Monadic Regions *

Matthew Fluet
Cornell University
Department of Computer Science
Ithaca, NY 14853
fluet@cs.cornell.edu

Greg Morrisett
Harvard University
Division of Engineering and Applied Science
Cambridge, MA 02138
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, so 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 region-based language based on core Cyclone to a monadic variant of System F with region primitives whose types and operations are inspired by (and generalize) the ST monad.

Categories and Subject Descriptors: D.3.1 **[Programming Languages]**: Formal Definitions and Theory – *Semantics*; D.3.3 **[Programming Languages]**: Language Constructs and Features; F.3.2 **[Logics and Meanings of Programs]**: Semantics of Programming Languages.

General Terms: Languages, Theory.

Keywords: effect, monad, parametric polymorphism, region, region-based memory management, type system.

1 Background

Tofte and Talpin introduced a new technique for type-safe memory management based on *regions* [26, 27]. In their calculus, regions are areas of memory holding heap allocated data. Regions are introduced and eliminated with a lexically-scoped construct:

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

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:

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

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:

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

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.

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. Consider for example the following program:

$$\begin{array}{l}\text{letregion}\ \rho_a\ \text{in}\\ \quad \text{let g} = \text{letregion}\ \rho_b\ \text{in}\\ \qquad\qquad \text{let p} = (3\ \text{at}\ \rho_a, 4\ \text{at}\ \rho_b)\ \text{at}\ \rho_a\\ \qquad\qquad \text{in}\ \lambda z{:}\text{unit}.\,\texttt{fst}\ [\rho_a,\rho_b,\rho_a]\ \texttt{p}\\ \quad \text{in g}\ ()\end{array}$$

The pair p and its first component are allocated in the outer region ρ_a whereas p's second component is allocated in an inner region ρ_b. The closure bound to g is a thunk that calls `fst` on p. Note that the

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

Copyright 2004 ACM 1-58113-905-5/04/0009 ...$5.00

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 [25] used automatic region inference to translate SML into a region-based language instead of relying upon traditional garbage collection. In contrast, the Cyclone Safe-C language [8] 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.

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 [5, 2, 10, 3, 4, 11]. All of these problems are amplified in Cyclone because of region subtyping where the meta-theory is considerably more complicated [9].

1.1 Overview

The goal of this work is to find a simpler account of region-based type systems. In particular, our goal is to explain the type soundness of a Cyclone-like language via translation to a language with only parametric polymorphism, such as System F [20, 7].

Our work was inspired by the ST monad of Launchbury and Peyton Jones [15, 14] 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 . 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 [16] 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 core-Cyclone 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. In Section 4 we present 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 meaning-preserving fashion, thereby establishing our claim that parametric polymorphism is sufficient for encoding the type-and-effects systems of region calculi. In Section 6, we consider related work and in Section 7 we summarize and note some directions for future work. Due to space limitations, the proofs of the theorems presented here can be found in a companion technical report [6].

2 From ST to RGN

Launchbury and Peyton Jones [15, 14] 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:

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

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

The type $\mathsf{ST}\ s\ a$ is the type of computations which transform a store indexed by s and deliver a value of type a. The type $\mathsf{STRef}\ s\ a$ 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 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.

As an example, here is a function (in Haskell syntax, using the do notation, which implicitly invokes thenST) yielding a computation that pairs the contents of two references into a new reference:

```
pair :: ∀s.∀a.∀b.STRef s a → STRef s b →
                 ST s (STRef s (a,b))
pair 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, 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 RGNVar r a is the type of (immutable) variables allocated in a region indexed by r and containing a value of type a. Each of the operations in the ST monad has an analogue in the RGN monad:

$$
\begin{aligned}
\mathsf{returnRGN} &:: \forall r.\forall a.a \to \mathsf{RGN}\ r\ a \\
\mathsf{thenRGN} &:: \forall r.\forall a.\forall b.\mathsf{RGN}\ r\ a \to (a \to \mathsf{RGN}\ r\ b) \to \mathsf{RGN}\ r\ b \\
\mathsf{newRGNVar} &:: \forall r.\forall a.a \to \mathsf{RGN}\ r\ (\mathsf{RGNVar}\ r\ a) \\
\mathsf{readRGNVar} &:: \forall r.\forall a.\mathsf{RGNVar}\ r\ a \to \mathsf{RGN}\ r\ a \\
\mathsf{runRGN} &:: \forall a.(\forall r.\mathsf{RGN}\ r\ a) \to a
\end{aligned}
$$

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 variables in and read variables 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 and allocating output data from 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, the following term, where we think of the STRefs bound to a as an input, b as a temporary, and c as an output, does not type check:

```
runST (do a <- newSTRef 1
          c <- runST (do v <- readSTRef a
                         b <- newSTRef 2
                         newSTRef v)
          readSTRef c)
```

The error arises from the fact that the inner and the outer runST each require their argument to generalize over the store index type; this forces a to have the type STRef s_1 Int and readSTRef a to have the type ST s_2 Int, but this is incompatible with the type of readSTRef, which requires the store index types to be equal. A similar problem arises with newSTRef v and readSTRef c.

Launchbury and Sabry [16] argue that the principle behind runST can be generalized to provide nested scope. They introduce two additional operations

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

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[1] suggests introducing the constant

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

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.

We can rewrite our example using blockST and liftST:

```
runST (do a <- newSTRef 1
          c <- blockST (do v <- liftST (readSTRef a)
                           b <- newSTRef 2
                           liftST (newSTRef v))
          readSTRef c)
```

(Note that blockST and importSTRef are not sufficient; we would also need an exportSTRef operation.)

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. However, another critical aspect of region-based languages is region polymorphism. For example, consider a generalization of the pair function above, where each of the two input variables are allocated in different regions, the output variable 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:

```
gpair :: ∀r1.∀a.∀r2.∀b.∀r3.∀r4.
           RGNVar r1 a → RGNVar r2 b →
           RGN r4 (RGNVar r3 (a,b))
```

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

```
gpair v w =
  liftRGN (do a <- readRGNVar v
              b <- liftRGN (readRGNVar w)
              liftRGN (liftRGN (newRGNVar (a,b))) )
```

then we produce a function with the type:

```
gpair :: ∀r1.∀a.∀r2.∀b.∀r3.∀r4.
           RGNVar ((r1, r2), r3) a →
           RGNVar (r1, r2) b →
           RGN (((r1, r2), r3), r4) (RGNVar r1 (a, b))
```

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 variable is allocated in the outermost region, the input variables 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.

```
gpair :: ∀r1.∀a.∀r2.∀b.∀r3.∀r4.
           (∀c.RGN r1 c → RGN r4 c) →
           (∀c.RGN r2 c → RGN r4 c) →
           (∀c.RGN r3 c → RGN r4 c) →
           RGNVar r1 a → RGNVar r2 b →
           RGN r4 (RGNVar r3 (a,b))
gpair ev1 ev2 ev3 v w = do a <- ev1 (readRGNVar v)
                           b <- ev2 (readRGNVar w)
                           ev3 (newRGNVar (a,b))
```

While this evidence can be assembled from liftRGN terms, we find that the key notion is subtyping on regions and evidence that wit-

[1] private communication

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

Types

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

Terms

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

Values

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

RGN monad constants

$$
\begin{array}{rcl}
\kappa_r & ::= & \mathsf{runRGN} \\
\kappa & ::= & \mathsf{letRGN} \mid \mathsf{returnRGN} \mid \mathsf{thenRGN} \mid \\
 & & \mathsf{newRGNVar} \mid \mathsf{readRGNVar} \mid \mathsf{fixRGNVar}
\end{array}
$$

$$
\tau \preceq \tau' \equiv \forall \beta . \mathsf{RGN}\ \tau\ \beta \to \mathsf{RGN}\ \tau'\ \beta
$$

$$
\begin{array}{rcl}
\mathsf{runRGN} & :: & \forall \alpha . (\forall r . \mathsf{RGN}\ r\ \alpha) \to \alpha \\
\mathsf{letRGN} & :: & \forall r . \forall \alpha . (\forall s . r \preceq s \to \mathsf{RGN}\ s\ \alpha) \to \mathsf{RGN}\ r\ \alpha \\
\mathsf{returnRGN} & :: & \forall r . \forall \alpha . \alpha \to \mathsf{RGN}\ r\ \alpha \\
\mathsf{thenRGN} & :: & \forall r . \forall \alpha . \forall \beta . \mathsf{RGN}\ r\ \alpha \to (\alpha \to \mathsf{RGN}\ r\ \beta) \to \mathsf{RGN}\ r\ \beta \\
\mathsf{newRGNVar} & :: & \forall r . \forall \alpha . \alpha \to \mathsf{RGN}\ r\ (\mathsf{RGNVar}\ r\ \alpha) \\
\mathsf{readRGNVar} & :: & \forall r . \forall \alpha . \mathsf{RGNVar}\ r\ \alpha \to \mathsf{RGN}\ r\ \alpha \\
\mathsf{fixRGNVar} & :: & \forall r . \forall \alpha . (\mathsf{RGNVar}\ r\ \alpha \to \alpha) \to \mathsf{RGN}\ r\ (\mathsf{RGNVar}\ r\ \alpha)
\end{array}
$$

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

nesses 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:

$$
\begin{array}{rcl}
r \preceq s & \equiv & \forall b . \mathsf{RGN}\ r\ b \to \mathsf{RGN}\ s\ b \\
\mathsf{letRGN} & :: & \forall r . \forall a . (\forall s . r \preceq s \to \mathsf{RGN}\ s\ a) \to \mathsf{RGN}\ r\ a
\end{array}
$$

The argument to letRGN is given the evidence that the outer (older) region is a subtype of the new region, which it can use in the region computation. The same parametricity argument that applied to runST applies here: variables 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 $r \preceq s$ 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 RGNVar variables.

$$
\begin{array}{rcl}
l & \in & \mathit{Locs} \\
r & \in & \mathit{RNames} \\
s & \in & \mathit{SNames}
\end{array}
$$

Region placeholders

$$
\rho \ ::= \ r \mid \bullet
$$

Stack placeholders

$$
\sigma \ ::= \ s \mid \circ
$$

Types, terms, values, RGN monad constants

$$
\begin{array}{rcl}
\tau & ::= & \ldots \mid \sigma \sharp \rho \\
e & ::= & \ldots \mid \langle l \rangle_{\sigma \sharp \rho} \mid \underline{\kappa_r} \mid \overline{\kappa_r} \mid \underline{\kappa} \mid \overline{\kappa} \\
\kappa & ::= & \ldots \mid \mathsf{witnessRGN}\ \sigma \sharp \rho_1\ \sigma \sharp \rho_2 \\
\underline{\kappa_r}, \underline{\kappa} & ::= & \textit{partially applied constants} \\
\overline{\kappa_r}, \overline{\kappa} & ::= & \textit{fully applied constants} \\
v & ::= & \ldots \mid \langle l \rangle_{\sigma \sharp \rho} \mid \underline{\kappa_r} \mid \underline{\kappa} \mid \overline{\kappa}
\end{array}
$$

Regions

$$
R \ ::= \ \{ l_1 \mapsto v_1, \ldots, l_n \mapsto v_n \}
$$

Region stacks / Stacks

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

Figure 2. Operational semantics of $\mathsf{F}^{\mathsf{RGN}}$ (I)

3 The $\mathsf{F}^{\mathsf{RGN}}$ Calculus

The language $\mathsf{F}^{\mathsf{RGN}}$ is an extension of System F [20, 7] (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 [15, 14, 16, 1, 23, 19].

Figure 1 presents the syntax of "surface programs" in $\mathsf{F}^{\mathsf{RGN}}$ (that is, excluding intermediate terms that appear in the operational semantics). In the following sections, we expand upon the discussion in Section 2 to explain and motivate the design of $\mathsf{F}^{\mathsf{RGN}}$.

3.1 Types and terms

Types in $\mathsf{F}^{\mathsf{RGN}}$ include all those found in System F: the primitive types int and bool, function and product types, and type abstractions. The RGN τ_r τ_a and RGNVar τ_r τ_a types were introduced in the previous section.

Although the remainder of this paper will never require a region index to be represented by anything other than a type variable, we choose to allow an arbitrary type in the first argument of the RGN monad type constructor. We can thus interpret RGN as a primitive type constructor, without any special restrictions that may not be expressible in a practical programming language. Furthermore, in an operational semantics not based on type-erasure, type variables used as region indices will be instantiated with region types.

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 operations, given as constants with polymorphic types in a call-by-value interpretation of $\mathsf{F}^{\mathsf{RGN}}$. Each of the constants has been described earlier, except fixRGNVar, which provides a means of allocating recursive structures. We discuss this monadic operation in more detail below.

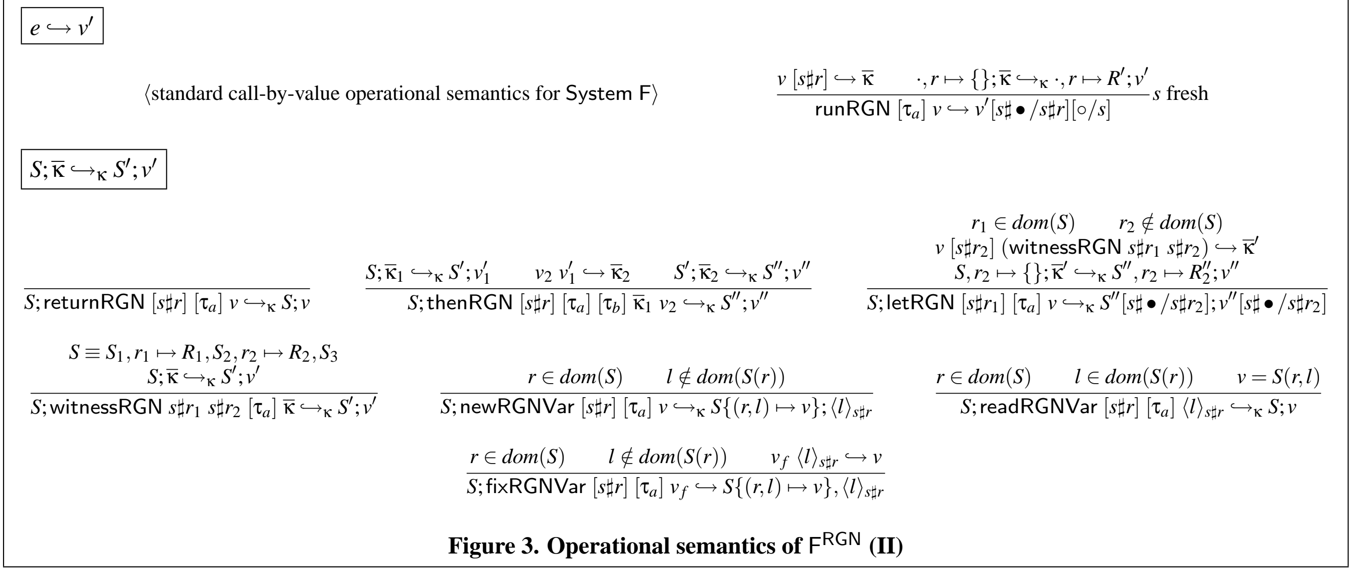

Figure 3. Operational semantics of $\mathsf{F}^{\mathsf{RGN}}$ (II)

3.2 Type system

We adopt the standard type system for System F; the only typing judgement of interest is $\Delta;\Gamma \vdash_{\text{exp}} e : \tau$ meaning that term e has type τ where Δ records the free type variables, and Γ records the free value variables and their types. The polymorphic types for each of the monadic operations are given in Figure 1, repeating those presented in Section 2.

Finally, we note that the (derived) typing rule for fixRGNVar $[\tau_r]$ $[\tau_a]$ f requires that the function f has the type RGNVar τ_r $\tau_a \to \tau_a$. This is a pure function, not a monadic computation. Hence, it is safe to use the variable where the allocated value is to be stored, because no computation (hence, no reading of region variables) can occur during the evaluation of the application of f to the variable. On the other hand, f can return a computation that reads the variable, since this computation cannot be run until after the knot has been tied.

3.3 Operational semantics and soundness

Figures 2 and 3 gives a large-step operational semantics in terms of a run-time stack of regions. Regions are mappings of locations to values, while stacks are (ordered) maps of region names to regions. Two mutually inductive judgements (one for pure expressions and one for monadic commands) define the semantics; the judgement for monadic commands takes a stack of regions and returns a modified stack and a final result. Deallocation of a region is modeled by replacing all occurrences of the region name with the dead region $\bullet$; the rule for readRGNVar does not apply to a location indexed by $\bullet$. The rule for runRGN chooses a fresh stack name s to distinguish computations encapsulated by this runRGN from computations encapsulated by other runRGNs.

While the large-step semantics 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* [28, 24], 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.

We can give a syntactic proof that the type system is sound with respect to the operational semantics:

THEOREM 1 ($\mathsf{F}^{\mathsf{RGN}}$ TYPE SOUNDNESS).
If $\cdot;\cdot \vdash_{\text{exp}} e : \tau$*, then any execution of e either terminates with a value v (such that* $\cdot;\cdot \vdash_{\text{exp}} v : \tau$*) or diverges.*

4 Region Calculi

Our goal is to show that $\mathsf{F}^{\mathsf{RGN}}$ is sufficient for encoding the region typing mechanisms of complicated languages such as Cyclone that include special purpose typing features for checking regions and effects. In our technical report, we have constructed a core model for Cyclone, called the Bounded Region Calculus (BRC), and have shown how it can be compiled in a meaning-preserving fashion to $\mathsf{F}^{\mathsf{RGN}}$. The translation is broken into two steps: we first translate BRC to a restricted form that is called the Single Effect Calculus (SEC). This translation encodes computational effects, which are sets of regions, into a single region coupled with a set of subtyping constraints. We then give a direct translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$.

Subtyping in both BRC and SEC derives from the fact that a LIFO stack of regions imposes a partial order on live (allocated) regions. Regions lower on the stack outlive regions higher on the stack. Hence, we consider a region to be a subtype of all the regions that it outlives.

In this section, we only briefly discuss the Bounded Region Calculus but present the Single Effect Calculus in detail. In the following section, we give the complete translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$, which is considerably more involved, and discuss the highlights of the proof of correctness.

4.1 The Bounded Region Calculus

The Bounded Region Calculus is adapted from the region calculus described by Henglein, Makholm, and Niss [11]. BRC is a direct-style, call-by-value language with a type-and-effects system similar to the original Tofte-Talpin region calculus. To keep things simple, the language does not include *effect polymorphism*, but this can be

simulated (see below). To accommodate Cyclone-like languages, BRC includes a form of bounded region subtyping. Thus, the type structure for BRC is as follows:

$$\begin{array}{llll} & \text{Region variables} & \rho & \in & \mathit{RVars}^{BRC} \\ & \text{Places} & \rho & ::= & \rho \mid \ldots \\ & \text{Effects} & \varphi & ::= & \{\rho_1,\ldots,\rho_n\} \\ \text{Types} & \tau & ::= & \mathsf{bool} \mid (\mu,\rho) \\ \text{Boxed types} & \mu & ::= & \mathsf{int} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid \tau_1 \times \tau_2 \mid \Pi\rho \succeq \varphi'.^{\varphi}\tau \end{array}$$

A region is associated with every type that requires heap allocated storage; we assume that integers, pairs, function closures, and region abstractions do so, while booleans do not. The type (μ,ρ) pairs together a boxed type (a type requiring heap allocated storage) and a place (a region); we interpret (μ,ρ) as the type of values of boxed type μ allocated in region ρ. At the source-level, places range over region variables (RVars^{BRC}), but in the dynamic semantics, we also have places corresponding to concrete region names and a distinguished constant representing a deallocated region.

Effects φ are finite sets of places. In the function and region-abstraction types, the effect φ denotes a *latent effect*: those regions read from or written to when a function is called. In a region-abstraction type $\Pi\rho \succeq \varphi'.^{\varphi}\tau$, the effect φ' serves as a lower bound on the lifetime of the region ρ. That is, 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.

A translation from BRC 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 type 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 by first sketching a translation to a simplified language, the Single Effect Calculus.

4.2 The Single Effect Calculus

The Single Effect Calculus is a restricted form of BRC where latent effects consist of a *single place* instead of a set of places. It is straightforward to translate from the full BRC language to SEC by leveraging the bounded subtyping. At the type level, this translation expands each type with a latent effect into a region abstraction bounded by that effect. For example, the translation of arrow types is as follows:

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

We have replaced the set of places φ with a single region variable ρ. However, we constrain ρ so that its liveness implies the liveness of all of the places in φ. Because the translation is so simple, we do not present the details of BRC here, though its full details, together with the translation to SEC are given in the report [6]. Instead, we concentrate on the formal definition of SEC and its translation to $\mathsf{F}^{\mathsf{RGN}}$.

Figure 4 presents the syntax of "surface programs" in the Single Effect Calculus (that is, excluding intermediate terms that appear in the operational semantics). Figure 5 present a type system for this external language. The technical report [6] gives a large-step operational semantics in terms of a run-time store; the semantics is completely standard for a region-based language. In the following sections, we explain and motivate the main constructs and typing rules of SEC.

$$\begin{array}{lll} i & \in & \mathbb{Z} \\ \vartheta,\rho & \in & \mathit{RVars}^{SEC} \quad \text{where } \mathcal{H} \in \mathit{RVars}^{SEC} \\ f,x & \in & \mathit{Vars}^{SEC} \\ \\ \text{Places} & & \\ \theta,\rho & ::= & \rho \\ \text{Effects} & & \\ \varphi & ::= & \{\rho_1,\ldots,\rho_n\} \\ \text{Types} & & \\ \tau & ::= & \mathsf{bool} \mid (\mu,\rho) \\ \text{Boxed types} & & \\ \mu & ::= & \mathsf{int} \mid \tau_1 \xrightarrow{\theta} \tau_2 \mid \tau_1 \times \tau_2 \mid \Pi\rho \succeq \varphi.^{\theta}\tau \\ \\ \text{Terms} & & \\ e & ::= & i \;\mathsf{at}\; \rho \mid e_1 \oplus e_2 \;\mathsf{at}\; \rho \mid e_1 \otimes e_2 \mid \\ & & \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\; e_b \;\mathsf{then}\; e_t \;\mathsf{else}\; e_f \mid \\ & & x \mid \lambda x:\tau.^{\theta}e \;\mathsf{at}\; \rho \mid e_1\; e_2 \mid \\ & & (e_1,e_2) \;\mathsf{at}\; \rho \mid \mathsf{fst}\; e \mid \mathsf{snd}\; e \mid \\ & & \mathsf{letregion}\; \rho \;\mathsf{in}\; e \mid \lambda\rho \succeq \varphi.^{\theta}u \;\mathsf{at}\; \rho \mid e\;[\rho] \\ & & \mathsf{fix}\; f:\tau.u \\ \text{Abstractions} & & \\ u & ::= & \lambda x:\tau.^{\theta}e \;\mathsf{at}\; \rho \mid \lambda\rho \succeq \varphi.^{\theta}u \;\mathsf{at}\; \rho \\ \text{Values} & & \\ v & ::= & \mathsf{tt} \mid \mathsf{ff} \mid x \end{array}$$

Figure 4. Surface syntax of SEC

4.2.1 *Types and terms*

The type structure of SEC is the same as that of BRC except that latent effects φ are restricted to a single place. As a convention, we will use θ to represent places that correspond to such effects.

As in the original region calculus, terms yielding heap allocated values have a region annotation at ρ, which indicates the region in which the value is to be allocated. New regions are introduced (and implicitly created and destroyed) by the letregion ρ in e term. The region variable ρ is bound within e, and values may be read from or allocated in the region ρ while evaluating e.

The term $\lambda\rho \succeq \varphi.^{\theta}u$ at ρ introduces a region abstraction (allocated in the region ρ), where the term u is polymorphic in the region ρ.[2] As explained previously, region abstractions make use of bounded quantification; the intention is that ρ is an upper bound on the set of regions φ. 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.

Finally, we include a fixed-point term, fix $f:\tau.u$. Since we intend SEC to obey a call-by-value evaluation semantics, we limit the body of a fixed-point to abstractions.

As an example, consider the following term to compute a factorial

[2]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.

Region contexts
$\Delta ::= \cdot \mid \Delta, \rho \succeq \varphi$

Value contexts
$\Gamma ::= \cdot \mid \Gamma, x : \tau$

$\boxed{\vdash_{\text{rctxt}} \Delta}$

$$\frac{}{\vdash_{\text{rctxt}} \cdot} \qquad \frac{\vdash_{\text{rctxt}} \Delta \quad \rho \notin dom(\Delta) \quad \Delta \vdash_{\text{eff}} \varphi}{\vdash_{\text{rctxt}} \Delta, \rho \succeq \varphi}$$

$\boxed{\Delta \vdash_{\text{place}} \rho}$

$$\frac{\vdash_{\text{rctxt}} \Delta \quad \rho \in dom(\Delta)}{\Delta \vdash_{\text{place}} \rho}$$

$\boxed{\Delta \vdash_{\text{eff}} \varphi}$

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

$\boxed{\Delta \vdash_{\text{btype}} \mu}$

$$\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{btype}} \mathsf{int}} \qquad \frac{\Delta \vdash_{\text{type}} \tau_1 \quad \Delta \vdash_{\text{place}} \theta \quad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{btype}} \tau_1 \xrightarrow{\theta} \tau_2} \qquad \frac{\Delta \vdash_{\text{type}} \tau_1 \quad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{btype}} \tau_1 \times \tau_2}$$

$$\frac{\Delta \vdash_{\text{eff}} \varphi \quad \Delta, \rho \succeq \varphi \vdash_{\text{place}} \theta \quad \Delta, \rho \succeq \varphi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{btype}} \Pi \rho \succeq \varphi .^{\theta} \tau}$$

$\boxed{\Delta \vdash_{\text{type}} \tau}$

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

$\boxed{\Delta \vdash_{\text{rr}} \rho \succeq \rho'}$

$$\frac{\vdash_{\text{rctxt}} \Delta \quad \Delta(\rho) = \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}}{\Delta \vdash_{\text{rr}} \rho \succeq \rho_i} \qquad \frac{\Delta \vdash_{\text{place}} \rho}{\Delta \vdash_{\text{rr}} \rho \succeq \rho} \qquad \frac{\Delta \vdash_{\text{rr}} \rho \succeq \rho' \quad \Delta \vdash_{\text{rr}} \rho' \succeq \rho''}{\Delta \vdash_{\text{rr}} \rho \succeq \rho''}$$

$\boxed{\Delta \vdash_{\text{re}} \rho \succeq \varphi}$

$$\frac{\vdash_{\text{rctxt}} \Delta \quad \Delta \vdash_{\text{rr}} \rho \succeq \rho_i \quad ^{i \in 1 \ldots n}}{\Delta \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

$\boxed{\Delta \vdash_{\text{vctxt}} \Gamma}$

$$\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{vctxt}} \cdot} \qquad \frac{\Delta \vdash_{\text{vctxt}} \Gamma \quad x \notin dom(\Gamma) \quad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{vctxt}} \Gamma, x : \tau}$$

$\boxed{\vdash_{\text{ctxt}} \Delta; \Gamma; \theta}$

$$\frac{\Delta \vdash_{\text{vctxt}} \Gamma \quad \Delta \vdash_{\text{place}} \theta}{\vdash_{\text{ctxt}} \Delta; \Gamma; \theta}$$

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

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \theta \quad \Delta \vdash_{\text{place}} \rho \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} i \ \mathsf{at}\ \rho : (\mathsf{int}, \rho), \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : (\mathsf{int}, \rho_1), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho_1 \quad \Delta; \Gamma \vdash_{\text{exp}} e_2 : (\mathsf{int}, \rho_2), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho_2 \quad \Delta \vdash_{\text{place}} \rho \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \oplus e_2 \ \mathsf{at}\ \rho : (\mathsf{int}, \rho), \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : (\mathsf{int}, \rho_1), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho_1 \quad \Delta; \Gamma \vdash_{\text{exp}} e_2 : (\mathsf{int}, \rho_2), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho_2}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \otimes e_2 : \mathsf{bool}, \theta}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \theta}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{tt} : \mathsf{bool}, \theta} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \theta}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{ff} : \mathsf{bool}, \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_b : \mathsf{bool}, \theta \quad \Delta; \Gamma \vdash_{\text{exp}} e_t : \tau, \theta \quad \Delta; \Gamma \vdash_{\text{exp}} e_f : \tau, \theta}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{if}\ e_b\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau, \theta}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \theta \quad x \in dom(\Gamma) \quad \Gamma(x) = \tau}{\Delta; \Gamma \vdash_{\text{exp}} x : \tau, \theta}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash_{\text{exp}} e' : \tau_2, \theta' \quad \Delta \vdash_{\text{place}} \rho \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \lambda x : \tau_1 .^{\theta'} e' \ \mathsf{at}\ \rho : (\tau_1 \xrightarrow{\theta'} \tau_2, \rho), \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : (\tau_1 \xrightarrow{\theta'} \tau_2, \rho_1'), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho_1' \quad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \tau_1, \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \theta'}{\Delta; \Gamma \vdash_{\text{exp}} e_1\ e_2 : \tau_2, \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : \tau_1, \theta \quad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \tau_2, \theta \quad \Delta \vdash_{\text{place}} \rho \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} (e_1, e_2) \ \mathsf{at}\ \rho : (\tau_1 \times \tau_2, \rho), \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \tau_2, \rho), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{fst}\ \ e : \tau_1, \theta} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \tau_2, \rho), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{snd}\ \ e : \tau_2, \theta}$$

$$\frac{\Delta \vdash_{\text{type}} \tau \quad \vdash_{\text{ctxt}} \Delta; \Gamma; \theta \quad \Delta, \rho \succeq \{\theta\}; \Gamma \vdash_{\text{exp}} e : \tau, \rho}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{letregion}\ \rho\ \mathsf{in}\ e : \tau, \theta}$$

$$\frac{\Delta, \rho \succeq \varphi; \Gamma \vdash_{\text{exp}} u' : \tau, \theta' \quad \Delta \vdash_{\text{place}} \rho \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \lambda \rho \succeq \varphi .^{\theta'} u' \ \mathsf{at}\ \rho : (\Pi \rho \succeq \varphi .^{\theta'} \tau, \rho), \theta}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : (\Pi \rho \succeq \varphi .^{\theta'} \tau, \rho_1'), \theta \quad \Delta \vdash_{\text{rr}} \theta \succeq \rho_1' \quad \Delta \vdash_{\text{place}} \rho_2 \quad \Delta \vdash_{\text{re}} \rho_2 \succeq \varphi \quad \Delta \vdash_{\text{rr}} \theta \succeq \theta'[\rho_2/\rho]}{\Delta; \Gamma \vdash_{\text{exp}} e_1\ [\rho_2] : \tau[\rho_2/\rho], \theta}$$

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

$\boxed{\vdash_{\text{prog}} e\ \mathsf{ok}}$

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

Figure 5. Static semantics of the Single Effect Calculus

(in which we elide the type annotation on *fact*):

$$
\begin{array}{l}
\mathsf{fix}\, fact. \\
\quad (\Pi\rho_i \succeq \{\}.^{\rho_f}(\Pi\rho_o \succeq \{\}.^{\rho_f}(\Pi\rho_b \succeq \{\rho_f,\rho_i,\rho_o\}.^{\rho_f} \\
\quad\; (\lambda n : (\mathsf{int},\rho_i).^{\rho_b} \\
\quad\quad \mathsf{if\ letregion}\ \rho\ \mathsf{in}\ n \le (1\ \mathsf{at}\ \rho) \\
\quad\quad\; \mathsf{then}\ 1\ \mathsf{at}\ \rho_o \\
\quad\quad\; \mathsf{else\ letregion}\ \rho_{i'}\ \mathsf{in}\ (\mathsf{letregion}\ \rho_{o'}\ \mathsf{in} \\
\quad\quad\quad\quad (fact\ [\rho_{i'}]\ [\rho_{o'}]\ [\rho_{o'}] \\
\quad\quad\quad\quad\quad\quad (\mathsf{letregion}\ \rho\ \mathsf{in}\ n - (1\ \mathsf{at}\ \rho)\ \mathsf{at}\ \rho_{i'}))) * n\ \mathsf{at}\ \rho_o \\
\quad)\ \mathsf{at}\ \rho_f)\ \mathsf{at}\ \rho_f)\ \mathsf{at}\ \rho_f)\ \mathsf{at}\ \rho_f
\end{array}
$$

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 $\rho_{o'}$ outlives ρ_f (through $\rho_{i'}$ and ρ_b), $\rho_{i'}$ is allocated before (and deallocated after) $\rho_{o'}$, and $\rho_{o'}$ clearly outlives itself.

4.2.2 Type system

The typing rules for SEC appear in Figure 5. Region contexts Δ are ordered lists of region variables bounded by effect sets. Value contexts Γ are ordered lists of variables and types. We summarize the main typing judgements in the following table:

Judgement	Meaning
$\Delta \vdash_{\text{btype}} \mu$	Boxed type μ is well-formed.
$\Delta \vdash_{\text{type}} \tau$	Type τ is well-formed.
$\Delta \vdash_{\text{rr}} \rho \succeq \rho'$	If region ρ is live, then region ρ' is live. (Alt.: region ρ' outlives region ρ.)
$\Delta \vdash_{\text{re}} \rho \succeq \varphi$	If region ρ is live, then all regions in φ are live. (Alt.: all regions in φ outlive region ρ.)
$\Delta;\Gamma \vdash_{\text{exp}} e : \tau,\theta$	Term e has type τ and effects bounded by region θ.
$\vdash_{\text{prog}} e\ \mathsf{ok}$	Program e is well-typed.

Previous formulations of region calculi (including BRC) make use of a judgement of the form $\Delta;\Gamma \vdash_{\text{exp}} e : \tau,\varphi$, where φ indicates the set of regions that may be affected by the evaluation of e. SEC simply replaces φ with a single region θ that bounds the effects in φ. 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).

We start by noting that the typing rules for the judgements $\Delta \vdash_{\text{rr}} \rho \succeq \rho'$ and $\Delta \vdash_{\text{re}} \rho \succeq \varphi$ 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.

The key judgement in region calculi is the typing rule for $\mathsf{letregion}\ \rho\ \mathsf{in}\ e$:

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta;\Gamma;\theta \qquad \Delta,\rho \succeq \{\theta\};\Gamma \vdash_{\text{exp}} e : \tau,\rho}{\Delta;\Gamma \vdash_{\text{exp}} \mathsf{letregion}\ \rho\ \mathsf{in}\ e : \tau,\theta}$$

Note that the (implicit) antecedent $\rho \notin dom(\Delta)$ and the (explicit) judgements $\Delta \vdash_{\text{type}} \tau$ and $\vdash_{\text{ctxt}} \Delta;\Gamma;\theta$ ensure that ρ does not appear in the result type or the types of the value environment. This new region is clearly related to the current region θ — it is outlived by the "old" current region and becomes the "new" current region for the evaluation of e. These facts are captured by the final antecedent $\Delta,\rho \succeq \{\theta\};\Gamma \vdash_{\text{exp}} e : \tau,\rho$.

It is worth comparing the treatment of latent effects in the Single Effect Calculus with their treatment in previous formulations of region calculi. In previous work, the typing rule for application appears as follows:

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

In SEC, the composite effect $\varphi \cup \varphi_1 \cup \varphi_2 \cup \{\rho\}$ is witnessed by a single effect 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 subexpressions e_1 and e_2. In order to execute the application, the operational semantics must read the function out of the region ρ; therefore, we require ρ to outlive the current region θ by the antecedent $\Delta \vdash_{\text{rr}} \theta \succeq \rho$. 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.

The typing rule for region application requires that we be able to show that the formal region parameter ρ is outlived by all of the regions in the region abstraction bound φ.

Finally, 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.

5 The Translation

In this section we present a type- and meaning-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. Many of the key components of the translation should be obvious, but we walk through the translation in stages, as there are some subtleties that require explanation.

We start with a few preliminaries. Technically, we assume injections from source variables $RVars^{SEC}$ and $Vars^{SEC}$ to target variables $TVars^{FRGN}$ and $Vars^{FRGN}$ respectively, but freely use variables from source objects in target objects. We further assume one additional injection from the set $RVars^{SEC}$ to the set $Vars^{FRGN}$, written w_ρ, that will denote the witnesses for the region ρ.

The translation is a typed call-by-value monad translation, similar to the standard translation given by Sabry and Wadler [22]. 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, and $\mathbb{E}[\![\cdot]\!]$ translates into expressions. 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.

Figure 6 shows the translation of types and contexts. As expected, the type (μ,ρ) is translated to the type $\mathsf{RGNVar}\ \rho\ \mathbb{T}[\![\mu]\!]$, whereby region allocated values in the source are also region allocated in the target. The interesting cases are function and region abstraction types. Functions with effects bounded by the region θ are trans-

Translations yielding types (from boxed types, types, and witnesses)

$$\begin{array}{rcl} \mathbb{T}[\![\mathsf{int}]\!] &=& \mathsf{int} \\ \mathbb{T}[\![\tau_1 \times \tau_2]\!] &=& \mathbb{T}[\![\tau_1]\!] \times \mathbb{T}[\![\tau_2]\!] \\ \mathbb{T}\left[\!\!\left[\tau_1 \xrightarrow{\theta} \tau_2\right]\!\!\right] &=& \mathbb{T}[\![\tau_1]\!] \to \mathsf{RGN}\ \theta\ \mathbb{T}[\![\tau_1]\!] \\ \mathbb{T}[\![\Pi\rho \succeq \varphi.^{\theta}\tau]\!] &=& \forall\rho.\mathbb{T}[\![\rho \succeq \varphi]\!] \to \mathsf{RGN}\ \theta\ \mathbb{T}[\![\tau]\!] \end{array} \qquad \begin{array}{rcl} \mathbb{T}[\![\mathsf{bool}]\!] &=& \mathsf{bool} \\ \mathbb{T}[\![(\mu,\rho)]\!] &=& \mathsf{RGNVar}\ \rho\ \mathbb{T}[\![\mu]\!] \end{array}$$

$$\begin{array}{rcl} \mathbb{T}[\![\rho \succeq \rho']\!] &=& \rho' \preceq \rho\ =\ \forall\beta.\mathsf{RGN}\ \rho'\ \beta \to \mathsf{RGN}\ \rho\ \beta \\ \mathbb{T}[\![\rho \succeq \{\rho_1,\ldots,\rho_n\}]\!] &=& \mathbb{T}[\![\rho \succeq \rho_1]\!] \times \cdots \times \mathbb{T}[\![\rho \succeq \rho_n]\!] \end{array}$$

Translations yielding type contexts (from region contexts)

$$\begin{array}{rcl} \mathbb{D}[\![\cdot]\!] &=& \cdot \\ \mathbb{D}[\![\Delta,\rho \succeq \varphi]\!] &=& \mathbb{D}[\![\Delta]\!],\rho \end{array}$$

Translations yielding value contexts (from region contexts and value contexts)

$$\begin{array}{rcl} \mathbb{G}[\![\cdot]\!] &=& \cdot \\ \mathbb{G}[\![\Delta,\rho \succeq \varphi]\!] &=& \mathbb{G}[\![\Delta]\!], w_\rho : \mathbb{T}[\![\rho \succeq \varphi]\!] \end{array} \qquad \begin{array}{rcl} \mathbb{G}[\![\cdot]\!] &=& \cdot \\ \mathbb{G}[\![\Gamma, x:\tau]\!] &=& \mathbb{G}[\![\Gamma]\!], x : \mathbb{T}[\![\tau]\!] \end{array}$$

LEMMA 1 (TRANSLATION PRESERVES TYPES (TYPES AND CONTEXTS)).

- *If* $\vdash_{\text{rctxt}} \Delta$, *then* $\vdash_{\text{rctxt}} \mathbb{D}[\![\Delta]\!]$.
- *If* $\Delta \vdash_{\text{btype}} \mu$, *then* $\mathbb{D}[\![\Delta]\!] \vdash_{\text{type}} \mathbb{T}[\![\mu]\!]$.
- *If* $\Delta \vdash_{\text{type}} \tau$, *then* $\mathbb{D}[\![\Delta]\!] \vdash_{\text{type}} \mathbb{T}[\![\tau]\!]$.
- *If* $\Delta \vdash_{\text{vctxt}} \Gamma$, *then* $\mathbb{D}[\![\Delta]\!] \vdash_{\text{vctxt}} \mathbb{G}[\![\Delta]\!], \mathbb{G}[\![\Gamma]\!]$.

Figure 6. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Types and Contexts)

lated into pure functions that yield computations in the RGN monad indexed by θ, whereas region abstractions are translated into type abstractions. The target calculus makes witness functions explicit, whereas in the source calculus such coercions are implied by $\succeq$ related regions. Hence, we translate $\rho \succeq \{\rho_1,\ldots,\rho_n\}$ to an n-tuple of functions, each witnessing a coercion from RGN ρ_i to RGN ρ.[3]

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 in value contexts, we have an additional translation from region contexts to value contexts. As explained above, witness functions are explicit values in the target calculus. Hence, our translation maintains the invariant that whenever a region variable $\rho \succeq \varphi$ is in scope in the source calculus, the variable w_ρ, of type $\mathbb{T}[\![\rho \succeq \varphi]\!]$, is in scope in the target calculus and holds the witness functions that coerce to region ρ.

Figure 7 shows the translation of witness terms. The first three translations map the reflexive, transitive closure of the syntactic constraints in a source Δ 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 region abstraction.

Figure 8 shows only the translation of key terms, as the cases for the other terms are straightforward. In order to make the translation easier to read, we introduce the following notation, reminiscent of Haskell's do notation:

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

where τ_r and τ_b are inferred from context.

The translation of an integer constant is a canonical example of allocation in the target calculus. The allocation is accomplished by the newRGNVar command, applied to the appropriate value. However, the resulting computation has type RGN ρ (RGNVar ρ 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 $\Delta \vdash_{\text{rr}} \theta \succeq \rho$. Allocation of a function closure proceeds in exactly the same manner. Function application, while notationally heavy, is simple. The bind sequences evaluating the function to a variable, reading the variable, 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 and w_ρ through Λ- and λ-abstractions. The coercion function is supplied by the letRGN command when the computation is executed.

The translation of region abstraction is similar to the translation of functions. Once again, witness functions are λ-bound in accordance to the invariants described above. During the translation of region applications, the appropriate tuple of witness functions (constructed by $\mathbb{E}[\![\Delta \vdash_{\text{re}} \rho_2 \succeq \varphi]\!]$) is supplied as an argument.

Figure 9 shows the translation of programs. The 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 coercions to the region $\mathcal{H}$.

5.1 Translation Properties

In each figure, we have indicated the particular type preservation lemma implied by each component of the translation. The proofs are by (mutual) induction on the structure of the typing judgements.

Furthermore, the translation is meaning preserving, with respect to the operational semantics of SEC and $\mathsf{F}^{\mathsf{RGN}}$. The essence of this proof relies on a *coherence* lemma stating that the translation of

[3] 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 expressions (from witness derivations)

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\mathrm{rctxt}} \Delta \quad \Delta(\rho) = \{\rho_1,\ldots,\rho_i,\ldots,\rho_n\}}{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho_i}\right]\!\!\right] = \Lambda\beta.\lambda k : \mathsf{RGN}\ \rho_i\ \beta.(\mathsf{sel}_i\ w_\rho)\ [\beta]\ k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\mathrm{place}} \rho}{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho}\right]\!\!\right] = \Lambda\beta.\lambda k : \mathsf{RGN}\ \rho\ \beta.k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho' \qquad \Delta \vdash_{\mathrm{rr}} \rho' \succeq \rho''}{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho''}\right]\!\!\right] = \Lambda\beta.\lambda k : \mathsf{RGN}\ \rho''\ \beta.\mathbb{E}[\![\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho']\!]\ [\beta]\ (\mathbb{E}[\![\Delta \vdash_{\mathrm{rr}} \rho' \succeq \rho'']\!]\ [\beta]\ k)$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\mathrm{rctxt}} \Delta \quad \overline{\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho_i}^{\,i\in 1\ldots n}}{\Delta \vdash_{\mathrm{re}} \rho \succeq \{\rho_1,\ldots,\rho_n\}}\right]\!\!\right] = (\mathbb{E}[\![\Delta \vdash_{\mathrm{re}} \rho \succeq \rho_1]\!],\ldots,\mathbb{E}[\![\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho_n]\!])$$

LEMMA 2 (TRANSLATION PRESERVES TYPES (WITNESSES)).

- *If* $\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho'$, *then* $\mathbb{D}[\![\Delta]\!];\mathbb{G}[\![\Delta]\!] \vdash_{\mathrm{exp}} \mathbb{E}[\![\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho']\!] : \mathbb{T}[\![\rho \succeq \rho']\!]$.
- *If* $\Delta \vdash_{\mathrm{rr}} \rho \succeq \varphi$, *then* $\mathbb{D}[\![\Delta]\!];\mathbb{G}[\![\Delta]\!] \vdash_{\mathrm{exp}} \mathbb{E}[\![\Delta \vdash_{\mathrm{re}} \rho \succeq \varphi]\!] : \mathbb{T}[\![\rho \succeq \varphi]\!]$.

Figure 7. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (witnesses)

witnesses yields functions that are operationally equivalent to the identify function:

LEMMA 5 (COHERENCE).
Suppose $\vdash_{\mathrm{stack}} S : \mathbb{S}$ *and* $\cdot;\mathbb{S} \vdash_{\mathrm{rr}} r \succeq r_i$.
Let $\mathbb{S}[\![\vdash_{\mathrm{stack}} S : \mathbb{S}]\!] = (S^*,\mathbb{S}^*)$ *and* $\mathbb{E}[\![\cdot;\mathbb{S} \vdash_{\mathrm{rr}} r \succeq r_i]\!] = e^*_w$.
If $\cdot;\cdot;\mathbb{S}^* \vdash_{\mathrm{exp}} \overline{\kappa}^* : \mathsf{RGN}\ s\sharp r_i\ \tau_a$ *and* $S^*;\overline{\kappa}^* \hookrightarrow_\kappa S'^*;v'^*$,
then $e^*_w\ [\tau_a]\ \overline{\kappa}^* \hookrightarrow \overline{\kappa}'$ *and* $S^*;\overline{\kappa}' \hookrightarrow_\kappa S'^*;v'^*$.

The judgement $\vdash_{\mathrm{stack}} S : \mathbb{S}$ asserts that S is well-formed with stack type $\mathbb{S}$; in the presence of allocated variables, the other typing judgements are augmented with the stack type to assign types to locations. The translation $\mathbb{S}[\![\cdot]\!]$ simply extends term and type translation to stacks and stack types. 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:

LEMMA 6.
Suppose $\vdash_{\mathrm{stack}} S : \mathbb{S}$, $\cdot;\cdot;\mathbb{S} \vdash_{\mathrm{exp}} e : \tau, r'$, *and* $S;e \hookrightarrow S';v'$,
and there exists $\mathbb{S}'$ *such that* $\vdash_{\mathrm{stack}} S' : \mathbb{S}'$ *and* $\mathbb{S}' \vdash_{\mathrm{cval}} v' : \tau$.
Let $\mathbb{S}[\![\vdash_{\mathrm{stack}} S : \mathbb{S}]\!] = (S^*,\mathbb{S}^*)$ *and* $\mathbb{E}[\![\cdot;\cdot;\mathbb{S} \vdash_{\mathrm{exp}} e : \tau, r']\!] = e^*$.
Then $e^* \hookrightarrow \overline{\kappa}^*$ *and* $S^*;\overline{\kappa}^* \hookrightarrow_\kappa S'^*;v'^*$, *where*
$\mathbb{S}[\![\vdash_{\mathrm{stack}} S' : \mathbb{S}']\!] = (S'^*,\mathbb{S}'^*)$ *and* $\mathbb{E}[\![\mathbb{S}' \vdash_{\mathrm{cval}} v' : \tau]\!] = v'^*$.

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 2 (TRANSLATION CORRECTNESS).
Suppose $\vdash_{\mathrm{prog}} e\ \mathsf{ok}$ *and* $e \hookrightarrow_{\mathrm{prog}} v'$.
Let $\mathbb{E}[\![\vdash_{\mathrm{prog}} e\ \mathsf{ok}]\!] = e^*$.
Then $e^* \hookrightarrow v'^*$, *where* $\mathbb{E}[\![\cdot \vdash_{\mathrm{cval}} v' : \mathsf{bool}]\!] = v'^*$.

Full details of this development are given in the report [6].

6 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 [26, 27]. Our Single Effect Calculus draws inspiration from the Capability Calculus [5] and Cyclone [9], where the "outlives" relationship between regions is recognized as an important component.

The work of Banerjee, Heintze, and Riecke [2] deserves special mention. They show how to translate the Tofte-Talpin region calculus into an extension of the polymorphic λ-calculus called $\mathsf{F}_\#$. 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 runtime significance in $\mathsf{F}_\#$ and there is no notion of deallocation upon exiting a region. The equality theory of types in $\mathsf{F}_\#$ is nontrivial, due to the treatment of #; in contrast, type equality on $\mathsf{F}^{\mathsf{RGN}}$ 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 $\mathsf{F}_\#$ lift and seq expressions and the monadic return and bind operations, although it is not mentioned or explored in the paper.

The second line of research on which we draw is the work done in monadic encapsulation of effects [17, 18, 21, 15, 29, 14, 16, 22, 1, 12, 23, 19, 30]. 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 [29, 19, 30] has considered more general combinations of effects and monads, no work has examined the combination of regions and monads.

Finally, we note that Wadler and Thiemann [30] advocate marrying effects and monads by translating a type $\tau_1 \xrightarrow{\sigma} \tau_2$ to the type $\mathbb{T}[\![\tau_1]\!] \to \mathsf{T}^\sigma\ \mathbb{T}[\![\tau_2]\!]$, 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 (encodings of) equivalent sets.

7 Conclusions and Future Work

We have given a type- and meaning-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. The translation provides a more

Translations yielding expressions (from expression derivations)

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{ctxt} \Delta;\Gamma;\theta \quad \Delta \vdash_{place} \rho \quad \Delta \vdash_{rr} \theta \succeq \rho}{\Delta;\Gamma \vdash_{exp} i \text{ at } \rho : (\mathsf{int},\rho),\theta}\right]\!\!\right] = \mathbb{E}[\![\Delta \vdash_{rr} \theta \succeq \rho]\!]\ (\mathsf{newRGNVar}\ [\rho]\ [\mathbb{T}[\![\mathsf{int}]\!]]\ i)$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{ctxt} \Delta;\Gamma;\theta \quad x \in dom(\Gamma) \quad \Gamma(x) = \tau}{\Delta;\Gamma \vdash_{exp} x : \tau,\theta}\right]\!\!\right] = \mathsf{returnRGN}\ [\theta]\ [\mathbb{T}[\![\tau]\!]]\ x$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma,x:\tau_1 \vdash_{exp} e' : \tau_2,\theta' \quad \Delta \vdash_{place} \rho \quad \Delta \vdash_{rr} \theta \succeq \rho}{\Delta;\Gamma \vdash_{exp} \lambda x:\tau_1.^{\theta'} e' \text{ at } \rho : (\tau_1 \xrightarrow{\theta'} \tau_2,\rho),\theta}\right]\!\!\right] =$$

$$\mathbb{E}[\![\Delta \vdash_{rr} \theta \succeq \rho]\!]\ [\mathbb{T}[\![(\tau_1 \xrightarrow{\theta'} \tau_2,\rho)]\!]]\ (\mathsf{newRGNVar}\ [\rho]\ [\mathbb{T}[\![\tau_1 \xrightarrow{\theta'} \tau_2]\!]]\ (\lambda x:\mathbb{T}[\![\tau_1]\!].\mathbb{E}[\![\Delta;\Gamma,x:\tau_1 \vdash_{exp} e:\tau_2,\theta']\!]))$$

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

$$\begin{array}{l}
\mathsf{bind}\ f : \mathbb{T}[\![(\tau_1 \xrightarrow{\theta'} \tau_2,\rho_1')]\!] \Leftarrow \mathbb{E}[\![\Delta;\Gamma \vdash_{exp} e_1 : (\tau_1 \xrightarrow{\theta'} \tau_2,\rho_1'),\theta]\!]\ ; \\
\mathsf{bind}\ g : \mathbb{T}[\![\tau_1 \xrightarrow{\theta'} \tau_2]\!] \Leftarrow \mathbb{E}[\![\Delta \vdash_{rr} \theta \succeq \rho_1']\!]\ [\mathbb{T}[\![\tau_1 \xrightarrow{\theta'} \tau_2]\!]]\ (\mathsf{readRGNVar}\ [\rho_1']\ [\mathbb{T}[\![\tau_1 \xrightarrow{\theta'} \tau_2]\!]]\ f); \\
\mathsf{bind}\ a : \mathbb{T}[\![\tau_1]\!] \Leftarrow \mathbb{E}[\![\Delta;\Gamma \vdash_{exp} e_2 : \tau_1,\theta]\!]\ ; \\
\mathsf{let}\ z = g\ a\ \mathsf{in} \\
\mathbb{E}[\![\Delta \vdash_{rr} \theta \succeq \theta']\!]\ [\mathbb{T}[\![\tau_2]\!]]\ z
\end{array}$$

where f,g,a,z fresh

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{type} \tau \quad \vdash_{ctxt} \Delta;\Gamma;\theta \quad \Delta,\rho \succeq \{\theta\};\Gamma \vdash_{exp} e : \tau,\rho}{\Delta;\Gamma \vdash_{exp} \mathsf{letregion}\ \rho\ \mathsf{in}\ e : \tau,\theta}\right]\!\!\right] = \mathsf{letRGN}\ [\theta]\ [\mathbb{T}[\![\tau]\!]]\ (\Lambda\rho.\lambda w_\rho : \mathbb{T}[\![\rho \succeq \{\theta\}]\!].\ \mathbb{E}[\![\Delta,\rho \succeq \{\theta\};\Gamma \vdash_{exp} e : \tau,\rho]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta,\rho \succeq \varphi;\Gamma \vdash_{exp} u' : \tau,\theta' \quad \Delta \vdash_{place} \rho \quad \Delta \vdash_{rr} \theta \succeq \rho}{\Delta;\Gamma \vdash_{exp} \lambda\rho \succeq \varphi.^{\theta'} u' \text{ at } \rho : (\Pi\rho \succeq \varphi.^{\theta'}\tau,\rho),\theta}\right]\!\!\right] =$$

$$\mathbb{E}[\![\Delta;\overline{S} \vdash_{rr} \theta \succeq \rho]\!]\ [\mathbb{T}[\![(\Pi\rho \succeq \varphi.^{\theta'}\tau,\rho)]\!]]\ (\mathsf{newRGNVar}\ [\rho]\ [\mathbb{T}[\![\Pi\rho \succeq \varphi.^{\theta'}\tau]\!]]\ (\Lambda\rho.\lambda w_\rho : \mathbb{T}[\![\rho \succeq \varphi]\!].\mathbb{E}[\![\Delta,\rho \succeq \varphi;\Gamma \vdash_{exp} u : \tau,\theta']\!]))$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{exp} e_1 : (\Pi\rho \succeq \varphi.^{\theta'}\tau,\rho_1'),\theta \quad \Delta \vdash_{rr} \theta \succeq \rho_1' \quad \Delta \vdash_{place} \rho_2 \quad \Delta \vdash_{re} \rho_2 \succeq \varphi \quad \Delta \vdash_{rr} \theta \succeq \theta'[\rho_2/\rho]}{\Delta;\Gamma \vdash_{exp} e_1\ [\rho_2] : \tau[\rho_2/\rho],\theta}\right]\!\!\right] =$$

$$\begin{array}{l}
\mathsf{bind}\ f : \mathbb{T}[\![(\Pi\rho \succeq \varphi.^{\theta'}\tau,\rho_1')]\!] \Leftarrow \mathbb{E}[\![\Delta;\Gamma \vdash_{exp} e : (\Pi\rho \succeq \varphi.^{\theta'}\tau,\rho_1'),\theta]\!]\ ; \\
\mathsf{bind}\ g : \mathbb{T}[\![\Pi\rho \succeq \varphi.^{\theta'}\tau]\!] \Leftarrow \mathbb{E}[\![\Delta \vdash_{rr} \theta \succeq \rho_1']\!]\ [\mathbb{T}[\![\Pi\rho \succeq \varphi.^{\theta'}\tau]\!]]\ (\mathsf{readRGNVar}\ [\rho_1']\ [\mathbb{T}[\![\Pi\rho \succeq \varphi.^{\theta'}\tau]\!]]\ f); \\
\mathsf{let}\ z = g\ [\rho_2]\ \mathbb{E}[\![\Delta \vdash_{re} \rho_2 \succeq \varphi]\!]\ \mathsf{in} \\
\mathbb{E}[\![\Delta \vdash_{rr} \theta \succeq \theta'[\rho_2/\rho]]\!]\ \mathbb{T}[\![\tau[\rho_2/\rho]]\!]\ z
\end{array}$$

where f,g,z fresh

LEMMA 3 (TRANSLATION PRESERVES TYPES (EXPRESSIONS)).

- *If* $\Delta;\Gamma \vdash_{exp} e : \tau,\theta$, *then* $\mathbb{D}[\![\Delta]\!];\mathbb{G}[\![\Delta]\!],\mathbb{G}[\![\Gamma]\!] \vdash_{exp} \mathbb{E}[\![\Delta;\Gamma \vdash_{exp} e : \tau,\theta]\!] : \mathsf{RGN}\ \theta\ \mathbb{T}[\![\tau]\!]$.

Figure 8. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Expressions)

Translations yielding terms (programs)

$$\mathbb{E}\left[\!\!\left[\frac{\cdot,\mathcal{H} \succeq \{\};\cdot \vdash_{exp} e : \mathsf{bool},\mathcal{H}}{\vdash_{prog} e\ \mathsf{ok}}\right]\!\!\right] = \mathsf{runRGN}\ [\mathbb{T}[\![\mathsf{bool}]\!]]\ (\Lambda\mathcal{H}.\mathsf{let}\ w_{\mathcal{H}} = ()\ \mathsf{in}\ \mathbb{E}[\![\cdot,\mathcal{H} \succeq \{\};\cdot \vdash_{exp} e : \mathsf{bool},\mathcal{H}]\!])$$

LEMMA 4 (TRANSLATION PRESERVES TYPES (FROM PROGRAM DERIVATIONS)).

- *If* $\vdash_{prog} e$ ok, *then* $\cdot;\cdot \vdash_{exp} \mathbb{E}[\![\cdot,\mathcal{H} \succeq \{\};\cdot \vdash_{exp} e : \mathsf{bool},\mathcal{H}]\!] : \mathbb{T}[\![\mathsf{bool}]\!]$.

Figure 9. Translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Programs)

modular account of the soundness of region-based languages such as Cyclone that incorporate region subtyping. The key insight is that traditional effects can be encoded using bounded region subtyping, and the subtyping can be eliminated by a coercion-based interpretation. Furthermore, the ideas behind the ST monad can be used to achieve encapsulation using only parametric polymorphism.

A simple extension [6] incorporates *region handles* – run-time values 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 the implementation of regions in languages like Cyclone. Region indices, like other types, have no run-time significance and may be erased from compiled code. In contrast, region handles are necessary at run-time to allocate values within a region.

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 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 [13] 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 witnesses.

8 References

[1] Z. Ariola and A. Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 62–74. ACM Press, 1998.

[2] A. Banerjee, N. Heintze, and J. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LCS'99)*, pages 88–97. IEEE Computer Society Press, 1999.

[3] C. Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 155–165. ACM Press, 2001.

[4] C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–332, Mar. 2002.

[5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 262–275. ACM Press, 1999.

[6] M. Fluet. Monadic regions: Formal type soundness and correctness. Technical Report TR2004-1936, Department of Computer Science, Cornell University, Apr. 2004.

[7] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[8] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-rased memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 282–293. ACM Press, 2002.

[9] D. Grossman, G. Morrisett, Y. Wang, T. Jim, M. Hicks, and J. Cheney. Formal type soundness for Cyclone's region system. Technical Report 2001-1856, Department of Computer Science, Cornell University, Nov. 2001.

[10] S. Helsen and P. Thiemann. Syntactic type soundness for the region calculus. In *Proceedings of the 4th International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'00)*, volume 41 of *Electronic Notes in Theoretical Computer Science*, pages 1–19. Elsevier Science Publishers, Sept. 2000.

[11] F. Henglein, H. Makholm, and H. Niss. Effect types and region-based memory management. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 5. MIT Press, 2004. In preparation.

[12] R. Kieburtz. Taming effects with monadic typing. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 51–62. ACM Press, 1998.

[13] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections, 5 June 2004. Submitted to the Haskell Workshop 2004.

[14] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 24–35. ACM Press, 1994.

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

[16] J. Launchbury and A. Sabry. Monadic state: Axiomatization and type safety. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 227–237. ACM Press, 1997.

[17] E. Moggi. Computational lambda calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LCS'89)*, pages 14–23, 1989.

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, Jan. 1991.

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

[20] J. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Apr. 1974.

[21] J. Riecke and R. Viswanathan. Isolating side effects in sequential languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 1–12. ACM Press, 1995.

[22] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, Nov. 1997.

[23] M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 8–17. ACM Press, 1999.

[24] G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1-3):49–72, 1998.

[25] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, , and P. Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, Apr. 2002.

[26] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 188–201. ACM Press, 1994.

[27] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.

[28] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CFSW'97)*, pages 156–168. IEEE Computer Society Press, 1997.

[29] P. Wadler. The marriage of effects and monads. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 63–74. ACM Press, 1995.

[30] P. Wadler and P. Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1):1–32, 2003.