} \tau \rrbracket$.

(16) If $\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} w : \tau$,
then $\cdot; \cdot; \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash_{\mathrm{exp}} \mathbb{E}\llbracket \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} w : \mu \rrbracket : \mathbb{T}\llbracket \cdot; \overline{\mathcal{S}} \vdash_{\mathrm{btype}} \mu \rrbracket$.

(17) If $\vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$, then $\vdash_{\mathrm{tower}} \mathbb{X}\llbracket \vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$.

(18) If $\vdash_{\mathrm{prog}} p\ \mathsf{ok}$, then $\cdot; \cdot; \cdot; \cdot \vdash_{\mathrm{exp}} \mathbb{E}\llbracket \vdash_{\mathrm{prog}} p\ \mathsf{ok} \rrbracket : \mathbb{T}\llbracket \cdot, \mathcal{H} \succeq \{\}; \cdot \vdash_{\mathrm{type}} \mathsf{bool} \rrbracket$.

Furthermore, the translation is meaning preserving, with respect to the dynamic semantics of SEC and $\mathsf{F}^{\mathsf{RGN}}$. The essence of this proof relies on a *coherence* lemma stating that the translation of witnesses yields functions that are operationally equivalent to the identify function:

Lemma 5.2 (*Coherence*)
Suppose $\vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$ and $\cdot; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} r \succeq r_i$.
Let $\mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket = \cdot, s \mapsto \overline{\mathcal{S}}^*$, $\mathbb{X}\llbracket \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket = \cdot, s \mapsto \mathcal{S}^*$, $\mathbb{X}\llbracket \vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}} \rrbracket = \cdot, s \mapsto S^*$, and $\mathbb{E}\llbracket \cdot; \overline{\mathcal{S}} \vdash_{\mathrm{rr}} r \succeq r_i \rrbracket = v_w^*$.
If $\cdot; \cdot; \cdot, s \mapsto \mathcal{S}^* : \cdot, s \mapsto \overline{\mathcal{S}}^* \vdash_{\mathrm{exp}} \kappa^* : \mathsf{RGN}\ s \sharp r_i\ \tau_a$ and $\cdot, s \mapsto S^*; \kappa^* \hookrightarrow_\kappa S'^*; v'^*$,
then $\cdot, s \mapsto S^*; v_w^*\ [\tau_a]\ \kappa^* \hookrightarrow \kappa^{*\prime}$ and $\cdot, s \mapsto S^*; \kappa^{*\prime} \hookrightarrow_\kappa S'^*; v'^*$.

Coherence is used throughout the proof of correctness to show that every evaluation derivation for the source can be simulated by a derivation involving the translation of the source:

Theorem 5.1 (*Translation Preserves Semantics*)
Suppose $\vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$, $\cdot; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e : \tau, r'$, and $S; e \hookrightarrow S'; v'$.
Then there exists $\overline{\mathcal{S}}' \sqsupseteq \overline{\mathcal{S}}$ and $\mathcal{S}' \sqsupseteq \mathcal{S}$ such that $\vdash_{\mathrm{stack}} S' : \mathcal{S}' : \overline{\mathcal{S}}'$ and $\mathcal{S}' : \overline{\mathcal{S}}' \vdash_{\mathrm{cval}} v' : \tau$.

Let $\mathbb{X}[\![\overline{\mathcal{S}}]\!] = \cdot, s \mapsto \overline{\mathcal{S}}^*$, $\mathbb{X}[\![\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}]\!] = \cdot, s \mapsto \mathcal{S}^*$, $\mathbb{X}[\![\vdash_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}]\!] = \cdot, s \mapsto S^*$, and $\mathbb{E}[\![\cdot; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, r']\!] = e^*$.
Then $\cdot, s \mapsto S^*; e^* \hookrightarrow \kappa^{*\prime}$ and $\cdot, s \mapsto S^*; \kappa^{*\prime} \hookrightarrow_\kappa S'^*; v'^*$,
where $\mathbb{X}\left[\!\!\left[\overline{\mathcal{S}}'\right]\!\!\right] = \overline{\mathcal{S}}'^*$, $\mathbb{X}\left[\!\!\left[\vdash_{\text{stype}} \mathcal{S}' : \overline{\mathcal{S}}'\right]\!\!\right] = \mathcal{S}'^*$, $\mathbb{X}\left[\!\!\left[\vdash_{\text{stack}} S' : \mathcal{S}' : \overline{\mathcal{S}}'\right]\!\!\right] = S'^*$, and $\mathbb{E}\left[\!\!\left[\mathcal{S}' : \overline{\mathcal{S}}' \vdash_{\text{cval}} v' : \tau\right]\!\!\right] = v'^*$.

We note that the proof is greatly simplified by using large-step operational semantics for both the source and target languages, since for many expression forms, a single operational step in the source language is expanded to many operational steps in the target language.

A simple application of this result shows that the when a source program evaluates to a value, then encapsulating and running its translation also evaluates to the value:

Theorem 5.2 (*Translation Correctness (Programs)*)
Suppose $\vdash_{\text{prog}} e$ ok and $e \hookrightarrow v'$.
Let $\mathbb{E}[\![\vdash_{\text{prog}} e\ \mathsf{ok}]\!] = e^*$.
Then $\cdot; e^* \hookrightarrow v'^*$, where $\mathbb{E}[\![\cdot : \cdot \vdash_{\text{cval}} v' : \mathsf{bool}]\!] = v'^*$.

Full details of this development are given in the report (Fluet, 2004).

6 Expressiveness

An important issue to consider is the expressiveness of the Single Effect Calculus relative to Tofte and Talpin's original region calculus. Tofte and Talpin's formulation of the region calculus as the implicit target of an inference system makes a direct comparison difficult. Fortunately, there has been sufficient interest in region-based memory management to warrant direct presentations of region calculi (Helsen & Thiemann, 2000; Calcagno, 2001; Calcagno *et al.*, 2002; Henglein *et al.*, 2005), which are better suited for comparison. Three aspects of the region calculus are highlighted as essential features: region polymorphism, region polymorphic recursion, and effect polymorphism. Additionally, we believe that region bounds, presented in Cyclone (Grossman *et al.*, 2001) and adopted by the Single Effect Calculus, are a natural generalization that provide additional insight into region calculi.

6.1 Region Polymorphic Recursion

Region polymorphic recursion can be supported in the Single Effect Calculus by adding fix and fixing a region abstraction (as is shown by Henglein, Makholm, and Niss for the Tofte-Talpin region calculus (2005)); Figure 31 presents the extensions to SEC necessary to support fix.

As an example, consider the following term to compute a factorial (in which we elide the type annotation on *fact*):

$$\text{Terms} \quad e \quad ::= \quad \ldots \mid \mathsf{fix}\ f{:}\tau.u$$

$$\boxed{S; e \hookrightarrow S'; e}$$

$$\frac{r \in \mathit{dom}(S) \qquad l \notin \mathit{dom}(S(r))}{S; \mathsf{fix}\ f{:}(\tau_1 \xrightarrow{\theta'} \tau_2, \rho).\lambda x{:}\tau_1.^{\theta'} e'\ \mathsf{at}\ r \hookrightarrow S\{(r,l) \mapsto \lambda x{:}\tau_1.^{\theta'} e'[\langle l\rangle_r/f]\}; \langle l\rangle_r}$$

$$\frac{r \in \mathit{dom}(S) \qquad l \notin \mathit{dom}(S(r))}{S; \mathsf{fix}\ f{:}(\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho).\lambda\varrho \succeq \varphi.^{\theta'} u'\ \mathsf{at}\ r \hookrightarrow S\{(r,l) \mapsto \lambda\varrho \succeq \varphi.^{\theta'} u'[\langle l\rangle_r/f]\}; \langle l\rangle_r}$$

$$\boxed{\Delta; \Gamma; \mathbb{S} : \overline{\mathbb{S}} \vdash_{\text{exp}} e : \tau, \theta}$$

$$\frac{\Delta; \Gamma, f{:}\tau; \mathbb{S} : \overline{\mathbb{S}} \vdash_{\text{exp}} u : \tau, \theta}{\Delta; \Gamma; \mathbb{S} : \overline{\mathbb{S}} \vdash_{\text{exp}} \mathsf{fix}\ f{:}\tau.u : \tau, \theta}$$

Fig. 31. Extensions to SEC for fix

```
fix $fact.(\Pi\varrho_i \succeq \{\}.^{\rho_f} (\Pi\varrho_o \succeq \{\}.^{\rho_f} (\Pi\varrho_b \succeq \{\rho_f, \varrho_i, \varrho_o\}.^{\rho_f}$
        $(\lambda n{:}(\mathsf{int}, \varrho_i).^{\varrho_b}$
         if letregion $\varrho$ in $n \le (1$ at $\varrho)$
          then $1$ at $\varrho_o$
          else letregion $\varrho_{i'}$ in (letregion $\varrho_{o'}$ in
                 $(fact\ [\varrho_{i'}]\ [\varrho_{o'}]\ [\varrho_{o'}]$ (letregion $\varrho$ in $n - (1$ at $\varrho)$ at $\varrho_{i'}))) $* n$ at $\varrho_o$
       ) at $\rho_f$) at $\rho_f$) at $\rho_f$) at $\varrho_f$
```

The function *fact* is parameterized by three regions: ϱ_i is the region in which the input integer is allocated, ϱ_o is the region in which the output integer is to be allocated, and ϱ_b is a region that bounds the latent effect of the function. (Region ρ_f is assumed to be bound in an outer context and holds the closure.) We see that the bounds on ϱ_i and ϱ_o indicate that they are not constrained to be outlived by any other regions. On the other hand, the bound on ϱ_b indicates that ρ_f, ϱ_i, and ϱ_o must outlive ϱ_b. Hence, ϱ_b suffices to bound the effects within the body of the function, in which we expect regions ρ_f (at the recursive call) and ϱ_i to be read from and region ϱ_o to be allocated in. Note that the regions passed to the recursive call of *fact* satisfy the bounds, as $\varrho_{o'}$ outlives ρ_f (through $\varrho_{i'}$ and ϱ_b), $\varrho_{i'}$ is allocated before (and deallocated after) $\varrho_{o'}$, and $\varrho_{o'}$ clearly outlives itself.

This extension of SEC can be translated into $\mathsf{F}^{\mathsf{RGN}}$ extended with $\mathsf{fixRGNLoc}$ (see Section 3.6.2). The translation of Section 5 is extended with the following:

$$\mathbb{E}\left[\!\!\left[\frac{\dfrac{\begin{array}{c}\Delta; \Gamma, f{:}(\tau_1 \xrightarrow{\theta'} \tau_2, \rho), x : \tau_1; \mathbb{S} : \overline{\mathbb{S}} \vdash_{\text{exp}} e' : \tau_2, \theta' \\ \Delta; \overline{\mathbb{S}} \vdash_{\text{place}} \rho \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \theta \succeq \rho\end{array}}{\Delta; \Gamma, f{:}(\tau_1 \xrightarrow{\theta'} \tau_2, \rho); \mathbb{S} : \overline{\mathbb{S}} \vdash_{\text{exp}} \lambda x{:}\tau_1.^{\theta'} e'\ \mathsf{at}\ \rho : (\tau_1 \xrightarrow{\theta'} \tau_2, \rho), \theta}}{\Delta; \Gamma; \mathbb{S} : \overline{\mathbb{S}} \vdash_{\text{exp}} \mathsf{fix}\ f{:}\tau_1 \xrightarrow{\theta'} \tau_2.\lambda x{:}\tau_1.^{\theta'} e'\ \mathsf{at}\ \rho : (\tau_1 \xrightarrow{\theta'} \tau_2, \rho), \theta} \right]\!\!\right] =$$

$$\mathbb{E}[\![\theta \succeq \rho]\!]\ \mathbb{T}\left[\!\!\left[(\tau_1 \xrightarrow{\theta'} \tau_2, \rho)\right]\!\!\right]$$

$$(\mathsf{fixRGNLoc}\ [\mathbb{T}[\![\rho]\!]]\ [\mathbb{T}\left[\!\!\left[\tau_1 \xrightarrow{\theta'} \tau_2\right]\!\!\right]]$$

$$\mathbb{E}[\![\rho]\!]\ \ (\lambda f : \mathbb{T}\left[\!\!\left[(\tau_1 \xrightarrow{\theta'} \tau_2, \rho)\right]\!\!\right].\lambda x : \mathbb{T}[\![\tau_1]\!].\mathbb{E}[\![e]\!]))$$

$$\mathbb{E}\left[\!\!\left[\frac{\dfrac{\begin{array}{c}\Delta, \varrho \succeq \varphi; \Gamma, f{:}(\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho); \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} u' : \tau, \theta' \\ \Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \rho \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \theta \succeq \rho\end{array}}{\Delta; \Gamma, f{:}(\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho); \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \lambda\varrho \succeq \varphi.^{\theta'} u' \ \mathsf{at}\ \rho : (\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho), \theta}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \mathsf{fix}\ f{:}(\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho).\lambda\varrho \succeq \varphi.^{\theta'} u' \ \mathsf{at}\ \rho : (\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho), \theta}\right]\!\!\right] =$$

$$\begin{array}{l}\mathbb{E}[\![\theta \succeq \rho]\!]\ \mathbb{T}\left[\!\!\left[(\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho)\right]\!\!\right] \\ \qquad (\mathsf{fixRGNLoc}\ [\mathbb{T}[\![\rho]\!]]\ [\mathbb{T}\left[\!\!\left[\Pi\varrho \succeq \varphi.^{\theta'} \tau\right]\!\!\right]] \\ \qquad\qquad \mathbb{E}[\![\rho]\!]\ (\lambda f : \mathbb{T}\left[\!\!\left[(\Pi\varrho \succeq \varphi.^{\theta'} \tau, \rho)\right]\!\!\right]. \\ \qquad\qquad\qquad \Lambda\varrho.\lambda w_\varrho{:}\mathbb{T}[\![\varrho \succeq \theta]\!]\,.\lambda h_\varrho : \mathsf{RGNHandle}\ \varrho.\mathbb{E}[\![u]\!])\end{array}$$

6.2 Effect Polymorphism

Recall that effect polymorphism provides a means to abstract over an effect (a set of regions). Effect instantiation applies an effect abstraction to an effect. Effect polymorphism is especially useful for typing higher-order functions. For example, the type of the list `map` function should be polymorphic in the effect of the functional argument. We note that effect polymorphism is most useful in the presence of type polymorphism. While we have presented the region calculi as a monomorphic languages, adding type polymorphism is entirely orthogonal to the development thus far.

Our translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ was simplified by using a single type (variable) for the index of the RGN monad. As the translation eliminates subtyping through coercions, we introduced the type of witness functions. One interesting aspect of the current translation is that there are no terms for nontrivial witnesses between RGN computations in (the surface syntax of) $\mathsf{F}^{\mathsf{RGN}}$. The only way to acquire a term of type $\tau_r \preceq \tau_s \equiv \forall\beta.\mathsf{RGN}\ \tau_r\ \beta \rightarrow \mathsf{RGN}\ \tau_s\ \beta$ is through letRGN. The two trivial casts (corresponding to reflexivity and transitivity) can be written in (pure) System F:

$$\begin{array}{lcl}\textit{refl} & :: & \forall\gamma_r.\gamma_r \preceq \gamma_r \\ \textit{refl} & \equiv & \Lambda\gamma_r.\Lambda\beta.\lambda k{:}\mathsf{RGN}\ \gamma_r\ \beta.k \\ \\ \textit{trans} & :: & \Lambda\gamma_{r_1}, \gamma_{r_2}, \gamma_{r_3}.(\gamma_{r_1} \preceq \gamma_{r_2}) \rightarrow (\gamma_{r_2} \preceq \gamma_{r_3}) \rightarrow (\gamma_{r_1} \preceq \gamma_{r_3}) \\ \textit{trans} & \equiv & \Lambda\gamma_{r_1}, \gamma_{r_2}, \gamma_{r_3}.\lambda f{:}\gamma_{r_1} \preceq \gamma_{r_2}.\lambda g{:}\gamma_{r_2} \preceq \gamma_{r_3}.\Lambda\beta.\lambda k{:}\mathsf{RGN}\ \gamma_{r_1}\ \beta.g\ [\beta]\ (f\ [\beta]\ k)\end{array}$$

If we were to adopt a source calculus with effects given by

$$\varphi ::= \emptyset \mid \{\rho\} \mid \varepsilon \mid \varphi_1 \cup \varphi_2$$

where effects may be any combination of regions and effect variables, then an "obvious" translation would be something like:

$$\begin{array}{rcl}\mathbb{T}[\![\emptyset]\!] & = & \mathsf{unit} \\ \mathbb{T}[\![\{\rho\}]\!] & = & \rho \\ \mathbb{T}[\![\varepsilon]\!] & = & \varepsilon \\ \mathbb{T}[\![\varphi_1 \cup \varphi_2]\!] & = & \mathbb{T}[\![\varphi_1]\!] \times \mathbb{T}[\![\varphi_2]\!]\end{array}$$

Now we would require some interpretation for effect relations. For example, the rule

$$\frac{\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_1 \qquad \Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_2}{\Delta \vdash_{\text{ee}} \varphi \supseteq \varphi_1 \cup \varphi_2}$$

needs to translate to a term with the type

$$\begin{aligned}&(\mathbb{T}[\![\varphi_1]\!] \preceq \mathbb{T}[\![\varphi]\!]) \to (\mathbb{T}[\![\varphi_2]\!] \preceq \mathbb{T}[\![\varphi]\!]) \to (\mathbb{T}[\![\varphi_1 \cup \varphi_2]\!] \preceq \mathbb{T}[\![\varphi]\!])\\ &\quad\equiv (\mathbb{T}[\![\varphi_1]\!] \preceq \mathbb{T}[\![\varphi]\!]) \to (\mathbb{T}[\![\varphi_2]\!] \preceq \mathbb{T}[\![\varphi]\!]) \to (\mathbb{T}[\![\varphi_1]\!] \times \mathbb{T}[\![\varphi_2]\!] \preceq \mathbb{T}[\![\varphi]\!])\end{aligned}$$

which can be seen as an instance of the type

$$\forall \gamma, \gamma_1, \gamma_2.(\gamma_1 \preceq \gamma) \to (\gamma_2 \preceq \gamma) \to (\gamma_1 \times \gamma_2 \preceq \gamma).$$

As there is no (pure) System F term with that type, we would need to introduce a number of witness terms (at the surface syntax) in order to accomodate all of the necessary coercions.

Simply put, in witnessing "sub-effecting" through explicit coercions, we need to introduce additional terms into the language. We note that the situation is really no better in a language with subtyping (e.g., System $F_{\leq}$), as the subset relation is "richer" than the subtype relation on standard types (e.g., product types).

7 Related Work

The work in this paper draws heavily from two lines of research. The first is the work done in type-safe region-based memory management, introduced by Tofte and Talpin (1994; 1997). Our Single Effect Calculus draws inspiration from the Capability Calculus (Crary *et al.*, 1999) and Cyclone (Grossman *et al.*, 2001), where the "outlives" relationship between regions is recognized as an important component of type systems for region calculi.

The work of Banerjee, Heintze, and Riecke (1999) deserves special mention. They show how to translate the Tofte-Talpin region calculus into an extension of the polymorphic λ-calculus called $F_{\#}$. A new type operator $\#$ is used as a mechanism to hide and reveal the structure of types. Capabilities to allocate and read values from a region are explicitly passed as polymorphic functions of types $\forall\alpha.\alpha \to (\alpha\#\rho)$ and $\forall\alpha.(\alpha\#\rho) \to \alpha$; however, regions have no run-time significance in $F_{\#}$ and there is no notion of deallocation upon exiting a region. The equality theory of types in $F_{\#}$ is nontrivial, due to the treatment of #; in contrast, type equality on F^{RGN} types is purely syntactic. Furthermore, their proof of soundness is based on denotational techniques, whereas ours are based on syntactic techniques which tend to scale more easily to other linguistic features. Finally, it is worth noting that there is almost certainly a connection between the $F_{\#}$ lift and seq expressions and the monadic return and bind operations, although it is not mentioned or explored in their paper.

The second line of research on which we draw is the work done in monadic encapsulation of effects (Moggi, 1989; Moggi, 1991; Riecke & Viswanathan, 1995; Launchbury & Peyton Jones, 1995; Wadler, 1995; Launchbury & Peyton Jones, 1994; Launchbury & Sabry, 1997; Sabry & Wadler, 1997; Ariola & Sabry, 1998; Kieburtz, 1998; Semmelroth & Sabry, 1999; Moggi & Sabry, 2001; Wadler & Thiemann, 2003). The majority of this work has focused on effects arising from reading and writing mutable state, which we reviewed in Section 2. While recent work (Wadler, 1995; Moggi & Sabry, 2001; Wadler & Thiemann, 2003) has considered more general combinations of effects and monads, only a small amount of work has examined the combination of regions and monads (Kagawa, 1997; Kagawa, 2001; Ganz, forthcoming).

We note that Wadler and Thiemann (2003) advocate marrying effects and monads by translating a type $\tau_1 \xrightarrow{\sigma} \tau_2$ to the type $\mathbb{T}[\![\tau_1]\!] \rightarrow \mathsf{T}^\sigma\ \mathbb{T}[\![\tau_2]\!]$, where $\mathsf{T}^\sigma\ \tau$ represents a computation that yields a value of type τ and has effects delimited by (the set) σ. As with the work of Banerjee et. al. described above, this introduces a nontrivial theory of equality (and subtyping) on types; the types $\mathsf{T}^\sigma\ \tau$ and $\mathsf{T}^{\sigma'}\ \tau$ are equal so long as σ and σ' are equivalent sets. However, few programming languages allow one to express such nontrivial equalities between types.

Kagawa (1997; 2001) anticipates a number of themes from this work, although a formal treatment is left to future work. As a means of bridging the work of Wadler (1992) and Launchbury and Peyton Jones (1995), Kagawa (1997) suggests extending the ST monad with the following type and operations:

$$\tau \quad ::= \quad \ldots \mid \mathsf{Mutable}\ \tau_s\ \tau_t$$

$$\begin{aligned}
\mathsf{appR} &:: \forall s, t.\forall a.\mathsf{Mutable}\ s\ t \rightarrow \mathsf{ST}\ t\ a \rightarrow \mathsf{ST}\ s\ a \\
\mathsf{cmpR} &:: \forall s, t, u.\mathsf{Mutable}\ s\ t \rightarrow \mathsf{Mutable}\ t\ u \rightarrow \mathsf{Mutable}\ s\ u \\
\mathsf{extendST} &:: \forall t.\forall a.(\forall s.\mathsf{Mutable}\ s\ t \rightarrow \mathsf{ST}\ t\ a) \rightarrow \mathsf{ST}\ t\ a
\end{aligned}$$

The intention is that the type Mutable s t is equivalent to the type $(s \rightarrow t) \times (s \rightarrow t \rightarrow s)$; hence, it serves as a witness to the embedding of the state t into a larger state s. extendST generalizes blockST of Section 2 in the same manner as our letRGN. appR coerces a state transformer, given the appropriate witness, while cmpR composes witnesses; hence, the latter is a "proof" of the transitivity of the state embedding. In our setting, the transparency of the $r_1 \preceq r_2$ type obviates the need for these explicit operations. The lack of formal dynamic and static semantics makes a thorough evaluation difficult; in particular, the relationship between the global state "conjured up" by runST and an individual mutable object is rather *ad hoc*. In the computation syntax of $\mathsf{F}^{\mathsf{RGN}}$, a witnessRGN term concretely captures the relationship between an older and a younger region.

In later work, Kagawa (2001) argues that these techniques can be extended to accomodate region-based memory management. In spite of the title and notation, the paper does not present an explicitly monadic language. Rather, the language is presented with a type-and-effect system, and the connection to a monadic setting is left (vaguely) implicit in the choice of notation and reference to the previous work. A dynamic semantics and type system, along with a proof that the new letextend operator can safely deallocate the extended region, is left to future work. Our present work addresses all these deficiencies by giving clear descriptions of both the Single Effect Calculus and $\mathsf{F}^{\mathsf{RGN}}$, proving the soundness of the $\mathsf{F}^{\mathsf{RGN}}$ type system, and giving a type- and meaning-preserving translation between the two languages. On the other hand, Kagawa presents a type inference algorithm for the language, which may suggest a means of reducing the notational overhead of passing witnesses and handles.

Ganz (forthcoming) relates the type-and-effect system of Tofte and Talpin to monad transformers. Ganz distinguishes among encapsulation with a single monad, encapsulation with a monad per region, and encapsulation with a monad transformer per region. He concludes that only a monad transformer per region is expressive enough to encode nested regions. This corresponds to our presentation where runRGN introduces a monad per stack of regions and letRGN introduces a monad transformer per region. Ganz imposes a peculiar restriction: upward references (i.e., allocating a reference to an inner region at an outer region) are not allowed. This is a severe restriction for a region-based language; it appears to arise from a failure to distinguish encapsulation of a stack of regions from encapsulation of a single region. Recall that while runRGN computations may be nested, it is not possible for the outer computation to have references to the inner computation; on the other hand, there may be arbitrary references among regions of a single stack. Finally, Ganz claims to support early deallocation of regions, a facet of region-based memory management that is not available in $\mathsf{F}^{\mathsf{RGN}}$.

Finally, other researchers have utilized the power of System F as a target language. For example, Washburn and Weirich (2003) demonstrate how to encode higher-order abstract syntax using parametric polymorphism, while Tse and Zdancewic (2004) show how to encode the dependency core calculus.

8 Conclusions and Future Work

We have given a type- and meaning-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. Both the source and the target calculi use a static type-system to delimit the effects of allocating in and reading from regions. The Single Effect Calculus uses the partial order implied by the "outlives" relation on regions to use single regions as bounds for sets of effects. We feel that this is an important insight that leads to a relatively straight-forward translation into the monadic setting. $\mathsf{F}^{\mathsf{RGN}}$ draws from the work on monadic encapsulation of state to give parametric types to runRGN and letRGN that prevent access of regions beyond their lifetimes. Explicit functions witness the outlives relationship between regions, enabling computations from outer regions to be cast to computations in inner regions. Witness functions cannot be forged and are only introduced via letRGN.

There are numerous directions for future work. One idea is to provide the RGN monad to Haskell programmers and to try to leverage type classes so that witnesses and handles can be passed implicitly, thereby reducing the notational overhead of programming with nested stores. While a direct encoding of subtyping leads to undecidable and overlapping instances, the use of type-indexed products (Kiselyov *et al.*, 2004) may provide a partial solution, at the expense of reintroducing a product type (see comments at the end of Section 2). Obviously, a language that incorporates subtyping directly, such as $\mathsf{F}_{\leq}$, would simplify the encoding.

Finally, as is well known, Tofte and Talpin's original region calculus can lead to inefficient memory usage for some programs. In practice, additional mechanisms are required to achieve good space utilization. Cyclone incorporates a number of these enhancements, including unique pointers and dynamic regions, and it remains to be seen whether these features can also be encoded into a simpler setting.

Acknowledgements

A preliminary version of this work appeared in the informal *Proceedings of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for*

Memory Management (SPACE'04). Thanks to Oleg Kiselyov for suggestions on using Haskell type classes to pass handles and witnesses. The anonymous reviewers provided helpful comments that have improved the presentation.

Supported in part by National Science Foundation Grants 0204193 and 9875536, AFOSR Grants F49620-03-1-0156 and F49620-01-1-0298, and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

Ariola, Zena, & Sabry, Amyr. (1998). Correctness of monadic state: An imperative call-by-need calculus. *Pages 62–74 of: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*. San Diego, CA: ACM Press.

Banerjee, Anindya, Heintze, Nevin, & Riecke, Jon. (1999). Region analysis and the polymorphic lambda calculus. *Pages 88–97 of: Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LCS'99)*. Trento, Italy: IEEE Computer Society Press.

Calcagno, Cristiano. (2001). Stratified operational semantics for safety and correctness of the region calculus. *Pages 155–165 of: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. London, England: ACM Press.

Calcagno, Cristiano, Helsen, Simon, & Thiemann, Peter. (2002). Syntactic type soundness results for the region calculus. *Information and Computation*, **173**(2), 199–332.

Crary, Karl, Walker, David, & Morrisett, Greg. (1999). Typed memory management in a calculus of capabilities. *Pages 262–275 of: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*. ACM Press.

Fluet, Matthew. 2004 (Apr.). *Monadic regions: Formal type soundness and correctness.* Tech. rept. TR2004-1936. Department of Computer Science, Cornell University.

Fluet, Matthew, & Morrisett, Greg. (2004). Monadic regions. *Pages 103–114 of: Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*. ACM Press.

Ganz, Steven E. (forthcoming). *Monadic encapsulation of state.* Ph.D. thesis, Indiana University, Bloomington, Indiana.

Girard, Jean-Yves, Taylor, Paul, & Lafont, Yves. (1989). *Proofs and types.* Cambridge University Press.

Grossman, Dan, Morrisett, Greg, Wang, Yanling, Jim, Trevor, Hicks, Michael, & Cheney, James. 2001 (Nov.). *Formal type soundness for Cyclone's region system.* Tech. rept. 2001-1856. Department of Computer Science, Cornell University.

Grossman, Dan, Morrisett, Greg, Jim, Trevor, Hicks, Michael, Wang, Yanling, & Cheney, James. (2002). Region-based memory management in Cyclone. *Pages 282–293 of: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press.

Helsen, Simon, & Thiemann, Peter. (2000). Syntactic type soundness for the region calculus. *Pages 1–19 of: Proceedings of the 4th International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'00)*. Electronic Notes in Theoretical Computer Science, vol. 41. Montreal, Canada: Elsevier Science Publishers.

Henglein, Fritz, Makholm, Henning, & Niss, Henning. (2005). Effect types and region-based memory management. *Chap. 3, pages 87–135 of:* Pierce, Benjamin (ed), *Advanced Topics in Types and Programming Languages.* Cambridge, MA: MIT Press.

Kagawa, Koji. (1997). Compositional references for stateful functional programming. *Pages 217–226 of: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97).* ACM Press.

Kagawa, Koji. (2001). Monadic encapsulation with stack of regions. *Pages 264–279 of: Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'01).* Lecture Notes in Computer Science, vol. 2024. Tokyo, Japan: Springer-Verlag.

Kieburtz, Richard. (1998). Taming effects with monadic typing. *Pages 51–62 of: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98).* Baltimore, MD: ACM Press.

Kiselyov, Oleg, Lämmel, Ralf, & Schupke, Keean. (2004). Strongly typed heterogeneous collections. *Pages 96–107 of: Proceedings of the ACM SIGPLAN Workshop on Haskell.* ACM Press.

Launchbury, John, & Peyton Jones, Simon. (1994). Lazy functional state threads. *Pages 24–35 of: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94).* Orlando, FL: ACM Press.

Launchbury, John, & Peyton Jones, Simon. (1995). State in Haskell. *Lisp and Symbolic Computation*, **8**(4), 293–341.

Launchbury, John, & Sabry, Amr. (1997). Monadic state: Axiomatization and type safety. *Pages 227–237 of: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97).* Amsterdam, The Netherlands: ACM Press.

Moggi, Eugino. (1989). Computational lambda calculus and monads. *Pages 14–23 of: Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LCS'89).*

Moggi, Eugino. (1991). Notions of computation and monads. *Information and Computation*, **93**(1), 55–92.

Moggi, Eugino, & Sabry, Amr. (2001). Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming*, **11**(6), 591–627.

Reynolds, John. (1974). Towards a theory of type structure. *Pages 408–425 of: Programming symposium.* Lecture Notes in Computer Science, vol. 19. Paris, France: Springer-Verlag.

Riecke, Jon, & Viswanathan, Ramesh. (1995). Isolating side effects in sequential languages. *Pages 1–12 of: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95).* San Francisco, CA: ACM Press.

Sabry, Amr, & Wadler, Philip. (1997). A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, **19**(6), 916–941.

Semmelroth, Miley, & Sabry, Amr. (1999). Monadic encapsulation in ML. *Pages 8–17 of: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99).* Paris, France: ACM Press.

Smith, Geoffrey, & Volpano, Dennis. (1998). A sound polymorphic type system for a dialect of C. *Science of computer programming*, **32**(1-3), 49–72.

Tofte, Mads, & Talpin, Jean-Pierre. (1994). Implementation of the typed call-by-value λ-calculus using a stack of regions. *Pages 188–201 of: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94).* Portland, OR: ACM Press.

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

Tofte, Mads, Birkedal, Lars, Elsman, Martin, Hallenberg, Niels, Olesen, Tommy Højfeld,

& Sestoft, Peter. 2002 (Apr.). *Programming with regions in the ML Kit (for version 4)*. Tech. rept. IT University of Copenhagen.

Tse, Stephen, & Zdancewic, Steve. (2004). Translating dependency into parametricity. *Pages 115–1125 of: Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*. ACM Press.

Volpano, Dennis, & Smith, Geoffrey. (1997). Eliminating covert flows with minimum typings. *Pages 156–168 of: Proceedings of the 10th IEEE Computer Security Foundations Workshop (CFSW'97)*. IEEE Computer Society Press.

Wadler, Philip. (1992). The essence of functional programming. *Pages 1–14 of: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*. ACM Press.

Wadler, Philip. (1995). The marriage of effects and monads. *Pages 63–74 of: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. Baltimore, MD: ACM Press.

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

Washburn, Geoffrey, & Weirich, Stephanie. (2003). Boxes go bannanas: Encoding higher-order abstract syntax with parametric polymorphism. *Pages 249 – 262 of: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*. ACM Press.

A Static Semantics of $\mathsf{F}^{\mathsf{RGN}}$

Figures A 1 and A 2 contain additional judgements for the static semantics of $\mathsf{F}^{\mathsf{RGN}}$.

$$\boxed{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau}$$

$$\frac{\vdash_{\mathrm{tctxt}} \Delta}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \mathsf{int}} \qquad \frac{\vdash_{\mathrm{tctxt}} \Delta}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \mathsf{bool}} \qquad \frac{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_1 \Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_2}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_1 \to \tau_2}$$

$$\frac{\vdash_{\mathrm{tctxt}} \Delta \qquad \Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_i \; {}^{i \in 1 \ldots n}}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_1 \times \cdots \times \tau_n} \qquad \frac{\vdash_{\mathrm{tctxt}} \Delta \qquad \alpha \in \mathit{dom}(\Delta)}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \alpha} \qquad \frac{\Delta, \alpha; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \forall \alpha . \tau}$$

$$\frac{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_r \qquad \Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_a}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \mathsf{RGN}\ \tau_r\ \tau_a} \qquad \frac{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_r \qquad \Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_a}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \mathsf{RGNLoc}\ \tau_r\ \tau_a}$$

$$\frac{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau_r}{\Delta; \overline{\mathfrak{T}} \vdash \mathsf{RGNHandle}\ \tau_r} \qquad \frac{\vdash_{\mathrm{tctxt}} \Delta}{\Delta; \overline{\mathfrak{T}} \vdash \circ \sharp \bullet} \qquad \frac{\vdash_{\mathrm{tctxt}} \Delta \qquad s \in \mathit{dom}(\overline{\mathfrak{T}})}{\Delta; \overline{\mathfrak{T}} \vdash s \sharp \bullet}$$

$$\frac{\vdash_{\mathrm{tctxt}} \Delta \qquad s \in \mathit{dom}(\overline{\mathfrak{T}}) \qquad r \in \mathit{dom}(\overline{\mathfrak{T}}(s))}{\Delta; \overline{\mathfrak{T}} \vdash s \sharp r}$$

Fig. A 1. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (types)

$$\boxed{\vdash_{\mathrm{tctxt}} \Delta}$$

$$\frac{}{\vdash_{\mathrm{tctxt}} \cdot} \qquad \frac{\vdash_{\mathrm{tctxt}} \Delta \qquad \alpha \notin \mathit{dom}(\Delta)}{\vdash_{\mathrm{tctxt}} \Delta, \alpha}$$

$$\boxed{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{vctxt}} \Gamma}$$

$$\frac{\vdash_{\mathrm{tctxt}} \Delta}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{vctxt}} \cdot} \qquad \frac{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{vctxt}} \Gamma \qquad x \notin \mathit{dom}(\Gamma) \qquad \Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{type}} \tau}{\Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{vctxt}} \Gamma, x : \tau}$$

$$\boxed{\vdash_{\mathrm{ctxt}} \Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}}}$$

$$\frac{\vdash_{\mathrm{ttype}} \mathfrak{T} : \overline{\mathfrak{T}} \qquad \Delta; \overline{\mathfrak{T}} \vdash_{\mathrm{vctxt}} \Gamma}{\vdash_{\mathrm{ctxt}} \Delta; \Gamma; \mathfrak{T} : \overline{\mathfrak{T}}}$$

Fig. A 2. Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (contexts)

B Static Semantics of SEC

Figures B 1, B 2, B 3, and B 4 contain additional judgements for the static semantics of the Single Effect Calculus.

The judgement $\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}$ asserts that stack type $\mathcal{S}$ is well-formed with stack domain $\overline{\mathcal{S}}$. In particular, the judgement asserts that $\mathcal{S}$ has the domain specified by $\overline{\mathcal{S}}$ and each (boxed) type in the range of $\mathcal{S}$ is well-formed. Note that the judgement is made with respect to the entire stack domain $\overline{\mathcal{S}}$. This allows types "lower" in the stack to reference region names that appear "higher" in the stack. This corresponds to the fact that one can have arbitrary pointers between region allocated data. Finally, the judgement $\vdash_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$ asserts that the stack S is well-formed with stack type $\mathcal{S}$ and stack domain $\overline{\mathcal{S}}$. Like the judgement $\vdash_{\text{stype}}$, it asserts that S has the domain specified by $\overline{\mathcal{S}}$ and each stored value in the range of S has the type specified by $\mathcal{S}$.

$$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \rho}$$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \in \mathit{dom}(\Delta)}{\Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \varrho} \qquad \frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad r \in \mathit{dom}(\overline{\mathcal{S}})}{\Delta; \overline{\mathcal{S}} \vdash_{\text{place}} r} \qquad \frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \bullet}$$

$$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \varphi}$$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \{\rho_1, \ldots, \rho_n\}}$$

$$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \mu}$$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \mathsf{int}} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_1 \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \theta \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_2}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \tau_1 \xrightarrow{\theta} \tau_2}$$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_1 \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_2}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \tau_1 \times \tau_2}$$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \varphi \qquad \Delta, \varrho \succeq \varphi; \overline{\mathcal{S}} \vdash_{\text{place}} \theta \qquad \Delta, \varrho \succeq \varphi; \overline{\mathcal{S}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \Pi \varrho \succeq \varphi. {}^{\theta}\tau}$$

$$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau}$$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \mathsf{bool}} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \mu \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{place}} \rho}{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} (\mu, \rho)}$$

Fig. B 1. Static semantics of SEC (types)

$\boxed{\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta}$

$$\frac{}{\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \cdot} \qquad \frac{\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta \quad \varrho \notin \mathit{dom}(\Delta) \quad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{eff}} \varphi}{\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta, \varrho \succeq \varphi}$$

$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma}$

$$\frac{\overline{\mathcal{S}} \vdash_{\mathrm{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \cdot} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma \quad x \notin \mathit{dom}(\Gamma) \quad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{type}} \tau}{\Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma, x : \tau}$$

$\boxed{\vdash_{\mathrm{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \theta}$

$$\frac{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \quad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{vctxt}} \Gamma \quad \Delta; \overline{\mathcal{S}} \vdash_{\mathrm{place}} \theta}{\vdash_{\mathrm{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \theta}$$

Fig. B 2. Static semantics of SEC (contexts)

$\boxed{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} v : \tau}$

$$\frac{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} \mathsf{tt} : \mathsf{bool}} \qquad \frac{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} \mathsf{ff} : \mathsf{bool}} \qquad \frac{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \quad \cdot; \overline{\mathcal{S}} \vdash_{\mathrm{btype}} \mu}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} \langle l \rangle_{\bullet} : (\mu, \bullet)}$$

$$\frac{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \quad r \in \mathit{dom}(\overline{\mathcal{S}}) \quad l \in \mathit{dom}(\overline{\mathcal{S}}(r)) \quad \mu = \mathcal{S}(r, l)}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} \langle l \rangle_r : (\mu, r)}$$

$\boxed{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} w : \mu}$

$$\frac{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} i : \mathsf{int}} \qquad \frac{\cdot; \cdot, x{:}\tau_1; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} e' : \tau_2, \theta'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} \lambda x{:}\tau_1.^{\theta'} e' : \tau_1 \xrightarrow{\theta'} \tau_2}$$

$$\frac{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} v_1 : \tau_1 \quad \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{cval}} v_2 : \tau_2}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} (v_1, v_2) : \tau_1 \times \tau_2} \qquad \frac{\cdot, \varrho \succeq \varphi; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{exp}} u' : \tau, \theta'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} \lambda \varrho \succeq \varphi.^{\theta'} u' : \Pi \varrho \succeq \varphi.^{\theta'} \tau}$$

Fig. B 3. Static semantics of SEC (closed and storable values)

$\boxed{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}}$

$$\frac{\begin{array}{c} \mathit{dom}(\overline{\mathcal{S}}) = \mathit{dom}(\mathcal{S}) \\ \forall r \in \mathit{dom}(\overline{\mathcal{S}}).\ \mathit{dom}(\overline{\mathcal{S}}(r)) = \mathit{dom}(\mathcal{S}(r)) \\ \forall r \in \mathit{dom}(\overline{\mathcal{S}}).\forall l \in \mathit{dom}(\overline{\mathcal{S}}(r)).\ \cdot; \overline{\mathcal{S}} \vdash_{\mathrm{btype}} \mathcal{S}(r, l) \end{array}}{\vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}}}$$

$\boxed{\vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}}}$

$$\frac{\begin{array}{c} \vdash_{\mathrm{stype}} \mathcal{S} : \overline{\mathcal{S}} \\ \mathit{dom}(\overline{\mathcal{S}}) = \mathit{dom}(\mathcal{S}) = \mathit{dom}(S) \\ \forall r \in \mathit{dom}(\overline{\mathcal{S}}).\ \mathit{dom}(\overline{\mathcal{S}}(r)) = \mathit{dom}(\mathcal{S}(r)) = \mathit{dom}(S(r)) \\ \forall r \in \mathit{dom}(\overline{\mathcal{S}}).\forall l \in \mathit{dom}(\overline{\mathcal{S}}(r)).\ \mathcal{S} : \overline{\mathcal{S}} \vdash_{\mathrm{sto}} S(r, l) : (\mathcal{S}(r, l), r) \end{array}}{\vdash_{\mathrm{stack}} S : \mathcal{S} : \overline{\mathcal{S}}}$$

Fig. B 4. Static semantics of SEC (stacks and regions)

C The Translation

Figures C 1, C 2, C 3, and C 4 contain additional translations from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$.

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\vdash_{\text{ctxt}} \Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}};\theta \\ \Delta;\overline{\mathbb{S}} \vdash_{\text{place}} \rho \qquad \Delta;\overline{\mathbb{S}} \vdash_{\text{rr}} \theta \succeq \rho\end{array}}{\Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} i \;\mathsf{at}\; \rho : (\mathsf{int},\rho),\theta}\right]\!\!\right] =$$

$$\mathbb{E}[\![\theta \succeq \rho]\!] \; [\mathbb{T}[\![(\mathsf{int},\rho)]\!]] \; (\mathsf{newRGNLoc}\; [\mathbb{T}[\![\rho]\!]] \; [\mathbb{T}[\![\mathsf{int}]\!]] \; \mathbb{E}[\![\rho]\!] \; i)$$

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} e_1 : (\mathsf{int},\rho_1),\theta \qquad \Delta;\overline{\mathbb{S}} \vdash_{\text{rr}} \theta \succeq \rho_1 \\ \Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} e_2 : (\mathsf{int},\rho_2),\theta \qquad \Delta;\overline{\mathbb{S}} \vdash_{\text{rr}} \theta \succeq \rho_2\end{array}}{\Delta;\Gamma \vdash_{\text{exp}} e_1 \otimes e_2 : \mathsf{bool},\theta}\right]\!\!\right] =$$

$$\begin{array}{l}
\mathsf{bind}\; a{:}\mathbb{T}[\![(\mathsf{int},\rho_1)]\!] \Leftarrow \mathbb{E}[\![e_1]\!]\,;\\
\mathsf{bind}\; a'{:}\mathbb{T}[\![\mathsf{int}]\!] \Leftarrow \mathbb{E}[\![\theta \succeq \rho_1]\!] \; [\mathbb{T}[\![\mathsf{int}]\!]] \; (\mathsf{readRGNLoc}\; [\mathbb{T}[\![\rho_1]\!]] \; [\mathbb{T}[\![\mathsf{int}]\!]] \; a);\\
\mathsf{bind}\; b{:}\mathbb{T}[\![(\mathsf{int},\rho_2)]\!] \Leftarrow \mathbb{E}[\![e_2]\!]\,;\\
\mathsf{bind}\; b'{:}\mathbb{T}[\![\mathsf{int}]\!] \Leftarrow \mathbb{E}[\![\theta \succeq \rho_2]\!] \; [\mathbb{T}[\![\mathsf{int}]\!]] \; (\mathsf{readRGNLoc}\; [\mathbb{T}[\![\rho_2]\!]] \; [\mathbb{T}[\![\mathsf{int}]\!]] \; b);\\
\mathsf{let}\; z = a' \otimes b' \;\mathsf{in}\\
\mathsf{returnRGN}\; [\mathbb{T}[\![\theta]\!]] \; [\mathbb{T}[\![\mathsf{bool}]\!]] \; z
\end{array}$$

where a, a', b, b', z fresh

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}};\theta}{\Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} \mathsf{tt} : \mathsf{bool},\theta}\right]\!\!\right] = \mathsf{returnRGN}\; [\mathbb{T}[\![\theta]\!]] \; [\mathbb{T}[\![\mathsf{bool}]\!]] \; \mathsf{tt}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}};\theta}{\Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} \mathsf{ff} : \mathsf{bool},\theta}\right]\!\!\right] = \mathsf{returnRGN}\; [\mathbb{T}[\![\theta]\!]] \; [\mathbb{T}[\![\mathsf{bool}]\!]] \; \mathsf{ff}$$

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} e_b : \mathsf{bool},\theta \\ \Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} e_t : \tau,\theta \qquad \Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} e_f : \tau,\theta\end{array}}{\Delta;\Gamma;\mathbb{S}:\overline{\mathbb{S}} \vdash_{\text{exp}} \mathsf{if}\; e_b \;\mathsf{then}\; e_t \;\mathsf{else}\; e_f : \tau,\theta}\right]\!\!\right] =$$

$$\mathsf{bind}\; z{:}\mathbb{T}[\![\mathsf{bool}]\!] \Leftarrow \mathbb{E}[\![e_b]\!]\,; \mathsf{if}\; z \;\mathsf{then}\; \mathbb{E}[\![e_t]\!] \;\mathsf{else}\; \mathbb{E}[\![e_f]\!]$$

where z fresh

Fig. C 1. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Terms (III))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} e_1 : \tau_1,\theta \\ \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} e_2 : \tau_2,\theta \\ \Delta;\overline{\mathcal{S}}\vdash_{\text{place}}\rho \qquad \Delta;\overline{\mathcal{S}}\vdash_{\text{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} (e_1,e_2)\ \mathsf{at}\ \rho : (\tau_1\times\tau_2,\rho),\theta}\right]\!\!\right] =$$

$$\begin{array}{l}\mathsf{bind}\ a{:}\mathbb{T}[\![\tau_1]\!] \Leftarrow \mathbb{E}[\![e_1]\!]\,; \\ \mathsf{bind}\ b{:}\mathbb{T}[\![\tau_2]\!] \Leftarrow \mathbb{E}[\![e_2]\!]\,; \\ \mathbb{E}[\![\theta\succeq\rho]\!]\ [\mathbb{T}[\![(\tau_1\times\tau_2,\rho)]\!]]\ (\mathsf{newRGNLoc}\ [\mathbb{T}[\![\rho]\!]]\ [\mathbb{T}[\![\tau_1\times\tau_2]\!]]\ \mathbb{E}[\![\rho]\!]\ (a,b))\end{array}$$

where a, b fresh

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} e : (\tau_1\times\tau_2,\rho),\theta \\ \Delta;\overline{\mathcal{S}}\vdash_{\text{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} \mathsf{fst}\ \ e : \tau_1,\theta}\right]\!\!\right] =$$

$$\begin{array}{l}\mathsf{bind}\ x{:}\mathbb{T}[\![(\tau_1\times\tau_2,\rho)]\!] \Leftarrow \mathbb{E}[\![e]\!]\,; \\ \mathsf{bind}\ y{:}\mathbb{T}[\![\tau_1\times\tau_2]\!] \Leftarrow \mathbb{E}[\![\theta\succeq\rho]\!]\ [\mathbb{T}[\![\tau_1\times\tau_2]\!]]\ (\mathsf{readRGNLoc}\ [\mathbb{T}[\![\rho]\!]]\ [\mathbb{T}[\![\tau_1\times\tau_2]\!]]\ a); \\ \mathsf{let}\ z = \mathsf{sel}_1\ y\ \mathsf{in} \\ \mathsf{returnRGN}\ [\mathbb{T}[\![\theta]\!]]\ [\mathbb{T}[\![\tau_1]\!]]\ z\end{array}$$

where x, y, z fresh

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} e : (\tau_1\times\tau_2,\rho),\theta \\ \Delta;\overline{\mathcal{S}}\vdash_{\text{rr}}\theta\succeq\rho\end{array}}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} \mathsf{snd}\ \ e : \tau_2,\theta}\right]\!\!\right] =$$

$$\begin{array}{l}\mathsf{bind}\ x{:}\mathbb{T}[\![(\tau_1\times\tau_2,\rho)]\!] \Leftarrow \mathbb{E}[\![e]\!]\,; \\ \mathsf{bind}\ y{:}\mathbb{T}[\![\tau_1\times\tau_2]\!] \Leftarrow \mathbb{E}[\![\theta\succeq\rho]\!]\ [\mathbb{T}[\![\tau_1\times\tau_2]\!]]\ (\mathsf{readRGNLoc}\ [\mathbb{T}[\![\rho]\!]]\ [\mathbb{T}[\![\tau_1\times\tau_2]\!]]\ a); \\ \mathsf{let}\ z = \mathsf{sel}_2\ y\ \mathsf{in} \\ \mathsf{returnRGN}\ [\mathbb{T}[\![\theta]\!]]\ [\mathbb{T}[\![\tau_2]\!]]\ z\end{array}$$

where x, y, z fresh

Fig. C 2. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Terms (IV))

Translations yielding closed values

Closed values

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}}\mathcal{S}:\overline{\mathcal{S}}}{\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{cval}}\mathsf{tt}:\mathsf{bool}}\right]\!\!\right] = \mathsf{tt}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}}\mathcal{S}:\overline{\mathcal{S}}}{\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{cval}}\mathsf{ff}:\mathsf{bool}}\right]\!\!\right] = \mathsf{ff}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}}\mathcal{S}:\overline{\mathcal{S}} \qquad r\in \mathit{dom}(\overline{\mathcal{S}}) \qquad l\in \mathit{dom}(\overline{\mathcal{S}}(r)) \qquad \mu=\mathcal{S}(r,l)}{\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{cval}}\langle l\rangle_r : (\mu,r)}\right]\!\!\right] = \langle l\rangle_{s\sharp r}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}}\mathcal{S}:\overline{\mathcal{S}} \qquad \cdot;\overline{\mathcal{S}}\vdash_{\text{btype}}\mu}{\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{cval}}\langle l\rangle_{\bullet} : (\mu,\bullet)}\right]\!\!\right] = \begin{cases}\langle l\rangle_{\circ\sharp\bullet} & \text{if } \overline{\mathcal{S}}=\cdot \\ \langle l\rangle_{s\sharp\bullet} & \text{otherwise}\end{cases}$$

Fig. C 3. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Closed values)

Translations yielding closed values

Storable values

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} i : \mathsf{int}}\right]\!\!\right] = i$$

$$\mathbb{E}\left[\!\!\left[\frac{\cdot; \cdot, x{:}\tau_1; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e' : \tau_2, \theta'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \lambda x{:}\tau_1.^{\theta'} e' : \tau_1 \xrightarrow{\theta'} \tau_2}\right]\!\!\right] = \lambda x{:}\mathbb{T}[\![\tau_1]\!].\mathbb{E}[\![e]\!]$$

$$\mathbb{E}\left[\!\!\left[\frac{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{cval}} v_1 : \tau_1 \qquad \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{cval}} v_2 : \tau_2}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} (v_1, v_2) : \tau_1 \times \tau_2}\right]\!\!\right] = (\mathbb{E}[\![v_1]\!], \mathbb{E}[\![v_2]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\cdot, \varrho \succeq \varphi; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} u' : \tau, \theta'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \lambda \varrho \succeq \varphi.^{\theta'} u' : \Pi \varrho \succeq \varphi.^{\theta'} \tau}\right]\!\!\right] =$$

$$\Lambda \varrho.\lambda w_\varrho{:}\mathbb{T}[\![\varrho \succeq \varphi]\!].\lambda h_\varrho{:}\mathsf{RGNHandle}\ \varrho.\mathbb{E}[\![u']\!]$$

Fig. C 4. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Storable values)