Region analysis and the polymorphic lambda calculus

Anindya Banerjee*
Stevens Institute of Technology
ab@cs.stevens-tech.edu

Nevin Heintze
Bell Laboratories
nch@bell-labs.com

Jon G. Riecke
Bell Laboratories
riecke@bell-labs.com

Abstract

We show how to translate the region calculus of Tofte and Talpin, a typed lambda calculus that can statically delimit the lifetimes of objects, into an extension of the polymorphic lambda calculus called $F_\#$. We give a denotational semantics of $F_\#$, and use it to give a simple and abstract proof of the correctness of memory deallocation.

1 Introduction

Implementations of modern programming languages divide dynamically allocated memory into two parts. The *stack* is used for data that has a simple last-in, first-out lifetime determined by block structure; the other part (often called the *heap*) is used for data whose lifetime extends beyond the scope of program blocks. The heap is periodically "garbage collected" to reclaim memory that is no longer needed. Tofte and Talpin's *region calculus* [23] attempts to unify these two styles of memory management. The region calculus divides memory into regions, and provides a local scoping mechanism for those regions. Every value created by the program is placed in one of these regions. When program execution enters the scope of a region, memory for that region is allocated, and when program execution leaves the scope, memory for that region is deallocated. Hence memory is managed using a simple, block-structure allocation and deallocation mechanism.

This paper describes an abstract characterization of a version of the region calculus and gives a simple proof of its correctness. The proof relies on a translation into a version of the polymorphic lambda calculus called $F_\#$. Our proof is more abstract than the version presented by Tofte and Talpin [23]. Tofte and Talpin give a high-level operational semantics of a subset of ML [11], a translation of that language into their region calculus, and an operational semantics of the region calculus that explicitly manipulates memory. Their main theorem is a proof of adequacy: evaluation in either operational semantics yields the same results. The proof relies on greatest fixed points and coinduction, partly because of the coupling of the region calculus and the translation, and partly because deallocation can yield "dangling references". In contrast, we work at a higher level: we prove that once the program leaves the scope of a region, values held in that region do not affect the final results of the program. This fact implies that once the program leaves a block associated with a region, the memory for that region may be safely reclaimed.

The connections between the region calculus and parametric polymorphism can be brought out by a few examples. In the region calculus, each type is annotated with a region, and the type system tracks all reads and writes to regions. For example, the typing judgement

$$x : (\mathtt{int}, \rho) \vdash e : (\mathtt{int}, \rho), \{\mathtt{get}(\rho), \mathtt{put}(\rho)\}$$

means that there is a free variable of type `int` called x which lives in region ρ. The body e also produces an `int` in region ρ, and during the computation, values may be both read from and written into ρ. In general, a computation that accesses a region ρ has an associated "effect" $\mathtt{get}(\rho)$; conversely, any computation that stores into region ρ has an associated effect $\mathtt{put}(\rho)$. In the above judgement, e may arbitrarily read from and write to ρ. Contrast this with the typing judgement

$$x : (\mathtt{int}, \rho) \vdash e : (\mathtt{int}, \rho), \{\mathtt{get}(\rho)\}$$

In this case, e can access the value of x, but it may not store any values in region ρ. We can view this type as a restriction on the behavior of the function computed by the expression. In this case, if there is a value produced by the computation, it must be the value of x; the computation may diverge or converge to x depending on the value of x. The typing judgement

$$x : (\mathtt{int}, \rho) \vdash e : (\mathtt{int}, \rho), \emptyset$$

is even more restrictive: it specifies that there are no accesses to the region ρ. The only functions of this type are the identity function and the function that diverges on all

*Member of Church Project, and supported in part by NSF grant EIA-9806835.

parameters. This last example resembles the argument that the only functions one may write of type $(\forall\alpha.\ \alpha \rightarrow \alpha)$ are the identity and the everywhere divergent function.

It is perhaps not surprising that there should be some connection between parametric polymorphism and regions; Reynolds and others have made the same point about the connections between block structure in Idealized Algol and polymorphism [13, 17]. Others, too, have noted connections between garbage collection and polymorphism. Fradet, for instance, notes that if a subterm of a term can be typed with an abstract type α—and α does not appear in the type of the term—the subterm must be garbage [7]. That is, no matter what subterm is replaced for the original subterm, it will not affect the final results computed by the program. Hosoya and Yonezawa note a similar phenomenon [9].

The region calculus does, however, raise new semantic issues. One might, for instance, suggest that types such as $(\mathtt{int},\rho)$ be represented by type variables. The type $(\mathtt{int},\rho)$ is not, however, completely abstract: even if the program cannot access values of this type, these values must still be integers. Also, values of different types may be stored in the same region. For example, types $(\mathtt{int},\rho)$ and $(\mathtt{bool},\rho)$ can co-exist in the same program.

To model the behavior of the region calculus, our version of the polymorphic lambda calculus contains a new type operator #. One of the novel features of our calculus is that it combines a mechanism to hide the structure of a type (i.e., make the type abstract) with another mechanism to reveal the structure of such a hidden type. Intuitively, when a value is put into a region, its structure is hidden; when it is removed from the region, this structure is recovered. For example, consider the region type $(\mathtt{int},\rho)$. This is modelled in $F_{\#}$ by the type $(\mathtt{int}\ \#\ \rho)$, where ρ now denotes a type variable instead of a region variable. The presence of ρ in the type $(\mathtt{int}\ \#\ \rho)$ makes the type of `int` abstract. The language $F_{\#}$ also includes a special type `univ` which is the unit of the # operation; a type of the form $(\mathtt{int}\ \#\ \rho)$ can then be converted back to `int` by binding ρ to `univ`.

We formalize the connection via a type-directed translation from the region calculus to $F_{\#}$. Using the semantics of $F_{\#}$, we show that the translation is adequate, i.e., the answers produced by source and target programs are the same. Our main result is a proof of the safety of region deallocation: we show that if the effects associated with an expression do not mention a region ρ, then evaluation of the expression is independent of the values placed in ρ. Our proof is based on a relational-style semantics [12, 18]. While the domain theory underlying our proof is complex, the adaptation of this standard machinery for proving correctness of region deallocation is small and simple.

The benefits of our translation approach, as opposed to the operational techniques of Tofte and Talpin, are twofold. First, the translation brings out the essential details of the region calculus. In our version of the calculus, for instance, we simplify the rule for lambda abstraction, dropping a side condition from the original rules of Tofte and Talpin [23]. An additional benefit of our approach is that it can be more easily modified to account for extensions of the underlying language. Second, the proof of correctness is more modular. The main part of the proof is the type correctness of the translation. We believe, for instance, that other region-like calculi could be proven correct using an appropriate translation. We might also use the approach to prove computational properties of terms, similar to parametricity.

2 Region calculus

The region calculus of Tofte and Talpin is the target of a type-directed translation from an ML-like core language [23]. Instead of presenting the original translation, we omit the translation and describe the region calculus on its own. This simplification makes the typing rules easier to read, and also allows us to modify the type system so that the rules are suitable for type checking rather than type reconstruction.

The region calculus associates a region with every type. Let ρ range over region variables. The types and primitive effects are defined by the grammar

$$\begin{array}{rcl} s,t & ::= & (\mathtt{int},\rho) \mid (s \xrightarrow{\varphi} t,\rho) \\ \varepsilon & ::= & \mathtt{get}(\rho) \mid \mathtt{put}(\rho) \end{array}$$

where φ ranges over sets of primitive effects ε. Effects are either reads from a region ρ, denoted $\mathtt{get}(\rho)$, or writes to a region ρ, denoted $\mathtt{put}(\rho)$. The set above the function space denotes the set of *latent effects* of the function, and is a familiar feature of effect systems [10, 21]. Here, the latent effects of a function are the possible reads to or writes from regions after an argument has been supplied to the function.

The values and expressions of the region calculus are defined by the grammar

$$\begin{array}{rcl} v & ::= & (n,\rho) \mid ((\lambda x : s.\ e),\rho) \\ e & ::= & v \mid x \mid (e\ e) \mid (\mathtt{succ}[\rho']\ e) \mid (\mathtt{pred}[\rho']\ e) \mid \\ & & (\mathtt{if0}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e) \mid (\mathtt{letregion}\ \rho\ \mathtt{in}\ e) \mid \\ & & ((\mathtt{fun}\ f\ (x:s) = e),\rho) \end{array}$$

where x ranges over variables and n ranges over natural numbers. The annotation ρ on integer and function values denotes the region into which the value is placed; the annotation ρ' on `succ` and `pred` specifies the region into which the result will be placed. Most of the expressions are familiar from PCF [15, 19], with the exception of the `letregion` construct. This annotation demarcates the scope of a region ρ: within the region, values may be read from or written into ρ. Upon exit from the `letregion`, all values in the region become garbage and the memory may be reclaimed.

Table 1. Type system for the region calculus.

[*Var*]	$\Gamma \vdash x : \Gamma(x), \emptyset$	[*Const*]	$\Gamma \vdash (n,\rho) : (\mathtt{int},\rho), \{\mathtt{put}(\rho)\}$
[*Const'*]	$\dfrac{\Gamma \vdash e : (\mathtt{int},\rho),\varphi \qquad c = \mathtt{succ} \text{ or } \mathtt{pred}}{\Gamma \vdash (c[\rho']\ e) : (\mathtt{int},\rho'), \varphi \cup \{\mathtt{get}(\rho), \mathtt{put}(\rho')\}}$	[*If*]	$\dfrac{\Gamma \vdash e : (\mathtt{int},\rho),\varphi \qquad \Gamma \vdash e_i : t, \varphi_i}{\Gamma \vdash (\mathtt{if0}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) : t, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathtt{get}(\rho)\}}$
[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2, \varphi}{\Gamma \vdash ((\lambda x : s_1 .\ e), \rho) : (s_1 \xrightarrow{\varphi} s_2, \rho), \{\mathtt{put}(\rho)\}}$	[*App*]	$\dfrac{\Gamma \vdash e_1 : (s_1 \xrightarrow{\varphi} s_2, \rho), \varphi_1 \qquad \Gamma \vdash e_2 : s_1, \varphi_2}{\Gamma \vdash (e_1\ e_2) : s_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathtt{get}(\rho)\}}$
[*Region*]	$\dfrac{\Gamma \vdash e : s, \varphi \qquad \rho \notin FRV(\Gamma) \cup FRV(s)}{\Gamma \vdash (\mathtt{letregion}\ \rho\ \mathtt{in}\ e) : s, \varphi \setminus \{\mathtt{get}(\rho), \mathtt{put}(\rho)\}}$	[*Rec*]	$\dfrac{\Gamma, f : s, x : s_1 \vdash e : s_2, \varphi \qquad s = (s_1 \xrightarrow{\varphi} s_2, \rho)}{\Gamma \vdash ((\mathtt{fun}\ f\ (x : s_1) = e), \rho) : s, \{\mathtt{put}(\rho)\}}$

The typing rules for the region calculus appear in Table 1. Type contexts Γ are finite maps from variables to types, and judgements have the form $\Gamma \vdash e : s, \varphi$. We use $FRV(\Gamma)$ and $FRV(\varphi)$ to denote the free regions of a type context or set of effects.

The type rules differ from the original region calculus of Tofte and Talpin [23] in three ways. First, the original calculus has no constants or operations on the integers. We include them here to give some more computational content to the calculus. Second, we include no polymorphism on region variables or type variables, and the calculus is tuned for type checking rather than type inference. In particular, the rule for recursion is not as complicated, and there are no effect variables for use in unifying two latent effects. We will discuss region polymorphism later in Section 6. Third, the original typing rule for abstraction includes a side condition requiring that the body of the abstraction have no free region variables not occurring in the type context [23]. The original proof of correctness of the region calculus seems to require this side condition, but we do not.

The region calculus is a call-by-value language, whose operational semantics can be defined in the evaluation-context style of Felleisen [6]. Evaluation contexts are defined by the grammar

$$\begin{array}{lcl} E & ::= & [\cdot] \mid (E\ e) \mid (v\ E) \mid (\mathtt{succ}[\rho']\ E) \mid (\mathtt{pred}[\rho']\ E) \mid \\ & & (\mathtt{if0}\ E\ \mathtt{then}\ e\ \mathtt{else}\ e) \mid (\mathtt{letregion}\ \rho\ \mathtt{in}\ E) \end{array}$$

and the local operational rules are

$$\begin{array}{lcl} (((\lambda x : s.\ e), \rho)\ v) & \to & e[x := v] \\ (\mathtt{succ}[\rho']\ (n,\rho)) & \to & (n+1, \rho') \\ (\mathtt{pred}[\rho']\ (0,\rho)) & \to & (0, \rho') \\ (\mathtt{pred}[\rho']\ (n+1,\rho)) & \to & (n, \rho') \\ (\mathtt{if0}\ (n,\rho)\ \mathtt{then}\ e\ \mathtt{else}\ e') & \to & e, \qquad n = 0 \\ (\mathtt{if0}\ (n,\rho)\ \mathtt{then}\ e\ \mathtt{else}\ e') & \to & e', \qquad n \neq 0 \\ ((\mathtt{fun}\ f\ (x : s_1) = e), \rho) & \to & \\ \quad ((\lambda x : s_1 .\ e), \rho)[f := ((\mathtt{fun}\ f\ (x : s_1) = e), \rho)] & & \end{array}$$

We write $e \Downarrow v$ when e rewrites to v and v cannot be rewritten.

The original correctness theorem of the region calculus talks about an abstract implementation with memory, and shows that deallocation at the end of `letregion` is "safe": no data in the deallocated region can affect the final result. Our main result states the property more abstractly: if the effects associated with an expression do not mention the region ρ, then evaluation of the expression is independent of the values placed in ρ. This justifies the `letregion` rule, and therefore, region reuse. Specifically, we shall prove the following property:

> Suppose $x : (s,\rho) \vdash e : (\mathtt{int}, \rho_0), \varphi$ where $\rho \neq \rho_0$ and $\mathtt{get}(\rho) \notin \varphi$. If $\emptyset \vdash (v_i, \rho) : (s,\rho), \{\mathtt{put}(\rho)\}$ for $i = 1, 2$, then
>
> $$\begin{array}{l} (((\lambda x : (s,\rho).\ e), \rho_2)\ v_1) \Downarrow (n, \rho_0) \ \text{ iff} \\ (((\lambda x : (s,\rho).\ e), \rho_2)\ v_2) \Downarrow (n, \rho_0) \end{array}$$

The proof will rely upon the translation of the region calculus into $F_\#$. We postpone the proof until Section 5.

3 Syntax and semantics of $F_\#$

3.1 Overview

The target of the translation is $F_\#$, an extension of the polymorphic lambda calculus [8, 16] with base types, products, an associative, commutative, idempotent operation $\#$ for making a type abstract, and a special constant `univ` which is the unit of $\#$. For example, $(\mathtt{int}\ \#\ \mathtt{int})\ \#\ \mathtt{univ} = \mathtt{int}$.

Algebraically, $\#$ behaves like an intersection or union type operator. Unlike the case for union and intersection type operators, however, our calculus has no introduction or elimination rules for $\#$ and also no subtyping rules for $\#$, and that has a significant impact. For example, with intersections and unions, we can construct closed terms of type $(s \to s \vee t)$ and $(s \wedge t \to s)$. In $F_\#$, however, we cannot construct closed terms for $(s \to s\ \#\ t)$ or $(s\ \#\ t \to s)$. In fact, it is precisely this feature of $F_\#$ that allows us to model

Table 2. Typing rules for $F_\#$.

[*Var*]	$\Gamma \vdash x : \Gamma(x)$	[*Unit*]	$\Gamma \vdash \langle\rangle : \mathtt{unit}$	[*Int*]	$\Gamma \vdash n : \mathtt{int}$
[*Succ*]	$\dfrac{\Gamma \vdash e : \mathtt{int}}{\Gamma \vdash (\mathtt{succ}\, e) : \mathtt{int}}$	[*Pred*]	$\dfrac{\Gamma \vdash e : \mathtt{int}}{\Gamma \vdash (\mathtt{pred}\, e) : \mathtt{int}}$	[*If*]	$\dfrac{\Gamma \vdash e : \mathtt{int} \qquad \Gamma \vdash e_i : t}{\Gamma \vdash (\mathtt{if0}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2) : t}$
[*Pair*]	$\dfrac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$	[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e) : (s_1 \to s_2)}$	[*Lift*]	$\dfrac{\Gamma \vdash e : s}{\Gamma \vdash (\mathtt{lift}\, e) : s_\perp}$
[*Proj*]	$\dfrac{\Gamma \vdash e : (s_1 \times s_2)}{\Gamma \vdash (\mathtt{proj}_i\, e) : s_i}$	[*App*]	$\dfrac{\Gamma \vdash e : (s_1 \to s_2) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\, e') : s_2}$	[*Seq*]	$\dfrac{\Gamma \vdash e : s_\perp \quad \Gamma, x : s \vdash e' : t_\perp}{\Gamma \vdash \mathtt{seq}\, x = e\, \mathtt{in}\, e' : t_\perp}$
[*ForallI*]	$\dfrac{\Gamma \vdash e : s \qquad \alpha \notin FTV(\Gamma)}{\Gamma \vdash (\Lambda\alpha.\, e) : (\forall\alpha.\, s)}$	[*Rec*]	$\dfrac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s.\, e) : s}$	[*EqType*]	$\dfrac{\Gamma \vdash e : s \qquad \vdash (s = t)}{\Gamma \vdash e : t}$
[*ForallE*]	$\dfrac{\Gamma \vdash e : (\forall\alpha.\, s)}{\Gamma \vdash (e\, t) : s[\alpha := t]}$				

regions. A value of type t in a region ρ will have the $F_\#$ type $(t \# \rho)$ under the translation. The action of storing a value of type t in region ρ (denoted by the effect $\mathtt{put}(\rho)$) corresponds to a function of type $(t \to t \# \rho)$. Conversely, getting a value of type t out of a region ρ (denoted by the effect $\mathtt{get}(\rho)$) corresponds to a function of type $(t \# \rho \to t)$. So, in order to put a value of type t into region ρ, one must possess the "capability" function $(t \to t \# \rho)$. To get a value of type t from a region ρ, one must possess the "capability" function $(t \# \rho \to t)$.

The interpretation of $\#$ in the model of $F_\#$ also departs from the standard interpretation of union and intersection. The operation $\#$ is best viewed as a form of abstraction operation. For example, the type $(\mathtt{int} \# \rho)$ (corresponding to region type $(\mathtt{int}, \rho)$) is a hidden or opaque version of `int`. It retains the structure of `int`, but that structure is not visible. To recover this structure, we instantiate ρ with `univ`, since $(\mathtt{int} \# \mathtt{univ}) = \mathtt{int}$. The control of how and where ρ may be instantiated is critical, and is intimately tied to the translation of the `letregion` construct.

3.2 Syntax

The types of the target language are

$$s \quad ::= \quad \alpha \mid \mathtt{unit} \mid \mathtt{int} \mid (s \times s) \mid (s \to s) \mid (s_\perp) \mid (\forall\alpha.\, s) \mid \mathtt{univ} \mid (s \# s)$$

where α ranges over type variables. To make the connections with the region calculus more simple, we will also use ρ to range over type variables. Although $F_\#$ permits types $s \# t$ where s and t are arbitrary types, our translation from the region calculus does not employ these types in their full generality—e.g., we do not use types such as $\mathtt{int} \# (\mathtt{int} \to \mathtt{int})$.

The constructs `univ` and $\#$ have a nontrivial equality theory given by the following axiom schemes:

$$\begin{array}{rclcrcl} (s \# s) & = & s & \qquad & (s \# t) & = & (t \# s) \\ (s \# \mathtt{univ}) & = & s & & (s \# (t \# u)) & = & ((s \# t) \# u) \end{array}$$

In other words, $\#$ and `univ` satisfy the ACUI equations. We write $\vdash (s = t)$ when the equality can be derived from the equations above and the usual rules of reflexivity, symmetry, transitivity, and congruence.

The terms of $F_\#$ are those of the polymorphic lambda calculus over the extended types. More precisely, the terms are defined by the grammar

$$\begin{array}{rcl} e & ::= & x \mid \langle\rangle \mid (\mu f : s.\, e) \mid \\ & & n \mid (\mathtt{succ}\, e) \mid (\mathtt{pred}\, e) \mid (\mathtt{if0}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e) \mid \\ & & \langle e, e \rangle \mid (\mathtt{proj}_1\, e) \mid (\mathtt{proj}_2\, e) \mid (\lambda x : s.\, e) \mid (e\, e) \mid \\ & & (\mathtt{lift}\, e) \mid (\mathtt{seq}\, x = e\, \mathtt{in}\, e) \mid (\Lambda\alpha.\, e) \mid (e\, s) \end{array}$$

The typing rules for $F_\#$ appear in Table 2. Neither the `univ` type nor types of the form $(s \# t)$ have any special introduction or elimination operations associated with them. Of course, type variables also have no introduction or elimination operations. This fact gives some evidence that values of type $(s \# t)$ may not be created except by special operations.

3.3 Untyped denotational semantics

The semantics of $F_\#$ is given in two steps. First, we specify a meaning function on terms in an untyped model of the lambda calculus with constants. Second, we carve up the untyped model into subsets, and show how to construct relations between the subsets from certain primitive relations. The relations play a key role in the definition of the polymorphic types.

Table 3. Some clauses in the untyped denotational semantics of $F_\#$.

$$
\begin{array}{rcl}
[\![\Gamma \vdash (\lambda x:s.\,e):(s_1 \to s_2)]\!]\eta & = & inj_4(f), \qquad \text{where } f(d) = [\![\Gamma, x:s_1 \vdash e:s_2]\!]\eta[x \mapsto d] \\
[\![\Gamma \vdash (e\,e'):s_2]\!]\eta & = & [\![\Gamma \vdash e:s_1 \to s_2]\!]\eta \bullet [\![\Gamma \vdash e':s_1]\!]\eta \\
[\![\Gamma \vdash (\mathtt{lift}\,e):s_\perp]\!]\eta & = & inj_5([\![\Gamma \vdash e:s]\!]\eta, 0) \\
[\![\Gamma \vdash (\mathtt{seq}\,x = e\,\mathtt{in}\,e'):t_\perp]\!]\eta & = & \begin{cases} [\![\Gamma, x:s \vdash e':t_\perp]\!]\eta[x \mapsto d] & \text{if } [\![\Gamma \vdash e:s_\perp]\!]\eta = inj_5(d,0) \\ \perp & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash (\Lambda\alpha.\,e):(\forall\alpha.\,s)]\!]\eta & = & [\![\Gamma \vdash e:s]\!]\eta \\
[\![\Gamma \vdash (e\,t):s[\alpha := t]]\!]\eta & = & [\![\Gamma \vdash e:(\forall\alpha.\,s)]\!]\eta
\end{array}
$$

The untyped semantics is straightforward and familiar. Let $\mathbf{O} = \{\perp, \top\}$ be the Sierpinski space. Let U be the initial solution, in the category of dcpos and embedding-projection pairs, of the following domain equation [20]:

$$U = \mathbf{O} + \mathbf{N}_\perp + (U \times U) + (U \to U) + U_\perp$$

The U stands for "universe". Unlike certain partial equivalence relation models, we do not require any special properties of the solution for U (*cf.* [2]). Let $inj_1 : \mathbf{O} \to U$, $inj_2 : \mathbf{N}_\perp \to U$, $inj_3 : (U \times U) \to U$, $inj_4 : (U \to U) \to U$, and $inj_5 : U_\perp \to U$ be the injections corresponding to the domain equation above.

It is probably possible to construct a domain U out of terms rather than via domain theory, but then reasoning about recursion would become more difficult.

Typing judgements may be interpreted directly in this domain. Let η be a map from variables to elements of U. To simplify the notation, define the application operation on elements of U by

$$(d \bullet d') = \begin{cases} f(d') & \text{if } d = inj_4(f) \\ \perp & \text{otherwise} \end{cases}$$

Also, elements of $U_\perp$ are either $(d, 0)$ for $d \in U$ or $\perp$. The meaning of terms is determined by induction on the typing derivation. A few of the clauses appear in Table 3.

Theorem 3.1 *Suppose* $\Gamma \vdash e : s$ *is derivable. Then* $[\![\Gamma \vdash e : s]\!]\eta \in U$.

3.4 Typed denotational semantics

To give the typed semantics, we need only specify the meanings of types; the meaning of terms is the same as in the untyped model. The meaning of a type has two components: a set of elements and a means of relating two interpretations of a type. Define a *U-domain* to be a subset of U that contains $\perp$ and is directed-complete. More specifically, X is a U-domain if, whenever Y is a directed subset of X, the least upper bound—*as calculated in U*—is an element of X. Given a *type environment* ι—i.e., a map from type variables to U-domains—we can then calculate the meaning of a type as a U-domain.

The relational meaning of a type is more involved. Define a *U-relation* between U-domains A, B to be a subset $R \subseteq A \times B$ such that the following two properties hold:

- Pointedness: $(\perp, \perp) \in A$.
- Directed-completeness: Suppose $X = \{x_i \mid i \in I\} \subseteq A$ and $Y = \{y_i \mid i \in I\} \subseteq B$ are directed, and $(x_i, y_i) \in R$ for all $i \in I$. Then $(\bigsqcup X, \bigsqcup Y) \in R$.

We write the relation $R : A \leftrightarrow B$ when R is such a relation. A *relation environment* χ between type environments ι_1, ι_2, written $\chi : \iota_1 \leftrightarrow \iota_2$, is a map from type variables to U-relations such that for all α, $\chi(\alpha) : \iota_1(\alpha) \leftrightarrow \iota_2(\alpha)$.

The domain and relational meanings of types are defined by simultaneous induction on the structure of types. For type variables and the base types,

$$
\begin{array}{rcl}
[\![\alpha]\!]\iota & = & \iota(\alpha) \\
\mathcal{R}[\![\alpha]\!]\chi & = & \chi(\alpha) \\
[\![\mathtt{unit}]\!]\iota & = & \{inj_1(x) \mid x \in \mathbf{O}\} \\
\mathcal{R}[\![\mathtt{unit}]\!]\chi & = & \{(inj_1(x), inj_1(x)) \mid x \in \mathbf{O}\} \\
[\![\mathtt{int}]\!]\iota & = & \{inj_2(x) \mid x \in \mathbf{N}_\perp\} \\
\mathcal{R}[\![\mathtt{int}]\!]\chi & = & \{(inj_2(x), inj_2(x)) \mid x \in \mathbf{N}_\perp\} \\
[\![\mathtt{univ}]\!]\iota & = & U \\
\mathcal{R}[\![\mathtt{univ}]\!]\chi & = & \{(\perp, \perp)\}
\end{array}
$$

For product types,

$$
\begin{array}{rcl}
[\![s_1 \times s_2]\!]\iota & = & \{inj_3\langle x_1, x_2\rangle \mid x_i \in [\![s_i]\!]\iota\} \\
\mathcal{R}[\![s_1 \times s_2]\!]\chi & = & (inj_3\langle x_1, x_2\rangle, inj_3\langle y_1, y_2\rangle) \mid \\
& & (x_i, y_i) \in \mathcal{R}[\![s_i]\!]\chi\}
\end{array}
$$

For function types,

$$
\begin{array}{rcl}
[\![s \to t]\!]\iota & = & \{inj_4(f) \mid \text{for all } x \in [\![s]\!]\iota,\, f(x) \in [\![t]\!]\iota\} \\
\mathcal{R}[\![s \to t]\!]\chi & = & \{(inj_4(f), inj_4(g)) \mid \\
& & \forall (x,y) \in \mathcal{R}[\![s]\!]\chi,\, (f(x), g(y)) \in \mathcal{R}[\![t]\!]\chi\}
\end{array}
$$

For lifted types,

$$\begin{array}{rcl} [\![s_\perp]\!]\iota & = & \{inj_5(x) \mid x \in ([\![s]\!]\iota)_\perp\} \\ \mathcal{R}[\![s_\perp]\!]\chi & = & \{(inj_5(x,0), inj_5(y,0)) \mid (x,y) \in \mathcal{R}[\![s]\!]\chi\} \cup \\ & & \{(\perp,\perp)\} \end{array}$$

For polymorphic types, define the relation environment $\Delta_\iota : \iota \leftrightarrow \iota$ by

$$\Delta_\iota(\alpha) = \{(x,x) \mid x \in \iota(\alpha)\}$$

and define

$$\begin{array}{rcl} [\![\forall\alpha.\, s]\!]\iota & = & \{x \in \bigcap_A [\![s]\!]\iota[\alpha \mapsto A] \mid \\ & & \quad \forall R : A_1 \leftrightarrow A_2,\, (x,x) \in \mathcal{R}[\![s]\!]\Delta_\iota[\alpha \mapsto R]\} \\ \mathcal{R}[\![\forall\alpha.\, s]\!]\chi & = & \{(x,y) \in \bigcap_{R:A_1 \leftrightarrow A_2} \mathcal{R}[\![s]\!]\chi[\alpha \mapsto R]\} \end{array}$$

Finally, for the $\#$ operation, if $\chi : \iota_1 \leftrightarrow \iota_2$, define

$$\begin{array}{rcl} [\![s \# t]\!]\iota & = & [\![s]\!]\iota \cap [\![t]\!]\iota \\ \mathcal{R}[\![s \# t]\!]\chi & = & \{(x,y) \in \mathcal{R}[\![s]\!]\chi \cup \mathcal{R}[\![t]\!]\chi \mid \\ & & \quad x \in [\![s]\!]\iota_1 \cap [\![t]\!]\iota_1 \text{ and } y \in [\![s]\!]\iota_2 \cap [\![t]\!]\iota_2\} \end{array}$$

Proposition 3.2 *Suppose s is a type.*

- *If ι is a type environment, then $[\![s]\!]\iota$ is a U-domain.*
- *If $\chi : \iota_1 \leftrightarrow \iota_2$ is a relation environment, then $\mathcal{R}[\![s]\!]\chi : [\![s]\!]\iota_1 \leftrightarrow [\![s]\!]\iota_2$ is a U-relation.*

Proof: The only real difficulty lies in showing that $\#$ works properly on relations, i.e., $\mathcal{R}[\![s \# t]\!]\chi$ is pointed and directed-complete. Note first that $(\perp,\perp) \in \mathcal{R}[\![s \# t]\!]\chi$. Next, suppose X is a directed subset of $\mathcal{R}[\![s \# t]\!]\chi$. Let $X_A = X \upharpoonright \mathcal{R}[\![s]\!]\chi$ and $X_B = X \upharpoonright \mathcal{R}[\![t]\!]\chi$ (these sets might overlap). There are three cases.

In the first case, for any tuple $(x,x') \in X_A$, there is a tuple $(y,y') \in X_B$ such that $x \sqsubseteq y$ and $x' \sqsubseteq y'$. We claim that X_B is directed. To see why, suppose $(y_1,y'_1),(y_2,y'_2) \in X_B$. Then there must be a tuple $(z,z') \in X$ such that $y_i \sqsubseteq z$ and $y'_i \sqsubseteq z'$. Now, if that tuple (z,z') is in X_B, we are done. If the tuple is in X_A, on the other hand, we know that we can pick some $(y,y') \in X_B$ such that $z \sqsubseteq y$ and $z' \sqsubseteq y'$. This tuple (y,y') is also an upper bound for the (y_i,y'_i)'s.

Let $Y = \{y \mid (y,y') \in X_B\}$ and $Y' = \{y' \mid (y,y') \in X_B\}$. Both of these sets are directed because X_B is. Thus,

$$(z,z') = \bigsqcup X_B = (\bigsqcup Y, \bigsqcup Y')$$

exists. Note that $z \in [\![s]\!]\iota_1 \cap [\![t]\!]\iota_1$ and similarly for z'. Also, because of the directed completeness of $\mathcal{R}[\![t]\!]\chi$, $(z,z') \in \mathcal{R}[\![s]\!]\chi \cup \mathcal{R}[\![t]\!]\chi$. Finally, (z,z') is the least upper bound of X_B and, by the hypothesis, is an upper bound for all of the X_A; thus, it is the least upper bound of X, and so we are done.

In the second case, for any tuple $(y,y') \in X_B$, there is a tuple $(x,x') \in X_A$ such that $y \sqsubseteq x$ and $y' \sqsubseteq x'$. This case is similar to the previous case.

Finally, suppose neither of the two cases above hold. That is, there is a tuple $(x_1,x_2) \in X_A$ such that no $(y,y') \in X_B$ is above it, and there is a tuple $(y_1,y_2) \in X_B$ such that no $(x,x') \in X_A$ is above it. Since X is directed, there must be a tuple (z_1,z_2) such that $x_i, y_i \sqsubseteq z_i$. Now, either (z_1,z_2) is in X_A or X_B. But both cases are ruled out by the hypothesis. Thus, this case holds vacuously. This completes the case analysis and hence the proof. ■

One may also prove that the equality theory on types is satisfied by the interpretation:

Proposition 3.3 *Suppose s,t,u are types. Then the equations above hold, e.g.,*

$$\begin{array}{rcl} [\![s \# s]\!]\iota & = & [\![s]\!]\iota \\ \mathcal{R}[\![s \# s]\!]\chi & = & \mathcal{R}[\![s]\!]\chi \end{array}$$

Finally, we can show that the meaning of any typing judgement is in the appropriate type. Suppose η are environments, Γ is a type environment, and ι is a map from type variables to U-domains. We say η is *compatible with* Γ,ι if for all $x \in \Gamma$, $\eta(x) \in [\![\Gamma(x)]\!]\iota$. Then

Theorem 3.4 *Suppose $\Gamma \vdash e : s$ is derivable and η is compatible with Γ,ι. Then $[\![\Gamma \vdash e : s]\!]\eta \in [\![s]\!]\iota$.*

The proof requires additional induction hypotheses, one of which amounts to Reynolds's Abstraction Theorem [18].

4 Translation of region calculus into $F_\#$

We are now ready to connect the region calculus with $F_\#$. We use a type-directed translation: types in the region calculus are represented by types in $F_\#$, and typing derivations in the region calculus are translated into typing derivations in $F_\#$. We then show how to use the semantic model to prove the correctness property stated in Section 2 for the region calculus.

We begin with the translation of types, which involves two functions. Define the translation $(\cdot)^*$ from types with regions to $F_\#$-types, and $(\cdot)^\dagger$, from types without regions to $F_\#$-types, as follows:

$$\begin{array}{rcl} (s,\rho)^* & = & (s^\dagger \# \rho) \\ \mathtt{int}^\dagger & = & \mathtt{int} \\ (s \xrightarrow{\varphi} t)^\dagger & = & (s^* \to \varphi^* \to (t^*)_\perp) \end{array}$$

These types are related to the interpretation of call-by-value in call-by-name (see [14]). The translation also involves the translation of sets of effects to $F_\#$-types. Define

$$\begin{array}{rcl} (\mathtt{get}(\rho))^* & = & (\forall\beta.\, (\beta \# \rho) \to \beta) \\ (\mathtt{put}(\rho))^* & = & (\forall\beta.\, \beta \to (\beta \# \rho)) \end{array}$$

Table 4. Translation of the region calculus.

[*Var*]	$\Gamma \vdash x : \Gamma(x), \emptyset$	$\Rightarrow$	$\Gamma^* \vdash (\lambda p : \emptyset^*.\ \mathtt{lift}\ x) : \emptyset^* \to (\Gamma(x))^*_\bot$
[*Const*]	$\Gamma \vdash (n,\rho) : (\mathtt{int},\rho), \{\mathtt{put}(\rho)\}$	$\Rightarrow$	$\Gamma^* \vdash (\lambda p : \{\mathtt{put}(\rho)\}^*.\ \mathtt{lift}\ (p\ \mathtt{int}\ n)) : \{\mathtt{put}(\rho)\}^* \to (\mathtt{int},\rho)^*_\bot$
[*Const'*]	$\dfrac{\begin{array}{c}\Gamma \vdash e : (\mathtt{int},\rho),\varphi \qquad c = \mathtt{succ}\text{ or }\mathtt{pred}\\ \varphi_1 = \varphi \cup \{\mathtt{get}(\rho),\mathtt{put}(\rho')\}\end{array}}{\Gamma \vdash (c[\rho']\ e) : (\mathtt{int},\rho'),\varphi_1}$	$\Rightarrow$	$\dfrac{\Gamma^* \vdash e^* : \varphi^* \to (\mathtt{int},\rho)^*_\bot \qquad x,b,b' \text{ fresh}}{\begin{array}{l}\Gamma^* \vdash (\lambda x : \varphi_1^*.\ \mathtt{seq}\, b = (e^*\ (\pi[\varphi_1,\varphi]\ x))\ \mathtt{in}\\ \qquad \mathtt{let}\ b' = (\pi[\varphi_1,\{\mathtt{get}(\rho)\}]\ x\ \mathtt{int}^\dagger\ b)\ \mathtt{in}\\ \qquad \mathtt{lift}\ (\pi[\varphi_1,\{\mathtt{put}(\rho')\}]\ x\ (c\ b'))) : \varphi_1^* \to (\mathtt{int},\rho')^*_\bot\end{array}}$
[*If*]	$\dfrac{\begin{array}{c}\Gamma \vdash e : (\mathtt{int},\rho),\varphi \qquad \Gamma \vdash e_i : t,\varphi_i\\ \varphi_3 = \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathtt{get}(\rho)\}\end{array}}{\Gamma \vdash (\mathtt{if0}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) : t,\varphi_3}$	$\Rightarrow$	$\dfrac{\Gamma^* \vdash e^* : \varphi^* \to (\mathtt{int},\rho)^*_\bot \qquad \Gamma^* \vdash e_i^* : \varphi_i^* \to t_\bot^* \qquad x,b,b' \text{ fresh}}{\begin{array}{l}\Gamma^* \vdash (\lambda x : \varphi_3^*.\ \mathtt{seq}\, b = (e^*\ (\pi[\varphi_3,\varphi]\ x))\ \mathtt{in}\\ \qquad \mathtt{let}\ b' = (\pi[\varphi_3,\{\mathtt{get}(\rho)\}]\ x\ \mathtt{int}^\dagger\ b)\ \mathtt{in}\\ \qquad \mathtt{if0}\ b'\ \mathtt{then}\ e_1^*\ (\pi[\varphi_3,\varphi_1]\ x)\ \mathtt{else}\ e_2^*\ (\pi[\varphi_3,\varphi_2]\ x)) : \varphi_3^* \to t_\bot^*\end{array}}$
[*Lam*]	$\dfrac{\Gamma, x : s_1 \vdash e : s_2,\varphi \qquad s = (s_1 \xrightarrow{\varphi} s_2,\rho)}{\Gamma \vdash ((\lambda x : s_1.\ e),\rho) : s,\{\mathtt{put}(\rho)\}}$	$\Rightarrow$	$\dfrac{\Gamma^*, x : s_1^* \vdash e^* : \varphi^* \to (s_2^*)_\bot \qquad p \text{ fresh}}{\Gamma^* \vdash (\lambda p : \{\mathtt{put}(\rho)\}^*.\ \mathtt{lift}\ (p\ (s_1 \xrightarrow{\varphi} s_2)^\dagger\ (\lambda x : s_1^*.\ e^*))) : \{\mathtt{put}(\rho)\}^* \to s_\bot^*}$
[*App*]	$\dfrac{\begin{array}{c}\Gamma \vdash e_1 : (s_1 \xrightarrow{\varphi} s_2,\rho),\varphi_1 \qquad \Gamma \vdash e_2 : s_1,\varphi_2\\ \varphi_3 = \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathtt{get}(\rho)\}\end{array}}{\Gamma \vdash (e_1\ e_2) : s_2,\varphi_3}$	$\Rightarrow$	$\dfrac{\Gamma^* \vdash e_1^* : \varphi_1^* \to (s_1 \xrightarrow{\varphi} s_2,\rho)^*_\bot \qquad \Gamma^* \vdash e_2^* : \varphi_2^* \to (s_1^*)_\bot \qquad x,f,g,v \text{ fresh}}{\begin{array}{l}\Gamma^* \vdash (\lambda x : \varphi_3^*.\ \mathtt{seq}\ f = (e_1^*\ (\pi[\varphi_3,\varphi_1]\ x))\ \mathtt{in}\\ \qquad \mathtt{let}\ g = (\pi[\varphi_3,\{\mathtt{get}(\rho)\}]\ x\ (s_1 \xrightarrow{\varphi} s_2)^\dagger\ f)\ \mathtt{in}\\ \qquad \mathtt{seq}\ v = e_2^*\ (\pi[\varphi_3,\varphi_2]\ x)\ \mathtt{in}\ (g\ v\ (\pi[\varphi_3,\varphi]\ x))) : \varphi_3^* \to (s_2^*)_\bot\end{array}}$
[*Region*]	$\dfrac{\begin{array}{c}\Gamma \vdash e : s,\varphi \qquad \rho \notin FRV(\Gamma) \cup FRV(s)\\ \varphi_1 = \varphi \setminus \{\mathtt{get}(\rho),\mathtt{put}(\rho)\}\end{array}}{\Gamma \vdash (\mathtt{letregion}\ \rho\ \mathtt{in}\ e) : s,\varphi_1}$	$\Rightarrow$	$\dfrac{\Gamma^* \vdash e^* : \varphi^* \to s_\bot^* \qquad x \text{ fresh}}{\Gamma^* \vdash (\lambda x : \varphi_1^*.\ ((\Lambda\rho.\ e^*)\ \mathtt{univ}\ (add[\varphi_1,\varphi]x))) : \varphi_1^* \to s_\bot^*}$
[*Rec*]	$\dfrac{\Gamma, f : s, x : s_1 \vdash e : s_2,\varphi \qquad s = (s_1 \xrightarrow{\varphi} s_2,\rho)}{\Gamma \vdash ((\mathtt{fun}\ f\ (x : s_1) = e),\rho) : s,\{\mathtt{put}(\rho)\}}$	$\Rightarrow$	$\dfrac{\Gamma^*, f : s^*, x : s_1^* \vdash e^* : \varphi^* \to (s_2)^*_\bot \qquad p \text{ fresh}}{\Gamma^* \vdash (\lambda p : \{\mathtt{put}(\rho)\}^*.\ \mathtt{lift}\ (\mu f : s^*.\ p\ (s_1 \xrightarrow{\varphi} s_2)^\dagger\ (\lambda x : s_1^*.\ e^*))) : \{\mathtt{put}(\rho)\}^* \to s_\bot^*}$

To be precise about the translation of sets of effects, assume some canonical ordering of effects. Suppose that the set $\{\varepsilon_1,\ldots,\varepsilon_n\}$ is ordered by this ordering. Define

$$\{\varepsilon_1,\ldots,\varepsilon_n\}^* \quad = \quad \begin{cases} \mathtt{unit} & \text{if } n = 0 \\ (\varepsilon_1)^* \times \ldots \times (\varepsilon_n)^* & \text{otherwise} \end{cases}$$

Note that the ordering ensures that two different means of writing the same effect set yield the same type.

The translation of typing derivations appears in Table 4. To simplify the translation, we use "let" notation (which can be desugared into abstraction and application), I for $(\Lambda\beta.\ \lambda x : \beta.\ x)$, and the notation $\pi[\varphi_1,\varphi_2] : \varphi_1^* \to \varphi_2^*$, where $\varphi_2 \subseteq \varphi_1$, for the canonical projection of the first set onto the second set. We also use the notation $add[\varphi_2,\varphi_1] : \varphi_2^* \to \varphi_1^*$ for a function that takes a tuple in φ_2^* and puts in I for the missing components to produce a tuple in φ_1^*.

Proposition 4.1 *Suppose $\Gamma \vdash e : s,\varphi$ is derivable, and the derivation translates to $\Gamma^* \vdash e^* : \varphi^* \to s_\bot^*$. Then the translation of the judgement is a legal typing derivation in $F_\#$.*

Theorem 4.2 (Adequacy) *Suppose $\emptyset \vdash e : (\mathtt{int},\rho),\varphi$ is derivable, and the derivation translates to*

$$\emptyset \vdash e^* : \varphi^* \to (\mathtt{int}\ \#\ \rho)_\bot$$

where $\vec{\rho}$ are the free region variables. Then $e \Downarrow (n,\rho)$ iff $[\![(\Lambda\vec{\rho}.\ e^*)\ \vec{\mathtt{univ}}\ \langle I,\ldots,I\rangle]\!] = [\![\mathtt{lift}\ (n)]\!]$.

To see how the translation works, we consider a small example. Let $\varphi_1 = \{\mathtt{put}(\rho_1)\}$, $\varphi_2 = \{\mathtt{put}(\rho_2)\}$, $\varphi_3 = \{\mathtt{get}(\rho_1)\}$, and $\varphi = \{\mathtt{put}(\rho_1),\mathtt{put}(\rho_2),\mathtt{get}(\rho_1)\}$. Also, let $s = ((\mathtt{int},\rho_2) \xrightarrow{\emptyset} (\mathtt{int},\rho_2),\rho_1)$. Consider the following closed expressions given with their types and effects.

$$\begin{array}{rcl} e_0 & = & (\mathtt{letregion}\ \rho_1\ \mathtt{in}\ e_1) : (\mathtt{int},\rho_2),\{\mathtt{put}(\rho_2)\} \\ e_1 & = & (e_2\ e_3) : (\mathtt{int},\rho_2),\varphi \\ e_2 & = & ((\lambda x : (\mathtt{int},\rho_2).\ x),\rho_1) : s,\{\mathtt{put}(\rho_1)\} \\ e_3 & = & (2,\rho_2) : (\mathtt{int},\rho_2),\{\mathtt{put}(\rho_2)\} \end{array}$$

Then the translation of e_3 is $(\lambda p : \varphi_2^*.\ \mathtt{lift}\ (p\ \mathtt{int}\ 2))$ which has type $\varphi_2^* \to t_\bot$ with $t = (\mathtt{int}\ \#\ \rho_2)$. To translate e_2, note

$$s^\dagger = ((\mathtt{int}\ \#\ \rho_2) \to \mathtt{unit} \to (\mathtt{int}\ \#\ \rho_2)_\bot).$$

Then $s^* = (s^\dagger\ \#\ \rho_1)$, and the translation for e_2 is

$$(\lambda p : \varphi_1^*.\ \mathtt{lift}\ (p\ s^\dagger\ (\lambda x : t.\ \lambda p : \mathtt{unit}.\ \mathtt{lift}\ (x))))$$

which has type $\varphi_1^* \to s_\bot^*$. The term e_1 gets translated as

$$\begin{array}{l}(\lambda z : \varphi^*.\ \mathtt{seq}\, f = e_2^*\ (\pi[\varphi,\varphi_1]\ z)\ \mathtt{in} \\ \qquad \mathtt{let}\, g = (\pi[\varphi,\varphi_3]\ z\ s^\dagger\ f)\ \mathtt{in} \\ \qquad \mathtt{seq}\, v = e_3^*\ (\pi[\varphi,\varphi_2]\ z)\ \mathtt{in} \\ \qquad (g\ v\ (\pi[\varphi,\{\}]\ z)))\end{array}$$

with type $\varphi^* \to t_\bot$. Finally, e_0's judgement is translated into

$$(\lambda u : \varphi_2^*.\ ((\Lambda\rho_1.\ e_1^*)\ \mathtt{univ}\ \langle I,u,I\rangle))$$

with type $\varphi_2^* \to t_\perp$.

This example also serves to illustrate the adequacy theorem. The judgement for e_0^* can be simplified to

$$\emptyset \vdash (\lambda u : \varphi_2^*.\, \texttt{lift}\ (u\ \texttt{int}\ 2)) : \varphi_2^* \to t_\perp.$$

Note that $e_0 \Downarrow (2, \rho_2)$ iff $[\![(\Lambda\rho_2.\, e_0^*)\texttt{univ}\langle I\rangle]\!] = [\![\texttt{lift}\ (2)]\!]$.

5 Applications

The # operation of $F_\#$ makes a type abstract (i.e. hides its content) in a manner that is similar to quantification over type variables in the polymorphic lambda calculus. In direct analogy to the property that functions of type $(\forall\alpha.\, \alpha \to \alpha)$ are either the identity function or the everywhere-$\perp$ function, we have the following property for functions of type $\forall\alpha.\, (\texttt{int}\,\#\,\alpha) \to (\texttt{int}\,\#\,\alpha)$:

Proposition 5.1 *If $f \in [\![\forall\alpha.\, (\texttt{int}\,\#\,\alpha) \to (\texttt{int}\,\#\,\alpha)]\!]\iota$, then*

- *For any $x \in [\![\texttt{int}]\!]\iota$, $(f \bullet x) = \perp$; or*
- *For any $x \in [\![\texttt{int}]\!]\iota$, $(f \bullet x) = x$.*

Proof: Consider any $x, y \in [\![\texttt{int}]\!]\iota$. Define the U-domains $A_1 = \{\perp, x\}$ and $A_2 = \{\perp, y\}$, and set $R : A_1 \leftrightarrow A_2$ by $R = \{(\perp,\perp), (x,y)\}$. Then we know that

$$(f \bullet x, f \bullet y) \in \mathcal{R}[\![\texttt{int}\,\#\,\alpha]\!]\Delta_\iota[\alpha \mapsto R]$$

by the definition of $\forall$. This means that for all $x, y \in [\![\texttt{int}]\!]\iota$, either $(f \bullet x) = \perp$ or $(f \bullet y) = y$. In other words,

- For any $x \in [\![\texttt{int}]\!]\iota$, $(f \bullet x) = \perp$, or
- For any $y \in [\![\texttt{int}]\!]\iota$, $(f \bullet y) = y$.

■

For another example of how the model can be used to establish properties, recall the judgement

$$x : (\texttt{int}, \rho) \vdash e : (\texttt{int}, \rho), \{\texttt{get}(\rho)\}$$

from Section 1, which is translated into $F_\#$ as:

$$x : (\texttt{int}\,\#\,\rho) \vdash e^* : (\forall\beta.\, (\beta\,\#\,\rho) \to \beta) \to (\texttt{int}\,\#\,\rho)_\perp$$

The following proposition shows that any function of this type must either return its argument or diverge.

Proposition 5.2 *Suppose*

$$f \in [\![\forall\alpha.\, (\texttt{int}\,\#\,\alpha) \to (\forall\beta.\, (\beta\,\#\,\alpha) \to \beta) \to (\texttt{int}\,\#\,\alpha)_\perp]\!]\iota.$$

For any U-domain A, any $x \in [\![\texttt{int}\,\#\,\alpha]\!]\iota[\alpha \mapsto A]$, and any $g \in [\![\forall\beta.\, (\beta\,\#\,\alpha) \to \beta]\!]\iota[\alpha \mapsto A]$, then either $(f \bullet x \bullet g) = x$ or $(f \bullet x \bullet g) = \perp$.

The next proposition and theorem use the model to establish the correctness of region deallocation.

Proposition 5.3 *If $f \in [\![\forall\alpha.\, (s\,\#\,\alpha) \to \texttt{int}_\perp]\!]\iota$, A is a U-domain, and $x, y \in [\![s\,\#\,\alpha]\!]\iota[\alpha \mapsto A]$, then $(f \bullet x) = (f \bullet y)$.*

Proof: Consider the relation $R : A \leftrightarrow A$ where $R = A \times A$. We know that

$$(f, f) \in \mathcal{R}[\![(s\,\#\,\alpha) \to \texttt{int}_\perp]\!]\Delta_\iota[\alpha \mapsto R]$$

Pick any elements $x, y \in [\![s\,\#\,\alpha]\!]\iota[\alpha \mapsto A]$. Then $(x, y) \in \mathcal{R}[\![s\,\#\,\alpha]\!]\Delta_\iota[\alpha \mapsto R]$, so

$$(f \bullet x, f \bullet y) \in \mathcal{R}[\![\texttt{int}_\perp]\!]\Delta_\iota[\alpha \mapsto R].$$

But then $(f \bullet x) = (f \bullet y)$ as desired. ■

Theorem 5.4 *Suppose $x : (s, \rho) \vdash e : (\texttt{int}, \rho_0), \varphi$ where $\rho \neq \rho_0$ and $\texttt{get}(\rho) \notin \varphi$. If $\emptyset \vdash (v_i, \rho) : (s, \rho), \{\texttt{put}(\rho)\}$ for $i = 1, 2$, then*

$$\begin{array}{l} (((\lambda x : (s,\rho).\, e), \rho_2)\ v_1) \Downarrow (n, \rho_0)\ \textit{iff} \\ (((\lambda x : (s,\rho).\, e), \rho_2)\ v_2) \Downarrow (n, \rho_0) \end{array}$$

Proof: We use the model of $F_\#$ in a crucial way to carry out the proof. Let $\vec{\rho}$ be the region variables other than ρ.

$$\begin{array}{rcl} t & = & ((s,\rho) \xrightarrow{\varphi} (\texttt{int}, \rho_0)) \\ \varphi_1 & = & \varphi \cup \{\texttt{put}(\rho), \texttt{get}(\rho_2), \texttt{put}(\rho_2)\} \\ e' & = & ((\lambda x : (s,\rho).\, e), \rho_2)^* \\ e_i & = & (\Lambda\vec{\rho}.\, \Lambda\rho.\, \lambda z : \varphi_1^*. \\ & & \texttt{seq}\, f = e'\ (\pi[\varphi_1, \{\texttt{put}(\rho_2)\}]\ z)\ \texttt{in} \\ & & \texttt{let}\, g = (\pi[\varphi_1, \{\texttt{get}(\rho_2)\}]\ z\ t^\dagger\ f)\ \texttt{in} \\ & & \texttt{seq}\, v = v_i^*\ (\pi[\varphi_1, \{\texttt{put}(\rho)\}]\ z)\ \texttt{in} \\ & & g\ v\ (\pi[\varphi_1, \varphi]\ z))\ \vec{\texttt{univ}}\ \texttt{univ}\ \langle I, \ldots, I\rangle \\ u & = & (s,\rho)^*[\vec{\rho} := \vec{\texttt{univ}}] \\ F & = & \Lambda\rho.\, \lambda v : u.\, \texttt{seq}\, f = e'[\vec{\rho} := \vec{\texttt{univ}}]\ I\ \texttt{in} \\ & & \quad f\ v\ \langle I, \ldots, I\rangle \end{array}$$

Note first that $u = (u'\,\#\,\rho)$ for some u'. Also, a simple inspection of the translation of values shows that

$$v_i^*[\vec{\rho} := \vec{\texttt{univ}}] = \lambda p : (\forall\beta.\, \beta \to \beta\,\#\,\rho).\, \texttt{lift}\ (v_i')$$

for some terms v_i'. Let

$$v_i'' = \lambda p : (\forall\beta.\, \beta \to \beta\,\#\,\rho).\, v_i'$$

Note that F has type $(\forall\rho.\, (u'\,\#\,\rho) \to \texttt{int}_\perp)$. Now

$$\begin{array}{ll} \multicolumn{2}{l}{(((\lambda x : (s,\rho).\, e), \rho_2)\ v_1) \Downarrow (n, \rho_0)} \\ \text{iff} & [\![e_1]\!]\iota = [\![\texttt{lift}\ (n)]\!]\iota \\ \text{iff} & [\![F\ \texttt{univ}\ (v_1''[\rho := \texttt{univ}]\ I)]\!]\iota = [\![\texttt{lift}\ (n)]\!]\iota \\ \text{iff} & [\![F\ \texttt{univ}\ (v_2''[\rho := \texttt{univ}]\ I)]\!]\iota = [\![\texttt{lift}\ (n)]\!]\iota \\ \text{iff} & [\![e_2]\!]\iota = [\![\texttt{lift}\ (n)]\!]\iota \\ \text{iff} & (((\lambda x : (s,\rho).\, e), \rho_2)\ v_2) \Downarrow (n, \rho_0) \end{array}$$

Table 5. Polymorphism and its translation.

$$[\mathit{Copy}]\quad \frac{\Gamma \vdash e : (t,\rho),\varphi \qquad \varphi_0 = \varphi \cup \{\mathtt{get}(\rho),\mathtt{put}(\rho')\}}{\Gamma \vdash (\mathtt{copy}[\rho']\; e) : (t,\rho'),\varphi_0} \;\Rightarrow\; \frac{\Gamma^* \vdash e^* : \varphi^* \to (t^\dagger \,\#\, \rho)_\perp \qquad x \text{ fresh}}{\begin{array}{l}\Gamma^* \vdash (\lambda x : \varphi_0^*.\ \mathtt{seq}\, f = (e^*\ (\pi[\varphi_0,\varphi]\ x))\ \mathtt{in} \\ \qquad \mathtt{lift}\ (\pi[\varphi_0,\{\mathtt{put}(\rho')\}]\ x\, t^\dagger\ (\pi[\varphi_0,\{\mathtt{get}(\rho)\}]\ x\, t^\dagger\ f))) : \varphi_0^* \to (t,\rho')_\perp^*\end{array}}$$

$$[\mathit{PolyFun}]\quad \frac{\Gamma, f : s, x : s_1 \vdash e : s_2,\varphi \qquad s = (\forall\vec{\rho}.\ s_1 \xrightarrow{\varphi} s_2,\rho')}{\Gamma \vdash ((\mathtt{fun}\, f\,(x : s_1) = e),\rho') : s, \{\mathtt{put}(\rho')\}} \;\Rightarrow\; \frac{\Gamma^*, f : s^*, x : s_1^* \vdash e^* : \varphi^* \to (s_2)_\perp^* \qquad t = (\forall\vec{\rho}.\ s_1 \xrightarrow{\varphi} s_2)^\dagger \qquad p \text{ fresh}}{\Gamma^* \vdash (\lambda p : \{\mathtt{put}(\rho')\}^*.\ \mathtt{lift}\ (\mu f : s^*.\ p\, t\ (\Lambda\vec{\rho}.\ \lambda x : s_1^*.\ e^*))) : \{\mathtt{put}(\rho)\}^* \to s_\perp^*}$$

$$[\mathit{PolyApp}]\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : (\forall\vec{\rho}.\ (s_1 \xrightarrow{\varphi} s_2),\rho'),\varphi_1 \\ \Gamma \vdash e_2 : s_1,\varphi_2 \qquad \varphi_3 = \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathtt{get}(\rho')\}\end{array}}{\Gamma \vdash (e_1\ [\vec{\rho}]\ e_2) : s_2,\varphi_3} \;\Rightarrow\; \frac{\Gamma^* \vdash e_1^* : \varphi_1^* \to (\forall\vec{\rho}.\ (s_1 \xrightarrow{\varphi} s_2),\rho')_\perp^* \qquad \Gamma^* \vdash e_2^* : \varphi_2^* \to (s_1)_\perp^* \qquad x \text{ fresh}}{\begin{array}{l}\Gamma^* \vdash (\lambda x : \varphi_3^*.\ \mathtt{seq}\, f = e_1^*\ (\pi[\varphi_3,\varphi_1]\ x)\ \mathtt{in} \\ \qquad \mathtt{seq}\, v = e_2^*\ (\pi[\varphi_3,\varphi_2]\ x)\ \mathtt{in} \\ \qquad \mathtt{let}\, w = ((\pi[\varphi_3,\{\mathtt{get}(\rho')\}]\ x)\ f\ \vec{\rho})\ \mathtt{in} \\ \qquad w\ v\ (\pi[\varphi_3,\varphi]\ x)) : \varphi_3^* \to (s_2)_\perp^*\end{array}}$$

where the first and last lines follow from the Adequacy Theorem, the second and fourth by an argument about the meanings of terms, and the third by Proposition 5.3. ■

Intuitively, this result says that if a computation has effect φ and produces a value in region ρ_0 according to the region calculus, then that computation cannot be influenced by values in region ρ *unless* φ contains $\mathtt{get}(\rho)$. In contrast, Tofte and Talpin [23] give a low-level operational semantics of the region calculus with explicit memory allocation and deallocation, and present region type system as a translation from a fragment of ML into the region calculus. Their main result [23, Theorem 6.1] is that if e is translated into e', and e evaluates to v (in ML), then e' (in the region calculus) evaluates to a value v' such that v' is "equivalent" to v. This shows that memory deallocation (in the region calculus) does not go wrong.

Clearly, these two theorems are at opposite ends of the spectrum with respect to their level of abstraction. The Tofte-Talpin theorem is based on an operational semantics that is close to an implementation model. However, the actual statement of their theorem is complex, particularly the definitions of consistency. In contrast, our theorem relies on a denotational model that is further removed from an implementation model, but it is more abstract and much simpler to state. The questions of how these results relate, or which is more useful in practice, are open.

6 Region Polymorphism

In the full region calculus [23], it is possible to write region-polymorphic functions in recursive function definitions, where the polymorphism is over region and effect variables only. For instance, one may write

$$(\mathtt{letrec}\,(f[\rho_1,\rho_2],\rho)(x : (\mathtt{int},\rho_1)) = e_1\ \mathtt{in}\, e_2)$$

where ρ_1 is the region where x is located, ρ_2 is the region of the return result, and ρ is the region of the recursive function. Region polymorphism is useful especially because of recursive calls: the actual parameters to a recursive call can be placed in regions different from the regions of the original parameters. For example, a recursive call $((f[\rho_3,\rho_4]\ \rho_5)\ (1,\rho_3))$ gets the parameter from ρ_3 and allocates space for the result in ρ_4.

We can extend our grammar for types, values, and expressions of the region calculus from Section 2 to allow region polymorphism over region variables. We redefine

$$\begin{array}{rcl} s,t & ::= & (\mathtt{int},\rho) \mid (\forall\vec{\rho}.\ s \xrightarrow{\varphi} t,\rho) \\ v & ::= & (n,\rho) \mid ((\mathtt{fun}\, f\,(x : s) = e),\rho) \end{array}$$

add the form $(\mathtt{copy}[\rho]\ e)$ to expressions, and replace the expression form $(e\ e')$ with $(e\ [\vec{\rho}]\ e')$. Thus, all functions are now potentially recursive and polymorphic over regions, and all applications must involve applications of region variables. The copy operation copies a value from one region to another, and is useful in allocating recursive calls in different regions. Both the new typing rules and the translations appear in Table 5. The translation of function types is modified so that $(\forall\vec{\rho}.\ s \xrightarrow{\varphi} t)^\dagger = (\forall\vec{\rho}.\ s^* \to \varphi^* \to t_\perp^*)$.

7 Discussion

We have shown how to prove the correctness of the region calculus via a translation into $F_\#$. The main novelties of the paper are twofold: first, we point out the close correspondence between the region calculus and the polymorphic lambda calculus; and second, we show that there are certain differences that motivate the extensions present in $F_\#$. Along the way, we show that the region calculus can be simplified by eliminating a side condition on the typing rule for functional abstraction.

Our work on modelling type systems that track dependency [1] originally led us to consider the region calculus. In the work on dependency, the target of the translation is a language called the "dependency core calculus", or DCC for short. The semantics of DCC uses logical relations, as we do here. However, our attempts to use DCC to model the region calculus failed. It would be interesting to see if there is some unification of the ideas into one calculus; possibly $F_\#$ already has enough structure to model notions of dependency.

We believe that $F_\#$ is a good candidate for modelling other region-based calculi. For instance, Aiken, Fähndrich, and Levien [3] describe a type system where the allocation and deallocation of regions is split from the scoping declaration of `letregion`. Other papers develop the more practical aspects of region-based memory management [4, 22]. A recent paper of Crary, Walker, and Morrisett describes a different type system [5] for regions based on capabilities. We plan to construct translations for these calculi into $F_\#$. For example, the capabilities in the calculus of Crary, Walker, and Morrisett may be closely connected with our use of functions to model the `get` and `put` effects; if the type does not mention `get` or `put`, there is no capability for accessing the region. Such translations might uncover simplifications in these other region calculi, as well as give greater confidence in their correctness.

We conjecture that a translation from the original effect systems for state [21] may be possible for a version of $F_\#$ with recursive types. In these effect systems, only assignable reference cells are associated with regions, and terms may be polymorphic over regions.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1999.

[2] M. Abadi and G. D. Plotkin. A PER model of polymorphism and recursive types. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 355–365, 1990.

[3] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1995.

[4] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 171–183. ACM, 1996.

[5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1999.

[6] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[7] P. Fradet. Collecting more garbage. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 24–33. ACM, 1994.

[8] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, 1971.

[9] H. Hosoya and A. Yonezawa. Garbage collection via dynamic type inference — a formal treatment. In *Proc. Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1998.

[10] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57. ACM, 1988.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[12] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda calculus. *J. ACM*, 1999. To appear.

[13] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42:658–709, 1995.

[14] C.-H. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. PhD thesis, Imperial College, University of London, 1988.

[15] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[16] J. C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[17] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[18] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.

[19] D. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Sci.*, 121:411–440, 1993. Published version of unpublished manuscript, Oxford University, 1969.

[20] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.

[21] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.

[22] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Programming Languages and Systems*, 20(4):734–767, July 1998.

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