

# Syntactic Type Soundness Results for the Region Calculus

Cristiano Calcagno<sup>†</sup>, Simon Helsen<sup>‡</sup> and Peter Thiemann<sup>‡</sup>

<sup>†</sup>*Queen Mary University of London, England and DISI, University of Genova, Italy*

<sup>‡</sup>*Institut für Informatik, Universität Freiburg, Germany*

---

The region calculus of Tofte and Talpin is a polymorphically typed lambda calculus with annotations that make memory allocation and deallocation explicit. It is intended as an intermediate language for implementing Hindley-Milner typed functional languages such as ML without traditional trace-based garbage collection. Static region and effect inference can be used to annotate a statically typed ML program with memory management primitives. Soundness of the calculus with respect to the region and effect system is crucial to guarantee safe deallocation of regions, i.e., deallocation should only take place for objects which are provably dead.

The original soundness proof by Tofte and Talpin requires a complex co-inductive safety relation. In this paper, we present two small-step operational semantics for the region calculus and prove their type soundness with respect to the region and effect system. Following the standard syntactic approach of Wright, Felleisen, and Harper, we obtain simple inductive proofs.

The first semantics is store-less. It is simple and elegant and gives rise to perspicuous proofs. The second semantics provides a store-based model for the region calculus. Albeit slightly more complicated, its additional expressiveness allows to model operations on references with destructive update. A pure fragment of both small-step semantics is then proven equivalent to the original big-step operational approach of Tofte and Talpin. This leads to an alternative soundness proof for their evaluation-style formulation.

---

*Key Words:* operational semantics; region calculus; type soundness

## 1. INTRODUCTION

Memory management for dynamic data structures is a problem in programming. While memory allocation is dictated by the problem at hand, there is considerable freedom in memory deallocation. If deallocation happens too late, the program suffers from memory bloat and space leaks, which impede performance. If deallocation happens too early, there might be dangling pointers into deallocated memory. Dereferencing a dangling pointer is unsafe and can lead to a crash, or worse, to wrong results.

Some languages (like C or Pascal) leave the deallocation problem entirely to the programmer, whereas others (like Lisp, Smalltalk, Java, and ML) perform automatic deallo-

cation by incorporating a trace-based garbage collector into the runtime system. While the programmer-based solution is immensely error-prone, programs can in principle be tuned for optimal memory use. Traditional garbage collection avoids a large class of errors, but it has some problems, too. Since the garbage collector is, in general, unaware of the semantics of the running program, it must preserve all pointers reachable from a given set of root pointers. This set is a conservative approximation of the set of pointers that will actually be used by the program. As a consequence, deallocation might happen too late, which can lead to space leaks. In addition, trace-based garbage collection takes extra, non-productive time and can cause erratic pauses in the execution of programs, hampering its use for real-time applications. Finally, inter-operability between garbage collected languages and non-garbage collected languages is difficult.

The region calculus of Tofte and Talpin [15, 14] (which we will refer to as TTRC) provides an alternative method of memory management for the functional language ML [10]. It is used as an intermediate language in an ML compiler, the ML-kit [2, 3, 12, 14, 15]. The basic idea of the region calculus is to split memory into regions that are allocated in a stack-like manner, directed by a construct of the language. Deallocation is instantaneous, it just pops the topmost region from the stack. Using this method, it is possible to implement ML without a trace-based garbage collector. In some instances, the region calculus can prove that a pointer is semantically dead, even though it is still reachable by the program. In these cases, the region it points to can be safely deallocated, something trace-based garbage collectors cannot do.

### 1.1. Related Work

The first proof of consistency, or type soundness, for the region calculus as it is given by Tofte and Talpin [15] is quite complicated and uses rule-based co-induction. The source of complication is twofold. First, Tofte and Talpin prove two properties at the same time: type soundness and *translation soundness*. The latter property guarantees that there is some relation between a non-region annotated value and its region-based counterpart. In this paper, we focus on the problem of type soundness, ie. the property which guarantees that regions are not deallocated while they are still in use.

The second source of complication is due to the co-inductive definition of their consistency relation, required because of the loss of information when deleting a region from the store in their big-step semantics. Their safety relation not only requires a co-inductive proof, but is rather complex and lacks intuition of why deallocation safety is obtained.

Recently, alternative type-soundness proofs for the region calculus have been proposed.

1. Crary, Walker, and Morrisett [5] provide an indirect soundness proof by translating the region calculus into their capability calculus. The capability calculus has a sophisticated type-and-effect system that supports safe allocation and deallocation of regions in an arbitrary order. This added flexibility may lead to a better use of memory at runtime, since there are cases where a region may be de-allocated earlier than in the region calculus. They provide a syntactic soundness proof for the capability calculus.

2. Banerjee, Heintze, and Riecke [1] translate the region calculus into  $F_{\#}$ , an extension of the polymorphic lambda calculus with a special type constructor for encapsulation. They construct an original denotational model for their calculus and give a semantic soundness proof based on the model.

3. Dal Zilio and Gordon [17] modify the operational semantics of Tofte and Talpin so that it also keeps track of deallocated regions. This extra information allows an inductive definition of the consistency relation and an inductive correctness proof. Then they go on to show that this result is a consequence of a more general result for a typed  $\pi$ -calculus with name groups. This is shown by using a translation from the region calculus into the typed  $\pi$ -calculus with name groups.

4. Helsen and Thiemann [9] define a store-less small-step operational semantics for the region calculus and prove type soundness using the syntactic method of Wright, Felleisen, and Harper.

5. Calcagno [4] defines a high-level big-step operational semantics and proves type soundness for it. Calcagno formally relates the high-level semantics to the original low-level semantics of TTRC.

## 1.2. Contribution and Overview

The present paper is based on the work of Calcagno, Helsen, and Thiemann [4, 9]. After recalling the syntax and the semantics of TTRC in Section 2, Section 3 gives a simplified account of a *store-less region calculus* (abbreviated SRC), using the reduction-style formulation pioneered by Plotkin [11]. Its syntactic type soundness is stated without proofs and without the treatment of polymorphism and recursion, which can be found elsewhere [9].

While the store-less formulation is extremely simple and elegant, it is desirable to model a calculus with references and destructive update. Therefore, in section 4, we introduce a new calculus with an explicitly passed store: the *imperative region calculus* or IRC. This calculus extends SRC (and TTRC) with operations on references, as they are actually implemented in the ML-kit [3]. We also give a small-step operational semantics, similar in spirit to the definition of the store-less region calculus.

Then, using the syntactic approach of Wright and Felleisen [16], in a variation pioneered by Harper [7], we prove type soundness of IRC without the standard treatment of polymorphism and recursion. Adding polymorphism and recursion makes the proofs more technical, but does add new concepts. The resulting proofs all follow a relatively simple inductive pattern, and are therefore considerably easier than the co-inductive proofs of Tofte and Talpin.

In previous work, Calcagno [4] proves type soundness of TTRC by defining a store-less big-step operational semantics, which is parametric in a set of currently allocated regions. He proves his store-less semantics equivalent to TTRC.

Inspired by this work, section 5 shows the equivalence of TTRC with IRC, as well as the equivalence of IRC and SRC. However, instead of relating two big-step semantics, we relate a big-step semantics (TTRC) with a small-step semantics (IRC) on the one hand and two small-step semantics (IRC and SRC) on the other hand. The former result leads to type-soundness of TTRC. In these equivalences, we ignore the reference operations of IRC for simplicity of the presentation. Finally, section 6 concludes.

## 1.3. Notation

Let  $s$  be a partial function or *map* from a set  $A$  to a set  $B$ . Then  $\text{dom}(s) = \{x \in A \mid \exists y \in B, s(x) = y\}$  is the domain of  $s$  and  $\text{ran}(s) = \{y \in B \mid \exists x \in A, s(x) = y\}$  is the range of  $s$ . The map  $s|_{A'}$  is the *restriction* of  $s$  to  $A' \subseteq \text{dom}(s)$ , defined by  $s|_{A'}(a) = s(a)$ ,

if  $a \in A'$ , and undefined otherwise. If  $x \in A$ , write  $s - x$  for the map  $s|_{\text{dom}(s) \setminus \{x\}}$ . Write  $s + \{a \mapsto b\}$  for the extended map  $s'$  defined by  $s'(x) = b$ , if  $x = a$ , and  $s'(x) = s(x)$ , if  $x \neq a$ .

Given two finite maps  $s_1$  and  $s_2$  where  $\text{ran}(s_1) \cup \text{ran}(s_2)$  is a set of finite maps, then write  $s_1 \leq s_2$  ( $s_2$  extends  $s_1$ ) if  $\text{dom}(s_1) \subseteq \text{dom}(s_2)$  and for all  $r \in \text{dom}(s_1)$ ,  $\text{dom}(s_1(r)) \subseteq \text{dom}(s_2(r))$  and  $s_2(r)|_{\text{dom}(s_1(r))} = s_1(r)$ . All the maps considered in this work are finite.

The notation,  $e[x \mapsto e']$ , stands for the term  $e$  with  $e'$  substituted for each free occurrence of  $x$ . As usual, substitution avoids capture of variables by renaming. Write  $e \equiv e'$  if  $e$  and  $e'$  are syntactically equal.

## 2. THE REGION CALCULUS OF TOFTE AND TALPIN

The original TTRC includes ML-style polymorphism and polymorphic recursion on regions. To simplify the present account, we consider a non-recursive monomorphic version of TTRC. The addition of polymorphism and recursion is tedious, but standard [9, 4].

### 2.1. Syntax of TTRC

The following grammar defines the syntax of TTRC.

*Terms*  $e ::= x \mid c \text{ at } \varrho \mid \lambda x. e \text{ at } \varrho \mid e @ e \mid \text{letregion } \varrho \text{ in } e \mid \text{copy } [\varrho, \varrho'] e$   
*Values*  $v ::= (r, l)$

Moreover, assume that

$x \in Vname$	variable names
$c \in Cname$	constant symbols
$\varrho \in RegionVars$	region variables
$r \in RegionNames$	region names
$l \in Locations$	store locations

where the sets  $Vname$ ,  $Cname$ ,  $RegionVars$ ,  $RegionNames$ ,  $Locations$  are mutually disjoint denumerable sets. Occurrences of  $\varrho$  in  $e$  are bound by  $\text{letregion } \varrho \text{ in } e$  and are subject to alpha-conversion.

In comparison to the lambda calculus, constants and lambda abstractions carry a region annotation  $\text{at } \varrho$ , which indicates the region in which the value is allocated. Since constants also carry this annotation, the region calculus formalizes a fully boxed implementation strategy. Executing  $\text{letregion } \varrho \text{ in } e$  allocates a new region of memory, then evaluates  $e$ , and finally deallocates the region again. The term  $\text{copy } [\varrho, \varrho'] e$  is a simple addition to the original region calculus. It copies a value of base type from one region to another and stands for a prototypical primitive operation. Finally, a *program* is a term without free variables  $x \in Vname$ , but may contain free occurrences of region variables  $\varrho \in RegionVars$ .

### 2.2. Dynamic Semantics of TTRC

This section paraphrases the dynamic semantics of TTRC using the original big-step operational formulation [15]. First, there are a few definitions.

A *region environment*,  $R$ , is a finite map from region variables  $\varrho$  to region names  $r$ . A *value environment*,  $VE$ , is a finite map from variables to values. A *storable value* is either a constant,  $\langle c \rangle$ , or a closure  $\langle x, e, VE, R \rangle$ , which consists of a formal parameter,  $x$ , the body of the closure,  $e$ , a value environment, and a region environment. A *region* is a finite

$$\begin{aligned}
R \in \text{RegionEnv} &= \text{RegionVars} \rightarrow \text{RegionNames} \\
VE \in \text{ValueEnv} &= \text{Vname} \rightarrow \{(r, l) \mid r \in \text{RegionNames}, l \in \text{Locations}\} \\
S \in \text{Store} &= \text{RegionNames} \rightarrow (\text{Locations} \rightarrow \text{Contents}) \\
\text{Contents} &= \{\langle c \rangle \mid c \in \text{Cname}\} \\
&\cup \{\langle x, e, VE, R \rangle \mid x \in \text{Vname}, e \in \text{Terms}, \\
&\quad VE \in \text{ValueEnv}, R \in \text{RegionEnv}\}
\end{aligned}$$

$$\begin{aligned}
(b\text{-const}) \quad & \frac{R(\varrho) = r \quad l \notin \text{dom}(S(r))}{S, VE, R \vdash_b c \text{ at } \varrho \Downarrow (r, l), S + \{r \mapsto S(r)\} + \{l \mapsto \langle c \rangle\}} \\
(b\text{-var}) \quad & \frac{VE(x) = v}{S, VE, R \vdash_b x \Downarrow v, S} \\
(b\text{-abstr}) \quad & \frac{R(\varrho) = r \quad l \notin \text{dom}(S(r))}{S, VE, R \vdash_b \lambda x. e \text{ at } \varrho \Downarrow (r, l), S + \{r \mapsto S(r)\} + \{l \mapsto \langle x, e, VE, R \rangle\}} \\
(b\text{-app}) \quad & \frac{S, VE, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1 \quad S_1(r_1)(l_1) = \langle x, e, VE', R' \rangle}{S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2 \quad S_2, VE' + \{x \mapsto v_2\}, R' \vdash_b e \Downarrow v, S'} \\
& \frac{}{S, VE, R \vdash_b e_1 @ e_2 \Downarrow v, S'} \\
(b\text{-letregion}) \quad & \frac{r \notin \text{dom}(S) \quad S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\} \vdash_b e \Downarrow v, S'}{S, VE, R \vdash_b \text{letregion } \varrho \text{ in } e \Downarrow v, S' - r} \\
(b\text{-copy}) \quad & \frac{S, VE, R \vdash_b e \Downarrow (r, l), S' \quad S'(r)(l) = \langle c \rangle}{R(\varrho) = r \quad R(\varrho') = r' \quad l' \notin \text{dom}(S'(r'))} \\
& \frac{}{S, VE, R \vdash_b \text{copy}[\varrho, \varrho'] e \Downarrow (r', l'), S' + \{r' \mapsto S'(r')\} + \{l' \mapsto \langle c \rangle\}}
\end{aligned}$$

FIG. 1. Big-step evaluation relation of TTRC [15]

map from locations,  $l$ , to storable values. A *store*,  $S$ , is a finite map from region names to regions.

With these definitions, the big-step evaluation relation  $S, VE, R \vdash_b e \Downarrow v, S'$  relates the expression,  $e$ , using the store,  $S$ , the value environment,  $VE$ , and the region environment,  $R$ , to its value  $v$  and the final store  $S'$ . Figure 1 shows its definition.

The following lemma states that the final store of an evaluation extends the initial store and the domains are identical. See also lemma 4.1. in [15]:

LEMMA 2.1.  $S_1, VE, R \vdash_b e \Downarrow v, S_2$  implies  $S_1 \leq S_2$  and  $\text{dom}(S_1) = \text{dom}(S_2)$

### 2.3. Static Semantics of the Region Calculus

The static semantics deals with several semantic objects, but first, we introduce the notion of a *region placeholder*,  $\rho$ , which ranges —for the moment— over region variables,  $\varrho$ . A region placeholder is merely a notational convenience to “reuse” type and reduction rules. Its use will become clearer in due course when we extend place holders to range over additional objects.

An *effect*  $\varphi$  is a finite set of region place-holders,  $\rho$ . The effect of a term,  $e$ , indicates the set of regions that may be affected by evaluation of  $e$ . Tofte and Talpin distinguish between *get*( $\varrho$ ) and *put*( $\varrho$ ) effects to denote whether an object is read from a region or written into a region, respectively. This qualification is irrelevant for type soundness and can easily be added if desired. *Types*,  $\tau$ , and *types with place*,  $\mu$ , are defined by mutual induction:

$$\begin{aligned}\tau &::= \text{int} \mid \mu \xrightarrow{\varphi} \mu \\ \mu &::= (\tau, \rho)\end{aligned}$$

A type is either an integer type or a function type. Function types carry a latent effect, which is produced when an argument is supplied to the function.

A type with place,  $\mu$ , is a pair of a type,  $\tau$ , and a region placeholder,  $\rho$ . The latter specifies where an object of type  $\tau$  is stored.

A *type environment*,  $TE$ , is a finite map from variables to types with place. The empty type environment is written  $\{ \}$ .

Write  $frv(\mu)$  and  $frv(\varphi)$  for the set of region variables that occur in  $\mu$  and  $\varphi$  respectively. Analogously,  $frv(TE) = \bigcup \{ frv(TE(x)) \mid x \in \text{dom}(TE) \}$ .

Figure 2 defines the static semantics of TTRC in terms of the judgment  $TE \vdash_{tt} e : \mu, \varphi$ . Its intended meaning is that in type environment  $TE$  the term  $e$  has type with place  $\mu$  and effect  $\varphi$ . It differs from the presentation of Tofte and Talpin [15] in two respects: first, as stated earlier, we are omitting recursion and polymorphism. Secondly, we ignore effect variables, which are an artefact for type unification [13]. In the absence of region polymorphism, they can be omitted from the type system.

### 3. THE STORE-LESS REGION CALCULUS

The store-less region calculus [9] provides a very simple and elegant syntactic soundness proof. It is based on the observation that terms like  $c \text{ at } \rho$  cannot be used as values. This is because evaluation of the term  $c \text{ at } \rho$  allocates memory, stores the constant  $c$  in it, and returns a pointer to the stored constant. As a consequence, the store-less formulation of the region calculus (Figure 3) must include terms to express pointers to constants and pointers to functions. These are the values  $\langle c \rangle_\rho$  and  $\langle \lambda x. e \rangle_\rho$ . Their region annotation indicates the region the pointer points to. A region placeholder is now either a region variable, as before, or a special constant  $\bullet$  (*dead region*) that denotes a deallocated region.

The reductions explicitly require a region variable  $\varrho$  to ensure that deallocated pointers cannot be used in a computation. For example,  $\langle \lambda x. e \rangle_\bullet$  is a dangling pointer to a function in a deallocated region. Such a pointer can be safely passed as a parameter, but invoking the function is not allowed.

The rules (1) and (2) deal with memory allocation of constants and lambda abstractions. The rules (4) and (5) are computation rules that define the `copy` operation and beta-value reduction. Rule (3) deallocates a region of memory by substituting  $\bullet$  for the `letregion`-bound region variable, once the body has turned into a syntactic value. The substitution

$$\begin{array}{c}
\text{(var)} \quad \frac{TE(x) = \mu}{TE \vdash_{tt} x : \mu, \emptyset} \\
\\
\text{(const)} \quad \frac{}{TE \vdash_{tt} c \text{ at } \rho : (\text{int}, \rho), \{\rho\}} \\
\\
\text{(abstr)} \quad \frac{TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_{tt} \lambda x. e \text{ at } \rho : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \{\rho\}} \\
\\
\text{(app)} \quad \frac{TE \vdash_{tt} e_1 : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \varphi_1 \quad TE \vdash_{tt} e_2 : \mu_1, \varphi_2}{TE \vdash_{tt} e_1 @ e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\rho\}} \\
\\
\text{(letregion)} \quad \frac{TE \vdash_{tt} e : \mu, \varphi \quad \varrho \notin \text{frv}(TE, \mu)}{TE \vdash_{tt} \text{letregion } \varrho \text{ in } e : \mu, \varphi \setminus \{\varrho\}} \\
\\
\text{(copy)} \quad \frac{TE \vdash_{tt} e : (\text{int}, \rho), \varphi}{TE \vdash_{tt} \text{copy}[\rho, \rho'] e : (\text{int}, \rho'), \varphi \cup \{\rho, \rho'\}}
\end{array}$$

FIG. 2. Static Semantics of TTRC [15]

$v[\varrho \mapsto \bullet]$  replaces all free occurrences of  $\varrho$  in  $v$  with  $\bullet$ . Finally, Rules (6) through (9) are context rules, which specify a left-to-right call-by-value semantics.

### 3.1. Static Semantics

In addition to the typing rules for TTRC (see Fig. 2), we need two new rules that account for the freshly acquired value terms. The rules are simple variations of the *(const)* and *(abstr)* rules of TTRC. The difference is that these rules *(stored-const)* and *(stored-abstr)* provide a typing for pointers and therefore have no effect.

$$\begin{array}{c}
\text{(stored-const)} \quad \frac{}{TE \vdash_{tt} \langle c \rangle_{\rho} : (\text{int}, \rho), \emptyset} \\
\\
\text{(stored-abstr)} \quad \frac{TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_{tt} \langle \lambda x. e \rangle_{\rho} : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \emptyset}
\end{array}$$

### 3.2. Type Soundness

This section proves the syntactic type soundness for the small-step transition relation  $\rightarrow_s$  with respect to the type system of Section 2.3 using the additional rules from Section 3.1. The proof is structured as usual: first we formulate some standard lemmas. Then, we prove *type preservation*, also known as *subject reduction* [16, 6], which states that a well-typed term remains well-typed under the small-step transition relation  $\rightarrow_s$ . The second result is the *progress* property, which states that a well-typed closed term is either a value or it can be further reduced. Taken together, these two results imply type soundness.

*Terms*  $e ::= v \mid x \mid c \text{ at } \rho \mid \lambda x. e \text{ at } \rho \mid e @ e \mid \text{letregion } \rho \text{ in } e \mid \text{copy} [\rho_1, \rho_2] e$   
*Values*  $v ::= \langle c \rangle_\rho \mid \langle \lambda x. e \rangle_\rho$   
*Placeholders*  $\rho ::= \varrho \mid \bullet$

Reduction rules

$$c \text{ at } \varrho \rightarrow_s \langle c \rangle_\varrho \quad (1)$$

$$\lambda x. e \text{ at } \varrho \rightarrow_s \langle \lambda x. e \rangle_\varrho \quad (2)$$

$$\text{letregion } \varrho \text{ in } v \rightarrow_s v[\varrho \mapsto \bullet] \quad (3)$$

$$\text{copy} [\varrho_1, \varrho_2] \langle c \rangle_{\varrho_1} \rightarrow_s \langle c \rangle_{\varrho_2} \quad (4)$$

$$\langle \lambda x. e \rangle_\varrho @ v \rightarrow_s e[x \mapsto v] \quad (5)$$

Reduction in context

$$\frac{e \rightarrow_s e'}{e @ e'' \rightarrow_s e' @ e''} \quad (6)$$

$$\frac{e \rightarrow_s e'}{v @ e \rightarrow_s v @ e'} \quad (7)$$

$$\frac{e \rightarrow_s e'}{\text{letregion } \varrho \text{ in } e \rightarrow_s \text{letregion } \varrho \text{ in } e'} \quad (8)$$

$$\frac{e \rightarrow_s e'}{\text{copy} [\rho_1, \rho_2] e \rightarrow_s \text{copy} [\rho_1, \rho_2] e'} \quad (9)$$

**FIG. 3.** Small-step reduction relation for SRC

We just state the lemmas and theorems here. Full proofs, including the treatment of polymorphism and recursion, may be found elsewhere [9]. Also, the proofs of type soundness for IRC, detailed in section 4.4, bear a lot of similarity with those for the lemmas and theorems below.

First, we observe that syntactic values have no effect.

LEMMA 3.1. *For all  $TE$ , values  $v$ , and types  $\mu$ , if  $TE \vdash_{tt} v : \mu, \varphi$  then  $\varphi = \emptyset$ .*

The set of closed expressions is closed under small-step transition.

LEMMA 3.2. *If  $e$  is closed and  $e \rightarrow_s e'$ , then  $e'$  is also closed.*

Substitution of a value of the correct type for a variable of the same type preserves the type of the enclosing term.

LEMMA 3.3. *Suppose  $TE + \{x \mapsto \mu\} \vdash_{tt} e : \mu', \varphi'$  and  $TE \vdash_{tt} v : \mu, \emptyset$ , then  $TE \vdash_{tt} e[x \mapsto v] : \mu', \varphi'$ .*

Typing is preserved under region substitution.

LEMMA 3.4. *If  $TE \vdash_{tt} e : \mu, \varphi$  then, for all substitutions  $S_r$  that map region variables to region place-holders, we have that  $S_r(TE) \vdash_{tt} S_r(e) : S_r(\mu), S_r(\varphi)$ .*

The following proposition states type preservation: if a well-typed term can be reduced, then its reduct has the same type as the original term, but possibly less effect.

PROPOSITION 3.1 (Type Preservation). *Suppose  $TE \vdash_{tt} e : \mu, \varphi$ . If  $e \rightarrow_s e'$  then there exists an effect  $\varphi'$  for which  $TE \vdash_{tt} e' : \mu, \varphi'$  and  $\varphi' \subseteq \varphi$ .*

The canonical forms lemma determines the form of a value, given its type.

LEMMA 3.5 (Canonical Forms). *The following hold:*

1. *If  $TE \vdash_{tt} v : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \varphi$  then there exist a  $x$  and  $e$  such that  $v = \langle \lambda x. e \rangle_\rho$ .*
2. *If  $TE \vdash_{tt} v : (\text{int}, \rho), \varphi$  then there exists a  $c$  such that  $v = \langle c \rangle_\rho$ .*

The progress property states that a closed, well-typed term is either a syntactic value or can be further reduced, unless it affects a dead region.

PROPOSITION 3.2 (Progress). *If  $\{ \} \vdash_{tt} e : \mu, \varphi$  and  $\bullet \notin \varphi$  then either*

1. *there exists  $e'$  such that  $e \rightarrow_s e'$  or*
2.  *$e$  is a value.*

Finally, the type soundness theorem says that a well-typed closed term either gives rise to an infinite reduction sequence, or it eventually reduces to a value of the same type. Let  $\rightarrow_s$  be the reflexive and transitive closure of the relation  $\rightarrow_s$ .

<i>Terms</i>	$e ::= v \mid x \mid c \text{ at } \rho \mid \lambda x. e \text{ at } \rho \mid e @ e \mid \text{copy}[\rho, \rho] e \mid$ $\text{letregion } \varrho \text{ in } e \mid \text{region } r \text{ in } e \mid$ $\text{ref } e \text{ at } \rho \mid ! e \mid e := e$
<i>Values</i>	$v ::= (r, l) \mid (\bullet, l)$
<i>Placeholders</i>	$\rho ::= \varrho \mid r \mid \bullet$
<i>Storables</i>	$w ::= \langle c \rangle \mid \langle \lambda x. e \rangle \mid \langle \text{ref } v \rangle$
<i>Types with places</i>	$\mu ::= (\tau, \rho)$
<i>Types</i>	$\tau ::= \text{int} \mid \mu \xrightarrow{\varphi} \mu \mid \text{ref } \mu$

FIG. 4. Syntactic categories of IRC

**THEOREM 3.1 (Type Soundness).** *Suppose that  $\{ \} \vdash_{tt} e : \mu, \varphi$  with  $\bullet \notin \varphi$ . Then, either there exists some value  $v$  with  $e \rightarrow_s v$  and  $\{ \} \vdash_{tt} v : \mu, \emptyset$  or, for each  $e'$  where  $e \rightarrow_s e'$ , there exists  $e''$  with  $e' \rightarrow_s e''$ .*

#### 4. THE IMPERATIVE REGION CALCULUS

The imperative region calculus extends TTRC with operations on references. Hence, operationally, IRC is a close match to the actual intermediate language used in the ML-kit. The update operations require an explicit store in the semantics, so the challenge is to come up with a store-based small-step operational semantics that again admits a syntactic type soundness proof.

##### 4.1. Syntax

Figure 4 shows the entire syntax of IRC. The set of terms now includes the usual operations on references in ML notation: creation of a reference,  $\text{ref } e \text{ at } \rho$ , dereferencing a reference,  $! e$ , and updating a reference  $e := e$ . In addition, there is a new intermediate term,  $\text{region } r \text{ in } e$ . The binding construct,  $\text{letregion } \varrho \text{ in } e$ , reduces to  $\text{region } r \text{ in } e[\varrho \mapsto r]$  as soon as evaluation has committed to a particular region. It is important to note that  $\text{region } r \text{ in } e$  does not bind  $r$ , but merely records the region used in the store. This annotation is required to successfully deallocate  $r$  after finishing the evaluation of  $e$ .

A value is once again a pointer, that is, a pair of a live ( $r$ ) or dead ( $\bullet$ ) region and a location  $l \in \text{Locations}$ . Place-holders are either region variables, region names, or the constant  $\bullet$ . In comparison to the definition of TTRC-*Values* (Sec. 2.1), the present definition allows for values of the form  $(\bullet, l)$ . Such a value denotes a pointer into a deallocated region. In contrast, a live value has the form  $(r, l)$  for some region name  $r$ .

Similarly to the original big-step semantics (Sec. 2.2), a *store*,  $s$ , is a finite map from region names to regions. A *region*,  $p$ , is a finite map from locations to storables. A *storable*,  $w$ , is either a constant, a closed lambda abstraction, or a reference to a value.

Types with places, types, and effects are as before. The only extension is the type  $\text{ref } \mu$  of references to objects of type (with place)  $\mu$ .

$$\begin{aligned}
& s, \text{let region } \varrho \text{ in } e \rightarrow_i s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e[\varrho \mapsto r] & (10) \\
& \quad \text{where } r \notin \text{dom}(s) \\
& s, \text{region } r \text{ in } v \rightarrow_i s[r \mapsto \bullet] - r, v[r \mapsto \bullet] & (11) \\
& s, c \text{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l) & (12) \\
& \quad \text{where } l \notin \text{dom}(s(r)) \\
& s, \lambda x. e \text{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda x. e \rangle\}\}, (r, l) & (13) \\
& \quad \text{where } l \notin \text{dom}(s(r)) \\
& s, (r, l) @ v \rightarrow_i s, e[x \mapsto v] & (14) \\
& \quad \text{where } s(r)(l) = \langle \lambda x. e \rangle \\
& s, \text{copy } [r_1, r_2] (r_1, l_1) \rightarrow_i s + \{r_2 \mapsto s(r_2) + \{l_2 \mapsto \langle c \rangle\}\}, (r_2, l_2) & (15) \\
& \quad \text{where } s(r_1)(l_1) = \langle c \rangle \text{ and } l_2 \notin \text{dom}(s(r_2)) \\
& s, \text{ref } v \text{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \text{ref } v \rangle\}\}, (r, l) & (16) \\
& \quad \text{where } l \notin \text{dom}(s(r)) \\
& s, ! (r, l) \rightarrow_i s, v & (17) \\
& \quad \text{where } s(r)(l) = \langle \text{ref } v \rangle \\
& s, (r, l) := v \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \text{ref } v \rangle\}\}, v & (18) \\
& \quad \text{where } s(r)(l) = \langle \text{ref } v' \rangle
\end{aligned}$$

FIG. 5. Dynamic semantics of IRC - reduction rules

#### 4.2. Dynamic Semantics

The dynamic semantics is defined as a relation on configurations. A configuration  $s, e$  is a pair of a store,  $s$ , and a closed expression,  $e$ . Figure 5 shows the reduction rules, figure 6 the reductions in context.

With some abuse of notation,  $e[r \mapsto \bullet]$  denotes the term  $e$  after substitution of  $\bullet$  for all occurrences of the constant  $r$  in  $e$ . Write  $\text{frn}(e)$  and  $\text{frn}(w)$  for the sets of region names in  $e$  and  $w$  respectively, where again we stress that `region`  $r$  in  $e$  does not bind  $r$  in  $e$ . Then, define  $\text{frn}(s) = \bigcup \{\text{frn}(s(r)(l)) \mid r \in \text{dom}(s), l \in \text{dom}(s(r))\}$ . Now,  $s[r \mapsto \bullet]$  denotes the store  $s$  after substitution of  $\bullet$  for all  $r \in \text{frn}(s)$ .

Rule (10) allocates a fresh region  $r$  on the store, whereas rule (11) deallocates a region if the body of the `region`-expression has been reduced to a value. Rules (12), (13) and (16) store constants, lambdas and references respectively. Beta reduction is specified by Rule (14) and the `copy` primitive is reduced with rule (15). Finally, rules (17) and (18) define pointer dereferencing and destructive update respectively. The context rules of figure 6 specify a left-to-right call-by-value semantics for IRC.

#### 4.3. Static Semantics

To formulate the static semantics, we need two additional finite maps: a *region type*,  $K$ , mapping locations to types and a *heap type*,  $H$ , mapping a region name to a region type, thus providing the typing for locations in the store. The definition of  $\text{frn}(-)$  for type environments  $TE$ , types  $\mu$  and effects  $\varphi$  is analogous to the definition of  $\text{frv}(-)$  for those

$$\frac{s, e \rightarrow_i s', e'}{s, \text{region } r \text{ in } e \rightarrow_i s', \text{region } r \text{ in } e'} \quad (19)$$

$$\frac{s, e_1 \rightarrow_i s', e'_1}{s, e_1 @ e_2 \rightarrow_i s', e'_1 @ e_2} \quad (20)$$

$$\frac{s, e \rightarrow_i s', e'}{s, v @ e \rightarrow_i s', v @ e'} \quad (21)$$

$$\frac{s, e \rightarrow_i s', e'}{s, \text{copy } [r_1, r_2] e \rightarrow_i s', \text{copy } [r_1, r_2] e'} \quad (22)$$

$$\frac{s, e \rightarrow_i s', e'}{s, \text{ref } e \text{ at } r \rightarrow_i s', \text{ref } e' \text{ at } r} \quad (23)$$

$$\frac{s, e \rightarrow_i s', e'}{s, ! e \rightarrow_i s', ! e'} \quad (24)$$

$$\frac{s, e_1 \rightarrow_i s', e'_1}{s, e_1 := e_2 \rightarrow_i s', e'_1 := e_2} \quad (25)$$

$$\frac{s, e \rightarrow_i s', e'}{s, v := e \rightarrow_i s', v := e'} \quad (26)$$

**FIG. 6.** Dynamic semantics of IRC - reduction in context

$$\begin{array}{c}
\text{(pointer)} \quad \frac{H(r)(l) = \tau}{H, TE \vdash (r, l) : (\tau, r), \emptyset} \quad \text{(dead)} \quad \frac{}{H, TE \vdash (\bullet, l) : (\tau, \bullet), \emptyset} \\
\\
\text{(var)} \quad \frac{TE(x) = \mu}{H, TE \vdash x : \mu, \emptyset} \quad \text{(const)} \quad \frac{}{H, TE \vdash c \text{ at } \rho : (\text{int}, \rho), \{\rho\}} \\
\\
\text{(abstr)} \quad \frac{H, TE + \{x \mapsto \mu_2\} \vdash e : \mu_1, \varphi \quad \varphi \subseteq \varphi'}{H, TE \vdash \lambda x. e \text{ at } \rho : (\mu_2 \xrightarrow{\varphi'} \mu_1, \rho), \{\rho\}} \\
\\
\text{(app)} \quad \frac{H, TE \vdash e_1 : (\mu_2 \xrightarrow{\varphi} \mu_1, \rho), \varphi_1 \quad H, TE \vdash e_2 : \mu_2, \varphi_2}{H, TE \vdash e_1 @ e_2 : \mu_2, \varphi_1 \cup \varphi_2 \cup \varphi \cup \{\rho\}} \\
\\
\text{(copy)} \quad \frac{H, TE \vdash e : (\text{int}, \rho_1), \varphi}{H, TE \vdash \text{copy}[\rho_1, \rho_2] e : (\text{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}} \\
\\
\text{(letregion)} \quad \frac{H, TE \vdash e : \mu, \varphi \quad \varrho \notin \text{frv}(TE, \mu)}{H, TE \vdash \text{letregion } \varrho \text{ in } e : \mu, \varphi \setminus \{\varrho\}} \\
\\
\text{(useregion)} \quad \frac{H, TE \vdash e : \mu, \varphi \quad r \notin \text{frn}(TE, \mu)}{H, TE \vdash \text{region } r \text{ in } e : \mu, \varphi \setminus \{r\}} \\
\\
\text{(ref)} \quad \frac{H, TE \vdash e : \mu, \varphi}{H, TE \vdash \text{ref } e \text{ at } \rho : (\text{ref } \mu, \rho), \varphi \cup \{\rho\}} \\
\\
\text{(deref)} \quad \frac{H, TE \vdash e : (\text{ref } \mu, \rho), \varphi}{H, TE \vdash ! e : \mu, \varphi \cup \{\rho\}} \\
\\
\text{(setref)} \quad \frac{H, TE \vdash e_1 : (\text{ref } \mu, \rho), \varphi_1 \quad H, TE \vdash e_2 : \mu, \varphi_2}{H, TE \vdash e_1 := e_2 : \mu, \varphi_1 \cup \varphi_2 \cup \{\rho\}}
\end{array}$$

FIG. 7. Expression typing

objects. The definition of  $\text{frn}(H)$  is analogous to the definition of  $\text{frn}(s)$ . We need the following judgments:

$$\begin{array}{ll}
\text{expression typing} & H, TE \vdash e : \mu, \varphi; \\
\text{configuration typing} & H \vdash s, e : \mu, \varphi; \\
\text{heap typing} & H \vdash s; \text{ and} \\
\text{storables typing} & H \vdash w : \tau.
\end{array}$$

Expression typing (Fig. 7) is a simple extension of the typing judgment in TTRC. The heap type is only used to provide a typing for live pointers in rule *(pointer)*. A dead pointer

$$\begin{array}{c}
\text{(stored-const)} \quad \frac{}{H \vdash \langle c \rangle : \text{int}} \\
\text{(stored-abstr)} \quad \frac{H, \{x \mapsto \mu_2\} \vdash e : \mu_1, \varphi \quad \varphi \subseteq \varphi'}{H \vdash \langle \lambda x. e \rangle : \mu_2 \xrightarrow{\varphi'} \mu_1} \\
\text{(stored-ref)} \quad \frac{H, \{ \} \vdash v : \mu, \emptyset}{H \vdash \langle \text{ref } v \rangle : \text{ref } \mu}
\end{array}$$

FIG. 8. Storable typing

can assume any type, due to rule *(dead)*. The remaining rules are as before, the heap type is just passed unchanged through the whole typing derivation.

The type rules *(ref)*, *(deref)* and *(setref)* are equivalent to the primitive type schemes for reference operations as defined in section 11.1 of [15].

*Remark.* The set of IRC-terms can be divided into *pure terms* (terms that do not contain sub-terms of the form `region r in e`) and *intermediate terms* (terms that do contain sub-terms of the form `region r in e`). A term of the form  $\langle \lambda x. e \rangle$  or `letregion  $\rho$  in e` only makes sense if  $e$  is pure. Furthermore, substitution always inserts pure terms (values  $(\rho, l)$ ) into pure terms.

There is just one rule for configuration typing: A configuration  $s, e$  has type with place  $\mu$  and effect  $\varphi$  under heap type  $H$ , if  $H$  is a valid heap type for  $s$  and  $e$  is a closed expression typeable with heap type  $H$ .

$$\text{(conf)} \quad \frac{H \vdash s \quad H, \{ \} \vdash e : \mu, \varphi}{H \vdash s, e : \mu, \varphi}$$

There is also just one rule for heap typing:

$$\text{(heap)} \quad \frac{\begin{array}{c} \text{dom}(H) = \text{dom}(s) \\ (\forall r \in \text{dom}(H)) \text{dom}(H(r)) = \text{dom}(s(r)) \\ (\forall r \in \text{dom}(H)) (\forall l \in \text{dom}(H(r))) H \vdash s(r)(l) : H(r)(l) \end{array}}{H \vdash s}$$

That is, the names of the regions in the heap type and the actual store must agree. Further, the domains of all regions must agree and, for each region  $r$  and each location  $l$  in  $r$ ,  $H$  must provide a valid type for the contents of  $s(r)(l)$ . The latter is asserted using a storable typing.

Storable typings (Fig. 8) provide the types for storables. Since all storables are closed, a storable typing simply refers to expression typing in the empty environment.

#### 4.4. Type Soundness

The type soundness proof is again structured in the standard way. Before we begin the proof, we have to formulate several lemmas.

LEMMA 4.1. *If  $H, TE \vdash v : \mu, \varphi$  then  $\varphi = \emptyset$ .*

LEMMA 4.2. *If  $H, TE \vdash e : \mu, \varphi$  then  $H, TE' + TE \vdash e : \mu, \varphi$ .*

LEMMA 4.3. *If  $H, TE \vdash e : \mu, \varphi$  and  $H \leq H'$  then  $H', TE \vdash e : \mu, \varphi$ .*

LEMMA 4.4. *If  $H, TE \vdash e : \mu, \varphi$  and  $r \in \text{dom}(H) \setminus \text{frn}(H, TE, e, \mu, \varphi)$  then  $H - r, TE \vdash e : \mu, \varphi$ .*

LEMMA 4.5 (Region Substitution). *Let  $\theta$  be a substitution that either maps a region name  $r$  to  $\bullet$  (i.e.,  $\theta_\bullet = \{r \mapsto \bullet\}$ ) or that maps a region variable to a region name (i.e.,  $\theta_r = \{\rho \mapsto r\}$ ).*

1. *If  $H, TE \vdash e : \mu, \varphi$  then  $\theta(H), \theta(TE) \vdash \theta(e) : \theta(\mu), \theta(\varphi)$ .*
2. *If  $H \vdash s$  then  $\theta(H) \vdash \theta(s)$ .*
3. *If  $H \vdash s, e : \mu, \varphi$  then  $\theta(H) \vdash \theta(s), \theta(e) : \theta(\mu), \theta(\varphi)$ .*
4. *If  $H \vdash w : \tau$  then  $\theta(H) \vdash \theta(w) : \theta(\tau)$ .*

*Proof.* By simultaneous induction on the derivations. The proof of the case for  $\theta_\bullet$  relies crucially on the typing rule (*dead*). ■

LEMMA 4.6 (Value Substitution).

*Suppose  $H, TE + \{x \mapsto \mu'\} \vdash e : \mu, \varphi$  and  $H, TE \vdash v : \mu', \emptyset$ . Then  $H, TE \vdash e[x \mapsto v] : \mu, \varphi$ .*

The following lemma is an adaptation of lemma 4.1 from [15] for IRC:

LEMMA 4.7. *Suppose  $e_1$  is a pure term (see remark). Then  $s_1, e_1 \rightarrow_i s_2, e_2$  implies  $s_1 \leq s_2$ . Moreover, if  $e_2 \in \text{Values}$  then  $\text{dom}(s_1) = \text{dom}(s_2)$ .*

Type preservation states that whenever a reduction is possible from a typed configuration, then the resulting configuration is also typed, but possibly with less effect.

PROPOSITION 4.1 (Type Preservation).

*Suppose  $H \vdash s, e : \mu, \varphi$ ,  $\text{frn}(\mu) \subseteq \text{dom}(H)$ , and  $s, e \rightarrow_i s', e'$ .*

*Then there exist  $H'$  and  $\varphi'$  such that  $H' \vdash s', e' : \mu, \varphi'$  where  $\varphi' \subseteq \varphi$ .*

By requiring that  $\text{frn}(\mu) \subseteq \text{dom}(H)$ , we guarantee that regions in use are allocated already.

*Proof.* By induction on the definition of  $s, e \rightarrow_i s', e'$ . We only consider some interesting cases.

**Case**  $s, \text{letregion } \varrho \text{ in } e \rightarrow_i s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e[\varrho \mapsto r]$   
 where  $r \notin \text{dom}(s)$ . Since  $H \vdash s, \text{letregion } \varrho \text{ in } e : \mu, \varphi$  is assumed it must be that  $H \vdash s$  and  $H, \{\} \vdash \text{letregion } \varrho \text{ in } e : \mu, \varphi$ , by rule (*conf*). By rule (*letregion*), it must be that  $H, \{\} \vdash e : \mu, \varphi'$  and  $\varrho \notin \text{frv}(\mu)$  and  $\varphi = \varphi' \setminus \{\varrho\}$ . By Lemma 4.5,  $H[\varrho \mapsto r], \{\}[\varrho \mapsto r] \vdash e[\varrho \mapsto r] : \mu[\varrho \mapsto r], \varphi'[\varrho \mapsto r]$ . Since  $\text{dom}(H) = \text{dom}(s)$ , by rule (*heap*) for  $H \vdash s$ , it follows that  $r \notin \text{dom}(H)$ . Therefore,  $r \notin \text{frn}(\mu)$  and because  $\varrho \notin \text{frv}(\mu)$ , the above judgment is equivalent to  $H, \{\} \vdash e[\varrho \mapsto r] : \mu, \varphi'[\varrho \mapsto r]$  where  $r$  does not occur in  $\mu$ . These are exactly the assumptions for rule (*useregion*), hence  $H, \{\} \vdash \text{region } r \text{ in } e : \mu, \varphi$ . Let  $H' = H + \{r \mapsto \{\}\}$ . Since  $H \leq H'$ , Lemma 4.3 yields that  $H', \{\} \vdash \text{region } r \text{ in } e : \mu, \varphi$ . Since  $H' \vdash s + \{r \mapsto \{\}\}$ , rule (*conf*) yields  $H' \vdash s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e : \mu, \varphi$ .

**Case**  $s, \text{region } r \text{ in } v \rightarrow_i s[r \mapsto \bullet] - r, v[r \mapsto \bullet]$ .

Since  $H \vdash s, \text{region } r \text{ in } v : \mu, \varphi$  is assumed it must be that  $H \vdash s$  and  $H, \{\} \vdash \text{region } r \text{ in } v : \mu, \varphi$ , by rule (*conf*). By rule (*useregion*), we have that  $H, \{\} \vdash v : \mu, \varphi'$ ,  $r \notin \text{frn}(\mu)$  and  $\varphi = \varphi' \setminus \{r\}$ . By Lemma 4.5,  $H[r \mapsto \bullet], \{\}[r \mapsto \bullet] \vdash v[r \mapsto \bullet] : \mu[r \mapsto \bullet], \varphi'[r \mapsto \bullet]$  also holds. By Lemma 4.1,  $\varphi' = \emptyset$ . Taken together with the assumption on the non-occurrence of  $r$  in  $\mu$  it holds that  $H[r \mapsto \bullet], \{\} \vdash v[r \mapsto \bullet] : \mu, \emptyset$ . As  $r \notin \text{frn}(H[r \mapsto \bullet], v[r \mapsto \bullet], \mu)$ , Lemma 4.4 implies that  $H', \{\} \vdash v[r \mapsto \bullet] : \mu, \emptyset$  where  $H' = H[r \mapsto \bullet] - r$ . Therefore, because  $H' \vdash s[r \mapsto \bullet] - r$ , rule (*conf*) yields that  $H' \vdash s[r \mapsto \bullet] - r, v[r \mapsto \bullet] : \mu, \emptyset$ .

**Case**  $s, \lambda x. e \text{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda x. e \rangle\}\}, (r, l)$

where  $l \notin \text{dom}(s(r))$ . By assumption,  $H \vdash s, \lambda x. e \text{ at } r : \mu, \varphi$ . By rule (*conf*) it must be that  $H \vdash s$  and  $H, \{\} \vdash \lambda x. e \text{ at } r : \mu, \varphi$ . Rule (*abstr*) gives that  $\mu = (\mu_1 \xrightarrow{\varphi'} \mu_2, r)$  and  $\varphi = \{r\}$ . Let  $H' = H + \{r \mapsto H(r) + \{l \mapsto \mu_1 \xrightarrow{\varphi'} \mu_2\}\}$  and  $s' = s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda x. e \rangle\}\}$ . Since  $\lambda x. e$  is closed and  $l \notin \text{dom}(s(r)) = \text{dom}(H(r))$ , rules (*stored-abstr*) and (*heap*) yield  $H' \vdash s'$ . Since  $H'(r)(l) = \mu_1 \xrightarrow{\varphi'} \mu_2$  rule (*pointer*) yields  $H', \{\} \vdash (r, l) : (\mu_1 \xrightarrow{\varphi'} \mu_2, r), \emptyset$ . Applying rule (*conf*) yields  $H' \vdash s', (r, l) : (\mu_1 \xrightarrow{\varphi'} \mu_2, r), \emptyset$ . This suffices the claim, since  $\emptyset \subseteq \{r\}$ .

**Case**  $s, (r, l) @ v \rightarrow_i s, e[x \mapsto v]$

where  $s(r)(l) = \langle \lambda x. e \rangle$ . By assumption,  $H \vdash s, (r, l) @ v : \mu, \varphi$ . By rule (*conf*) it must be that  $H \vdash s$  and  $H, \{\} \vdash (r, l) @ v : \mu, \varphi$ . From rule (*app*), we have that  $H, \{\} \vdash (r, l) : (\mu_2 \xrightarrow{\varphi'} \mu, r), \varphi_1$ , that  $H, \{\} \vdash v : \mu_2, \varphi_2$  and also that  $\varphi = \varphi_1 \cup \varphi_2 \cup \varphi' \cup \{r\}$ . So, from rule (*pointer*), it must be that  $H(r)(l) = \mu_2 \xrightarrow{\varphi'} \mu$  (this follows from (*heap*), too) and  $\varphi_1 = \emptyset$ . By rule (*heap*), it must be that  $H \vdash s(r)(l) : H(r)(l)$ . But this means that  $H \vdash \langle \lambda x. e \rangle : \mu_2 \xrightarrow{\varphi'} \mu$ . Therefore, by rule (*stored-abstr*), we have that  $H, \{x \mapsto \mu_2\} \vdash e : \mu, \varphi''$ , for some  $\varphi'' \subseteq \varphi'$ . Lemma 4.6 then gives  $H, \{\} \vdash e[x \mapsto v] : \mu, \varphi''$ , and hence, by rule (*conf*),  $H \vdash s, e[x \mapsto v] : \mu, \varphi''$ . This suffices for the claim because  $\varphi'' \subseteq \varphi' \subseteq \varphi$ .

**Case**  $s, ! (r, l) \rightarrow_i s, v$

where  $s(r)(l) = \langle \text{ref } v \rangle$ . From the assumption  $H \vdash s, ! (r, l) : \mu, \varphi$ . So, by rule (*conf*), it must be that  $H \vdash s$  and  $H, \{\} \vdash ! (r, l) : \mu, \varphi$ . From rule (*deref*), we have that  $H, \{\} \vdash (r, l) : (\text{ref } \mu, r), \varphi'$  and  $\varphi = \varphi' \cup \{\rho\}$ . Rule (*pointer*) gives that  $\rho = r$ . Rule (*heap*) gives  $H \vdash s(r)(l) : H(r)(l)$ , that is,  $H \vdash \langle \text{ref } v \rangle : \text{ref } \mu$ . Now, rule (*stored-ref*) says that  $H, \{\} \vdash v : \mu, \emptyset$  and applying rule (*conf*) yields  $H \vdash s, v : \mu, \emptyset$ . Conclude by observing that  $\emptyset \subseteq \varphi$ .

**Case**  $s, (r, l) := v \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \text{ref } v \rangle\}\}, v$ .

By assumption  $H \vdash s, (r, l) := v : \mu, \varphi$ . Again, by rule (*conf*), it must be that  $H \vdash s$  and  $H, \{ \} \vdash (r, l) := v : \mu, \varphi$ . By rule (*setref*), it holds that  $H, \{ \} \vdash (r, l) : (\text{ref } \mu, r), \varphi'$ , that  $H, \{ \} \vdash v : \mu, \varphi''$  and also that  $\varphi = \varphi' \cup \varphi'' \cup \{r\}$ .

So, from rule (*stored-ref*), we have that  $H \vdash \langle \text{ref } v \rangle : (\text{ref } \mu, r)$ . Hence, with  $s' = s + \{r \mapsto s(r) + \{l \mapsto \langle \text{ref } v \rangle\}\}$  this amounts to  $H \vdash s'(r)(l) : (\text{ref } \mu, r)$ , so that  $H \vdash s'$ .

Applying rule (*conf*) to the value yields  $H \vdash s', v : \mu, \emptyset$ . Conclude by observing that  $\emptyset \subseteq \varphi$ .

**Case**  $\frac{s, e \rightarrow_i s', e'}{s, \text{region } r \text{ in } e \rightarrow_i s, \text{region } r \text{ in } e'}$ .

By assumption,  $H \vdash s, \text{region } r \text{ in } e : \mu, \varphi$ . So, from rule (*conf*), we have that  $H \vdash s$  and  $H, \{ \} \vdash \text{region } r \text{ in } e : \mu, \varphi$ . By rule (*useregion*), it must be that  $H, \{ \} \vdash e : \mu, \varphi'$  and  $\varphi = \varphi' \setminus \{r\}$ .

By induction, there exist some  $H''$  and  $\varphi''$  such that  $H'' \vdash s', e' : \mu, \varphi''$  where  $\varphi'' \subseteq \varphi'$ .

Rule (*conf*) requires that  $H'' \vdash s'$  and  $H'', \{ \} \vdash e' : \mu, \varphi''$ . Rule (*useregion*) is applicable because  $\mu$  has not changed. It yields that  $H'', \{ \} \vdash \text{region } r \text{ in } e' : \mu, \varphi'' \setminus \{r\}$ . Finally, rule (*conf*) yields  $H'' \vdash s', \text{region } r \text{ in } e' : \mu, \varphi'' \setminus \{r\}$ . This yields the claim because  $\varphi'' \setminus \{r\} \subseteq \varphi' \setminus \{r\} = \varphi$ .

■

LEMMA 4.8 (Canonical Forms). *Suppose  $H \vdash s, (r, l) : \mu, \emptyset$ , then*

1. *If  $\mu = (\text{int}, r)$  then  $s(r)(l) = \langle c \rangle$  for some  $c$ ;*
2. *If  $\mu = (\mu_1 \xrightarrow{\varphi} \mu_2, r)$  then  $s(r)(l) = \langle \lambda x. e \rangle$ , for some  $x$  and  $e$ ;*
3. *If  $\mu = (\text{ref } \mu_1, r)$  then  $s(r)(l) = \langle \text{ref } v \rangle$ , for some  $v$ .*

*Proof.* Slightly more complicated than usual due to the indirection through the store.

**Case 1.** By rule (*conf*), we have that  $H \vdash s$ . By rule (*pointer*), it must be that  $H(r)(l) = \text{int}$ . Rule (*heap*) gives that  $H \vdash s(r)(l) : H(r)(l)$ , that is,  $H \vdash s(r)(l) : \text{int}$ . The only storable that fulfills this requirement is of the form  $\langle c \rangle$ : by rule (*stored-const*)  $H \vdash \langle c \rangle : \text{int}$ . Hence,  $s(r)(l) = \langle c \rangle$ .

**Cases 2. and 3.** Analogous.

■

PROPOSITION 4.2 (Progress). *Suppose  $H \vdash s, e : \mu, \varphi$  where  $\varphi \subseteq \text{RegionNames}$ . Then either*

1.  *$e$  is a value; or*
2. *there exist  $s', e'$  such that  $s, e \rightarrow_i s', e'$ .*

The assumption  $\varphi \subseteq \text{RegionNames}$  has two implications:

1.  $\bullet \notin \varphi$ , meaning that deallocated regions are not accessed, and

2. for all  $\varrho \in \text{RegionVars}$ ,  $\varrho \notin \varphi$ , implying that regions are not accessed before they are allocated.

*Proof.* By induction on the structure of  $e$ . Again, we only consider some non-trivial cases.

**Case**  $\lambda x. e \text{ at } \rho$ . By assumption  $H \vdash H, \lambda x. e \text{ at } \rho : \mu, \varphi$  with  $\varphi \subseteq \text{RegionNames}$ . Hence,  $\varphi = \{\rho\}$ ,  $\rho = r$  and  $\mu = (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho)$ . By reduction rule (13),  $s, \lambda x. e \text{ at } r$  is a redex (Case 2.).

**Case**  $e_1 @ e_2$ . By assumption  $H \vdash s, e_1 @ e_2 : \mu, \varphi$  with  $\varphi \subseteq \text{RegionNames}$ . By rules (*conf*) and (*app*), we have that  $H, \{ \} \vdash e_1 : (\mu_2 \xrightarrow{\varphi'} \mu, \rho), \varphi_1$ , that  $H, \{ \} \vdash e_2 : \mu_2, \varphi_2$ , and that  $\varphi = \varphi_1 \cup \varphi_2 \cup \varphi' \cup \{\rho\}$ , so  $\rho = r$  since  $\varphi \subseteq \text{RegionNames}$ .

If  $e_1$  is not a value then applying rule (*conf*) yields  $H \vdash s, e_1 : (\mu_2 \xrightarrow{\varphi'} \mu, r), \varphi_1$  with  $\varphi_1 \subseteq \text{RegionNames}$ . By induction,  $e_1$  is either a value (which contradicts our assumption) or there exist  $s'$  and  $e'_1$  such that  $s, e_1 \rightarrow s', e'_1$ . By rule (20),  $s, e_1 @ e_2 \rightarrow_i s', e'_1 @ e_2$ .

If  $e_1$  is a value then applying rule (*conf*) yields  $H \vdash s, e_2 : \mu_2, \varphi_2$  with  $\varphi_2 \subseteq \text{RegionNames}$ . Applying induction to  $e_2$  yields two cases.

If  $s, e_2 \rightarrow_i s', e'_2$  then, by rule (21),  $s, v_1 @ e_2 \rightarrow_i s', v_1 @ e'_2$ .

If  $e_2 = v_2$  is a value, too, then, by Lemma 4.8,  $e_1 = (r, l)$  and  $s(r)(l) = \langle \lambda x. e \rangle$ , for some  $x$  and  $e$ . Hence,  $s, (r, l) @ v_2$  is reduce-able with rule (14) (case 2.).

**Case**  $\text{ref } e \text{ at } \rho$ . By assumption  $H \vdash s, \text{ref } e \text{ at } \rho : \mu, \varphi$  with  $\varphi \subseteq \text{RegionNames}$ . Hence,  $\mu = (\text{ref } \mu_1, \rho)$ ,  $\varphi = \varphi' \cup \{\rho\}$  and  $\rho = r$ . So we conclude that  $s, \text{ref } e \text{ at } r$  is a redex (Case 2.).

**Case**  $! e$ . By assumption  $H \vdash s, ! e : \mu, \varphi$  with  $\varphi \subseteq \text{RegionNames}$ . By rules (*conf*) and (*deref*), it must be that  $H, \{ \} \vdash e : \mu', \varphi'$  where  $\mu' = (\text{ref } \mu, \rho)$ ,  $\varphi = \varphi' \cup \{\rho\}$  and  $\rho = r$ .

By rule (*conf*), it follows that  $H \vdash s, e : \mu', \varphi'$  where  $\varphi' \subseteq \text{RegionNames}$ .

By induction, either  $e$  is a value or it can be reduced. If  $e$  is a value then, by Lemma 4.8 item 3.,  $e = (r_1, l)$  and  $s(r_1)(l) = \langle \text{ref } v \rangle$ . Hence,  $s, ! (r_1, l)$  is a redex for reduction rule (17) (case 2.).

If  $s, e \rightarrow_i s', e'$  then, by rule (24),  $s, ! e \rightarrow_i s', ! e'$ .

**Case**  $e_1 := e_2$ . By assumption  $H \vdash s, e_1 := e_2 : \mu, \varphi$  with  $\varphi \subseteq \text{RegionNames}$ . By rules (*conf*) and (*setref*), we have that  $H, \{ \} \vdash e_1 : (\text{ref } \mu, \rho), \varphi_1$ , that  $H, \{ \} \vdash e_2 : \mu, \varphi_2$ , and that  $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$ , so  $\rho = r$  since  $\varphi \subseteq \text{RegionNames}$ .

If  $e_1$  is not a value then applying rule (*conf*) yields  $H, \{ \} \vdash s, e_1 : (\text{ref } \mu, r), \varphi_1$  with  $\varphi_1 \subseteq \text{RegionNames}$ . By induction,  $e_1$  is either a value (contradiction) or there exist  $s'$  and  $e'_1$  such that  $s, e_1 \rightarrow s', e'_1$ . By rule (25),  $s, e_1 := e_2 \rightarrow_i s', e'_1 := e_2$ .

If  $e_1$  is a value then applying rule (*conf*) yields  $H, s \vdash s, e_2 : \mu, \varphi_2$  with  $\varphi_2 \subseteq \text{RegionNames}$ .

Applying induction to  $e_2$  yields two cases. If  $e_2 = v_2$  is a value, then by Lemma 4.8,  $e_1 = (r, l)$  and  $s(r)(l) = \langle \text{ref } v \rangle$ , for some  $v$ . Hence,  $s, (r, l) := v_2$  is a redex for rule (18) (case 2.).

**Case**  $\text{region } r \text{ in } e$ . By assumption  $H \vdash s, \text{region } r \text{ in } e : \mu, \varphi$  with  $\varphi \subseteq \text{RegionNames}$ . By rules (*conf*) and (*userregion*), it must be that  $H, \{ \} \vdash e : \mu, \varphi'$  where  $\varphi' \subseteq \varphi \cup \{r\}$ . Since  $r \in \text{RegionNames}$ ,  $\varphi' \subseteq \text{RegionNames}$ . By rule (*conf*), it follows that  $H \vdash s, e : \mu, \varphi'$  where  $\varphi' \subseteq \text{RegionNames}$ .

By induction, either  $e$  is a value or it can be reduced. If  $e$  is a value then  $s, \text{region } r \text{ in } e$  is a redex for reduction rule (11) (case 2.).

If  $s, e \rightarrow_i s', e'$  then, by (19),  $s, \text{region } r \text{ in } e \rightarrow_i s', \text{region } r \text{ in } e'$ .

■

Let  $\rightarrow_i$  be the reflexive and transitive closure of small-step reduction  $\rightarrow_i$ . We can now state the type soundness theorem, which follows immediately from the previous results.

**THEOREM 4.1 (Type Soundness).**

Suppose for a term  $e$  that  $H \vdash s, e : \mu, \varphi, \text{frn}(\mu) \subseteq \text{dom}(H)$  and  $\varphi \subseteq \text{RegionNames}$ . Then, either

• there exist a value  $v$ , a heap type  $H'$  and a store  $s'$  such that  $s, e \rightarrow_i s', v$  and  $H' \vdash s', v : \mu, \emptyset$ ; or

• for each  $s', e'$  with  $s, e \rightarrow_i s', e'$ , there exist  $s'', e''$  such that  $s', e' \rightarrow_i s'', e''$ .

*Proof.* The proof is an induction on the number of reduction steps. We can repeatedly use proposition 4.1 and 4.2 since the condition  $\text{frn}(\mu) \subseteq \text{dom}(H)$  is preserved by reductions and the effect set only decreases. ■

## 5. EQUIVALENCES BETWEEN CALCULI

In this section, we relate the three different calculi TTRC, SRC and IRC to each other. For brevity, we consider pure IRC (PIRC), which excludes the operations on references and the `copy` operator. Moreover, due to the absence of a store, it is not possible to introduce destructive updates in SRC.

### 5.1. Equivalence between TTRC and PIRC

Superficially, the two store-based semantics have a lot in common. However, we are faced with the technical difficulty of relating a big-step operational semantics with a small-step operational semantics.

First, define the following product relations:

$$\begin{aligned} \text{QRel} &= \text{TTRC-Store} \times \text{TTRC-ValueEnv} \times \text{TTRC-RegionEnv} \times \\ &\quad \text{TTRC-Terms} \times \text{PIRC-Store} \times \text{PIRC-Terms} \\ \text{QRelV} &= \text{TTRC-Store} \times \text{TTRC-Values} \times \text{PIRC-Store} \times \text{PIRC-Values} \\ \text{QRelS} &= \text{TTRC-Store} \times \text{PIRC-Store} \end{aligned}$$

With the equivalences of figure 9, define the relation  $\mathcal{Q} \subseteq \text{QRel}$ , relating TTRC configurations to PIRC configurations, an auxiliary relation  $\mathcal{Q}_v \subseteq \text{QRelV}$ , relating TTRC values to PIRC values, and an auxiliary relation  $\mathcal{Q}_s \subseteq \text{QRelS}$  relating TTRC stores to PIRC stores. Observe that the auxiliary definitions for  $\mathcal{Q}_s$  and  $\mathcal{Q}_v$  can be eliminated by inserting them in the definition of  $\mathcal{Q}$ .

To show that the relation  $\mathcal{Q}$  is well-defined, interpret the equivalences from left to right as the defining clauses of a functional  $\mathcal{Q} : \mathcal{P}(\text{QRel}) \rightarrow \mathcal{P}(\text{QRel})$ . For example, in the case

$$\begin{aligned}
\mathcal{Q}_s(S, s) &\Leftrightarrow \text{dom}(S) = \text{dom}(s) \wedge (\forall r \in \text{dom}(S)) \text{dom}(S(r)) = \text{dom}(s(r)) \\
\mathcal{Q}_v(S, (r, l), s, v) &\Leftrightarrow (v \equiv (r, l) \wedge \mathcal{Q}_s(S, s) \wedge r \in \text{dom}(S) \wedge l \in \text{dom}(S(r)) \wedge \\
&\quad (S(r(l)) = \langle c \rangle = s(r(l)) \vee \\
&\quad (S(r(l)) = \langle x, e, VE, R \rangle \wedge s(r(l)) = \langle \lambda x. e' \rangle \wedge \\
&\quad \mathcal{Q}(S, VE - x, R, e, s, e')) \vee \\
&\quad (v \equiv (\bullet, l) \wedge \mathcal{Q}_s(S, s)) \\
\mathcal{Q}(S, VE, R, c \text{ at } \varrho, s, e') &\Leftrightarrow (e' \equiv c \text{ at } \varrho \wedge \mathcal{Q}_s(S, s) \wedge \varrho \notin \text{dom}(R)) \vee \\
&\quad (e' \equiv c \text{ at } r \wedge \mathcal{Q}_s(S, s) \wedge R(\varrho) = r) \vee \\
&\quad (e' \equiv c \text{ at } \bullet \wedge \mathcal{Q}_s(S, s)) \\
\mathcal{Q}(S, VE, R, x, s, e') &\Leftrightarrow (e' \equiv v \wedge \mathcal{Q}_v(S, VE(x), s, v)) \vee \\
&\quad (e' \equiv x \wedge \mathcal{Q}_s(S, s) \wedge x \notin \text{dom}(VE)) \\
\mathcal{Q}(S, VE, R, \lambda x. e \text{ at } \varrho, s, e') &\Leftrightarrow (e' \equiv (\lambda x. e'' \text{ at } \varrho) \wedge \mathcal{Q}_s(S, s) \wedge \varrho \notin \text{dom}(R) \wedge \\
&\quad \mathcal{Q}(S, VE - x, R, e, s, e'')) \vee \\
&\quad (e' \equiv (\lambda x. e'' \text{ at } r) \wedge \mathcal{Q}_s(S, s) \wedge R(\varrho) = r \wedge \\
&\quad \mathcal{Q}(S, VE - x, R, e, s, e'')) \vee \\
&\quad (e' \equiv (\lambda x. e'' \text{ at } \bullet) \wedge \mathcal{Q}_s(S, s)) \\
\mathcal{Q}(S, VE, R, e_1 @ e_2, s, e') &\Leftrightarrow e' \equiv (e'_1 @ e'_2) \wedge \mathcal{Q}_s(S, s) \wedge \mathcal{Q}(S, VE, R, e_1, s, e'_1) \wedge \\
&\quad \mathcal{Q}(S, VE, R, e_2, s, e'_2) \\
\mathcal{Q}(S, VE, R, \text{letregion } \varrho \text{ in } e, s, e') &\Leftrightarrow e' \equiv (\text{letregion } \varrho \text{ in } e'') \wedge \mathcal{Q}_s(S, s) \wedge \\
&\quad \mathcal{Q}(S, VE, R - \varrho, e, s, e'')
\end{aligned}$$

**FIG. 9.** Relation between TTRC and PIRC terms

of function application the definition is as follows:

$$\begin{aligned} \mathcal{Q}(\mathcal{Q}') = & \dots \\ & \cup \{(S, VE, R, e_1 @ e_2, s, e') \mid e' \equiv (e'_1 @ e'_2) \wedge \mathcal{Q}'_s(S, s) \wedge \\ & \qquad \qquad \qquad \mathcal{Q}'(S, VE, R, e_1, s, e'_1) \wedge \\ & \qquad \qquad \qquad \mathcal{Q}'(S, VE, R, e_2, s, e'_2)\} \\ & \cup \dots \end{aligned}$$

It is easy to verify that the functional  $\mathcal{Q}$  is monotone in the complete lattice  $(\mathcal{P}(\mathbf{QRel}), \subseteq)$ , so the existence of fixed points is guaranteed.

It remains to be decided which fixed point we are interested in. First, note that the empty relation is not a fixed point. In PIRC, a least fixed point would be enough, since it can be shown that  $\mathcal{Q}$  is well-founded, following a similar argument as in [4]. However, as soon as we consider full IRC with update-able references, it is generally possible to have cycles in the store. The well-foundedness argument breaks down and the least fixed point of  $\mathcal{Q}$  would exclude all programs that create or use a cycle in the store. This problem is easily avoided by defining  $\mathcal{Q}$  as the greatest fixed point of  $\mathcal{Q}$ .

The reader may find it alarming that we introduce greatest fixed points, and therefore possibly co-induction, one of the sources of complexity in the original Tofte-Talpin proof. First, note that the language considered by Tofte and Talpin does not include destructive updates and hence, the least fixed point would do. Secondly, our proofs hardly build on the structure of  $\mathcal{Q}$  itself. In fact, it turns out that only one lemma requires a simple co-induction.

Finally, we point out two peculiarities of  $\mathcal{Q}$ : the reader may have wondered that a TTRC store and a PIRC store are related by  $\mathcal{Q}_s$  as soon as the domains agree. This reflects the idea that values in the stores only need to correspond if a PIRC pointer refers to them. Also, the relation  $\mathcal{Q}$  does not relate intermediate PIRC terms, since those have no counterpart in TTRC.

Before formulating the equivalence theorem, we need several technical lemmas for the main proof.

The first lemma states that the relation is closed under extension of the stores and is proven by co-induction on the structure of  $\mathcal{Q}$ .

LEMMA 5.1. *Suppose  $\mathcal{Q}_s(S_2, s_2)$ ,  $S_1 \leq S_2$  and  $s_1 \leq s_2$  then*

1.  $\mathcal{Q}(S_1, VE, R, e, s_1, e')$  implies  $\mathcal{Q}(S_2, VE, R, e, s_2, e')$ ;
2.  $\mathcal{Q}_v(S_1, v, s_1, v')$  implies  $\mathcal{Q}_v(S_2, v, s_2, v')$ .

*Proof.* In order to prove the lemma by co-induction on  $\mathcal{Q}$ , we formulate the above property as a set  $X \subseteq \mathbf{QRel}$ :

$$\begin{aligned} X = \{ & (S, VE, R, e, s, e') \mid \mathcal{Q}_s(S, s) \wedge \exists S_1, s_1. \\ & S_1 \leq S \wedge s_1 \leq s \wedge \mathcal{Q}(S_1, VE, R, e, s_1, e')\} \end{aligned}$$

The co-induction principle states that if  $X \subseteq \mathcal{Q}(X)$ , then  $X \subseteq \mathcal{Q}$ ; the latter obviously proves the claim.

To show  $X \subseteq Q(X)$ , let us specify  $Q(X)$ . We only consider the case for function application:

$$\begin{aligned} Q(X) = & \dots \\ & \cup \{(S, VE, R, e_1 @ e_2, s, e') \mid e' \equiv (e'_1 @ e'_2) \wedge Q_s(S, s) \wedge \exists S_1, s_1. \\ & \quad S_1 \leq S \wedge s_1 \leq s \wedge \\ & \quad Q(S_1, VE, R, e_1, s_1, e'_1) \wedge \\ & \quad Q(S_1, VE, R, e_2, s_1, e'_2)\} \\ & \cup \dots \end{aligned}$$

Now, take  $(S, VE, R, e, s, e') \in X$ , then  $Q_s(S, s)$  and  $Q(S_1, VE, R, e, s_1, e')$ ,  $S_1 \leq S$ ,  $s_1 \leq s$  for some  $S_1$  and  $s_1$ . Again, we consider the case for function application, so  $e = e_1 @ e_2$ . Then, since  $Q(S_1, VE, R, e_1 @ e_2, s_1, e')$ , it must be that  $e' = e'_1 @ e'_2$ . Moreover,  $Q(S_1, VE, R, e_1, s_1, e'_1)$  and  $Q(S_1, VE, R, e_2, s_1, e'_2)$  hold, so, we can conclude that  $(S, VE, R, e, s, e') \in Q(X)$ . ■

Substitution of a value for a variable in PIRC is related to an extension of the type environment in TTRC.

LEMMA 5.2. *If  $Q(S, VE - x, R, e, s, e')$  and  $Q_v(S, v, s, v')$  then  $Q(S, VE + \{x \mapsto v\}, R, e, s, e'[x \mapsto v'])$ .*

*Proof.* By induction on  $e$ . The only interesting case is the one for variables.

**Case**  $e \equiv y$ . So  $Q(S, VE - x, R, y, s, e')$  and  $Q_v(S, v, s, v')$ .

**Subcase**  $x \neq y$  Then, either  $Q_v(S, (VE - x)(y), s, e')$  where  $e' = v''$ . Now, obviously we also have  $Q_v(S, (VE + \{x \mapsto v\})(y), s, v''[x \mapsto v'])$ , so  $Q(S, VE + \{x \mapsto v\}, R, y, s, e'[x \mapsto v'])$ .

Alternatively,  $Q(S, VE - x, R, y, s, y)$ , so we have that  $y \notin \text{dom}(VE - x)$ . Then, trivially we also have  $y \notin \text{dom}(VE + \{x \mapsto v\})$ . Hence,  $Q(S, VE + \{x \mapsto v\}, R, y, s, y[x \mapsto v'])$ .

**Subcase**  $x = y$  So,  $e' = x$  since  $x \notin \text{dom}(VE - x)$ . But  $Q_v(S, v, s, v')$ , therefore  $Q_v(S, (VE + \{x \mapsto v\})(y), s, v')$ . Hence,  $Q(S, VE + \{x \mapsto v\}, R, y, s, y[x \mapsto v'])$ .

■

The next lemma deals with region allocation.

LEMMA 5.3. *If  $Q(S, VE, R - \varrho, e, s, e')$  then  $Q(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\}\}, e'[\varrho \mapsto r])$*

*Proof.* By induction on  $e$ . Note that it is immediate that if  $Q_s(S, s)$ , also  $Q_s(S + \{r \mapsto \{\}\}, s + \{r \mapsto \{\}\})$ . We will use this fact implicitly.

**Case**  $x$ . By definition of  $Q(S, VE, R - \varrho, x, s, e')$ , there are two cases:

**Subcase**  $x \notin \text{dom}(VE)$ . In this case,  $e' = x$  and  $Q_s(S, s)$ . Since  $VE$  does not change, the claim follows immediately.

**Subcase**  $VE(x) = v$ . Hence,  $Q_v(S, v, s, v')$ , for some  $v'$  where  $e' = v'$ . Now, the claim holds independent of  $R$ .

**Case**  $c \text{ at } \varrho'$ . By definition of  $\mathcal{Q}(S, VE, R - \varrho, c \text{ at } \varrho', s, e')$ , there are three cases:

**Subcase**  $e' \equiv c \text{ at } \varrho'$ . Hence  $\varrho' \notin \text{dom}(R - \varrho)$ . To see that  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, c \text{ at } \varrho', s + \{r \mapsto \{\}\}, c \text{ at } \varrho'[\varrho \mapsto r])$ , there are two cases to consider. If  $\varrho = \varrho'$  then the related term becomes  $c \text{ at } r$  and the claim holds since  $(R + \{\varrho \mapsto r\})(\varrho) = r$ .

If  $\varrho \neq \varrho'$  then the related term is  $c \text{ at } \varrho'$  and the claim holds since  $\varrho' \notin \text{dom}(R' + \{\varrho \mapsto r\})$ .

**Subcase**  $e' \equiv c \text{ at } r'$ . Hence,  $(R' - \varrho)(\varrho') = r'$ . Therefore  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, c \text{ at } \varrho', s + \{r \mapsto \{\}\}, c \text{ at } r')$ .

**Subcase**  $e' \equiv c \text{ at } \bullet$ . In this case,  $e'$  remains unchanged under the substitution, so that  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, c \text{ at } \varrho', s + \{r \mapsto \{\}\}, c \text{ at } \bullet)$  is immediate.

**Case**  $\lambda x. e \text{ at } \varrho'$ . Analogous to the previous case, plus appealing to the induction hypothesis.

**Case**  $e_1 @ e_2$ . Immediate by appealing to the induction hypothesis.

**Case**  $\text{letregion } \varrho' \text{ in } e$ . Since  $\mathcal{Q}(S, VE, R - \varrho, \text{letregion } \varrho' \text{ in } e, s, e')$ , it must be that  $e' = \text{letregion } \varrho' \text{ in } e''$  and  $\mathcal{Q}(S, VE, R - \varrho - \varrho', e, s, e'')$ .

We can assume w.l.o.g. that  $\varrho \neq \varrho'$ . Since  $R - \varrho - \varrho' = R - \varrho' - \varrho$ , it holds that  $\mathcal{Q}(S, VE, R - \varrho' - \varrho, e, s, e'')$ . By induction,  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R - \varrho' + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\}\}, e''[\varrho \mapsto r])$ . Since  $R - \varrho' + \{\varrho \mapsto r\} = R + \{\varrho \mapsto r\} - \varrho'$  applying the definition of  $\mathcal{Q}$  yields that  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, \text{letregion } \varrho' \text{ in } e, s + \{r \mapsto \{\}\}, \text{letregion } \varrho' \text{ in } e''[\varrho \mapsto r])$ , which proves the claim.

■

As a final lemma, we prove that region deallocation does not influence the equivalence relation.

LEMMA 5.4. *The following hold:*

1. If  $\mathcal{Q}(S, VE, R, e, s', e')$  then  $\mathcal{Q}(S - r, VE, R, e, s'[r \mapsto \bullet] - r, e'[r \mapsto \bullet])$ .
2. If  $\mathcal{Q}_v(S, v, s', v')$  then  $\mathcal{Q}_v(S - r, v, s'[r \mapsto \bullet] - r, v'[r \mapsto \bullet])$ .

*Proof.* Again, we can make an immediate observation on deallocation in two related stores and use this fact throughout the proof: if  $\mathcal{Q}_s(S, s')$ , then  $\mathcal{Q}_s(S - r, s'[r \mapsto \bullet] - r)$ .

**Item 1.** By induction on  $e$ .

**Case**  $x$ . Since  $\mathcal{Q}(S, VE, R, x, s', e')$ , there are two cases:

**Subcase**  $VE(x) = v$ . So, it must be that  $e' = v'$  with  $\mathcal{Q}_v(S, v, s', v')$ . We have two cases. If  $v' = (\bullet, l)$  then the claim holds immediately. Suppose  $v' = (r', l)$ . If  $r' \neq r$  then the claim holds. If  $r' = r$  then  $v'[r \mapsto \bullet] = (\bullet, l)$  and the claim holds, too.

**Subcase**  $x \notin \text{dom}(VE)$ . Immediate.

**Case**  $c \text{ at } \varrho$ . Since  $\mathcal{Q}(S, VE, R, c \text{ at } \varrho, s', e')$ , there are three cases.

**Subcase**  $e' \equiv c \text{ at } \varrho$ . So,  $\varrho \notin \text{dom}(R)$  and  $\mathcal{Q}_s(S, s')$ .

Since  $(c \text{ at } \varrho)[r \mapsto \bullet] = c \text{ at } \varrho$ , it is immediate that  $\mathcal{Q}(S - r, VE, R, c \text{ at } \varrho, s'[r \mapsto \bullet] - r, c \text{ at } \varrho)$ .

**Subcase**  $e' \equiv c \text{ at } r'$ . So,  $R(\varrho) = r'$  where  $r' \in \text{dom}(S)$  and  $\mathcal{Q}_s(S, s')$ .

If  $r' \neq r$  then the claim holds. If  $r' = r$  then  $(c \text{ at } r')[r \mapsto \bullet] = c \text{ at } \bullet$  and the claim holds, too.

**Subcase**  $e' \equiv c$  at  $\bullet$ . Immediate.

**Case**  $\lambda x. e$  at  $\varrho$ . Analogous to case  $c$  at  $\varrho$ , applying the induction hypothesis to obtain the result for  $e$ .

**Case**  $e_1 @ e_2$ . Simply appeals to the induction hypothesis for  $e_1$  and  $e_2$ .

**Case** `letregion  $\varrho$  in  $e$` . Since  $\mathcal{Q}(S, VE, R, \text{letregion } \varrho \text{ in } e, s', e')$  it must be that  $e' = \text{letregion } \varrho \text{ in } e''$  and  $\mathcal{Q}(S, VE, R - \varrho, e, s', e'')$ . By induction  $\mathcal{Q}(S - r, VE, R - \varrho, e, s'[r \mapsto \bullet] - r, e''[r \mapsto \bullet])$ , so  $\mathcal{Q}(S - r, VE, R, \text{letregion } \varrho \text{ in } e, s'[r \mapsto \bullet] - r, \text{letregion } \varrho \text{ in } e''[r \mapsto \bullet])$ .

**Item 2.** By assumption we have  $\mathcal{Q}_v(S, (r', l), s', v')$ . If  $v' = (\bullet, l)$ , the claim is immediate. Otherwise we have that  $v' = (r', l)$ . If  $r \neq r'$ , then  $(r', l)[r \mapsto \bullet] = (r', l)$ , so we have that  $\mathcal{Q}_v(S - r, (r', l), s'[r \mapsto \bullet] - r, (r', l))$ . On the other hand, if  $r = r'$  then  $(r', l)[r \mapsto \bullet] = (\bullet, l)$ , so that  $\mathcal{Q}_v(S - r, (r', l), s'[r \mapsto \bullet] - r, (\bullet, l))$  holds.

■

The main equivalence theorem requires the notion of a *stuck* term.

DEFINITION 5.1.

1. An IRC configuration  $s, e$  is *stuck* if  $e$  is not a value and there do not exist  $s', e'$  such that  $s, e \rightarrow_i s', e'$ .
2. An IRC configuration  $s, e$  *becomes stuck* iff  $s, e \rightarrow_i s', e'$  and  $s', e'$  is stuck.

Now, we can formulate the equivalence theorem:

THEOREM 5.1. *The following hold:*

1. Suppose that  $S, VE, R \vdash_b e \Downarrow v, S'$ . For each  $s, e'$  such that  $\mathcal{Q}(S, VE, R, e, s, e')$ , either  $s, e'$  becomes stuck or there exist  $s', v'$  so that  $s, e' \rightarrow_i s', v'$  and  $\mathcal{Q}_v(S', v, s', v')$ .
2. If  $s, e' \rightarrow_i s', v'$  and  $\mathcal{Q}(S, VE, R, e, s, e')$ , then  $S, VE, R \vdash_b e \Downarrow v, S'$  for which  $\mathcal{Q}_v(S', v, s', v')$ .

Part 1. of theorem 5.1 is weaker than expected. However, the apparent weakness is due to the fact that the theorem deals with an untyped semantics. The problem is that, for some un-typeable expressions, the TTRC semantics is slightly less deterministic than the PIRC semantics. Consider the following example:

```
letregion  $\varrho$  in
  let  $f = \text{letregion } \varrho_1 \text{ in } \lambda x. (5 \text{ at } \varrho_1) \text{ at } \varrho \text{ in}$ 
    letregion  $\varrho_2 \text{ in } f @ (3 \text{ at } \varrho_2)$ 
```

This term definitely gets stuck in PIRC since the region allocated for  $\varrho_1$  in function  $f$  is substituted for  $\bullet$  after leaving the binding for  $f$ . In TTRC, on the other hand, this term may still evaluate to 5 if the region allocated to  $\varrho_1$  in the definition of  $f$  is re-allocated to  $\varrho_2$  in the last line.

Below we will show that this subtle difference between the two semantics becomes irrelevant for well-typed terms. First, we proceed with the proof of theorem 5.1.

*Proof.* (**part 1. of theorem 5.1**) By induction on the derivation of  $S, VE, R \vdash_b e \Downarrow v, S'$ .

$R(\varrho) = r \quad l \notin \text{dom}(S(r))$   
**Case**  $\frac{S, VE, R \vdash_b c \text{ at } \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto \langle c \rangle\}\}}{Q(S, VE, R, c \text{ at } \varrho, s, e')}$ . Let  $s, e'$  be such that  $Q(S, VE, R, c \text{ at } \varrho, s, e')$ . Then there are two cases:

**Subcase**  $e' \equiv c \text{ at } r$ . Then  $Q_s(S, s)$  where  $l \notin \text{dom}(s(r))$ . As a consequence,  $s, c \text{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l)$ . It also holds that  $Q_v(S + \{r \mapsto S(r) + \{l \mapsto \langle c \rangle\}\}, (r, l), s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l))$ .

**Subcase**  $e' \equiv c \text{ at } \bullet$ . This term is stuck.

$VE(x) = v$   
**Case**  $\frac{S, VE, R \vdash_b x \Downarrow v, S}{e' = v' \text{ where } Q_v(S, v, s, v'), \text{ which proves the claim.}}$

$R(\varrho) = r \quad l \notin \text{dom}(S(r))$   
**Case**  $\frac{S, VE, R \vdash_b \lambda x. e \text{ at } \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto \langle x, e, VE, R \rangle\}}}{\text{Let } s, e'' \text{ be such that } Q(S, VE, R, \lambda x. e \text{ at } \varrho, s, e''). \text{ There are two cases.}}$

**Subcase**  $e'' \equiv \lambda x. e' \text{ at } r$ . Then  $Q(S, VE - x, R', e, s, e')$  and  $Q_s(S, s)$ . Hence,  $l \notin \text{dom}(s(r))$  so that  $s, \lambda x. e' \text{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda x. e \rangle\}\}, (r, l)$ . From the assumptions it is immediate that  $Q_v(S + \{r \mapsto S(r) + \{l \mapsto \langle x, e, VE, R \rangle\}\}, (r, l), s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda x. e \rangle\}\}, (r, l))$ .

**Subcase**  $e'' \equiv \lambda x. e' \text{ at } \bullet$ . This term is stuck.

$S, VE, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1 \quad S_1(r_1)(l_1) = \langle x, e, VE', R' \rangle$   
**Case**  $\frac{S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2 \quad S_2, VE' + \{x \mapsto v_2\}, R' \vdash_b e \Downarrow v, S'}{S, VE, R \vdash_b e_1 @ e_2 \Downarrow v, S'}$

Let  $s, e''$  be such that  $Q(S, VE, R, e_1 @ e_2, s, e'')$ . Hence,  $Q_s(S, s)$ ,  $Q(S, VE, R, e_1, s, e'_1)$  and  $Q(S, VE, R, e_2, s, e'_2)$  where  $e'' = e'_1 @ e'_2$ .

By induction on  $S, VE, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1$  either  $s, e'_1$  becomes stuck (in which case  $s, e'_1 @ e'_2$  becomes also stuck) or there exist  $s'_1, v'_1$  such that  $s, e'_1 \rightarrow_i s'_1, v'_1$  and  $Q_v(S_1, (r_1, l_1), s'_1, v'_1)$ .

By induction on  $S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2$  (using lemmas 2.1, 4.7 and 5.1 to establish  $Q$  since  $Q_s(S_1, s'_1)$ ) either  $s'_1, e'_2$  becomes stuck (in which case  $s'_1, v'_1 @ e'_2$  becomes also stuck) or there exist  $s'_2, v'_2$  such that  $s'_1, e'_2 \rightarrow_i s'_2, v'_2$  and  $Q_v(S_2, v_2, s'_2, v'_2)$ .

From the definition of  $Q_v(S_1, (r_1, l_1), s_1, v'_1)$ , there are two cases for  $v'_1$ . Either  $v'_1 = (\bullet, l_1)$  and then the IRC configuration  $s'_2, v'_1 @ v'_2$  is stuck; or  $v'_1 = (r_1, l_1)$  and since  $S_1(r_1)(l_1) = \langle x, e, VE', R' \rangle$  it must be that  $s'_1(r_1)(l_1) = \lambda x. e'$  where  $Q(S_1, VE' - x, R', e, s'_1, e')$ . Hence,  $s'_2, v'_1 @ v'_2 \rightarrow_i s'_2, e'[x \mapsto v'_2]$ . By lemmas 2.1, 4.7 and 5.1, it holds that  $Q(S_2, VE' - x, R', e, s'_2, e')$  (since  $Q_s(S_2, s'_2)$ ). Lemma 5.2 then states that  $Q(S_2, VE' + \{x \mapsto v_2\}, R', e, s'_2, e'[x \mapsto v'_2])$ .

By induction, we obtain that either  $s'_2, e'[x \mapsto v'_2]$  becomes stuck (in which case  $s'_2, v'_1 @ v'_2$  becomes also stuck) or there exists some  $s', v'$  such that  $s'_2, e'[x \mapsto v'_2] \rightarrow_i s', v'$  and  $Q_v(S', v, s', v')$ .

Putting the parts together using reduction rules (14), (20) and (21), either  $s, e'_1 @ e'_2$  becomes stuck or  $s, e'_1 @ e'_2 \rightarrow_i s', v'$  where  $Q_v(S', v, s', v')$ .

**Case**  $\frac{r \notin \text{dom}(S) \quad S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\} \vdash_b e \Downarrow v, S'}{S, VE, R \vdash_b \text{letregion } \varrho \text{ in } e \Downarrow v, S' - r}$ . Let  $s, e''$  be such

that  $\mathcal{Q}(S, VE, R, \text{letregion } \varrho \text{ in } e, s, e'')$ . Hence we have that  $\mathcal{Q}_s(S, s)$ , that  $e'' = \text{letregion } \varrho \text{ in } e'$  and also that  $\mathcal{Q}(S, VE, R - \varrho, e, s, e')$ . Therefore,  $s, \text{letregion } \varrho \text{ in } e' \rightarrow_i s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e'[\varrho \mapsto r]$ . By Lemma 5.3, it holds that  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\}\}, e'[\varrho \mapsto r])$ .

By induction, either  $s + \{r \mapsto \{\}\}, e'[\varrho \mapsto r]$  becomes stuck (in which case  $s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e'[\varrho \mapsto r]$  becomes stuck, too) or there exist  $s', v'$  such that  $s + \{r \mapsto \{\}\}, e'[\varrho \mapsto r] \rightarrow_i s', v'$  and  $\mathcal{Q}_v(S', v, s', v')$ . Because of reduction rule (19), we have  $s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e'[\varrho \mapsto r] \rightarrow_i s', \text{region } r \text{ in } v'$ . But the last configuration is a redex (rule (11)):  $s', \text{region } r \text{ in } v' \rightarrow_i s'[r \mapsto \bullet] - r, v'[r \mapsto \bullet]$ . By Lemma 5.4, it follows that  $\mathcal{Q}_v(S' - r, v, s'[r \mapsto \bullet] - r, v'[r \mapsto \bullet])$ , which concludes the proof. ■

*Proof.* (**part 2. of theorem 5.1**) By induction on the number of steps in  $s_1, e' \rightarrow_i s_2, v'$ . We only consider two interesting cases:

**Case**  $s, \text{letregion } \varrho \text{ in } e' \rightarrow_i s + \{r \mapsto \{\}\}, \text{region } r \text{ in } e[\varrho \mapsto r] \rightarrow_i s', v'$  where  $r \notin \text{dom}(s)$ . Then  $s + \{r \mapsto \{\}\}, e[\varrho \mapsto r] \rightarrow_i s_1, v'_1$  where  $s_1[r \mapsto \bullet] - r = s'$  and  $v'_1[r \mapsto \bullet] = v'$ , so that  $s_1, \text{region } r \text{ in } v'_1 \rightarrow_i s', v'$  is the last step.

Suppose now  $\mathcal{Q}(S, VE, R, \text{letregion } \varrho \text{ in } e, s, \text{letregion } \varrho \text{ in } e')$ , then lemma 5.3 implies  $\mathcal{Q}(S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\}\}, e'[\varrho \mapsto r])$ .

By induction hypothesis,  $S + \{r \mapsto \{\}\}, VE, R + \{\varrho \mapsto r\} \vdash_b e \Downarrow v_1, S_1$  for which  $\mathcal{Q}_v(S_1, v_1, s_1, v'_1)$ . Hence, by rule (*b-letregion*)  $S, VE, R \vdash_b \text{letregion } \varrho \text{ in } e \Downarrow v_1, S_1 - r$ , and  $\mathcal{Q}_v(S_1 - r, v_1, s_1, v')$  follows from lemma 5.4.

**Case**  $s, e'_1 @ e'_2 \rightarrow_i s', v'$ . Then  $s, e'_1 @ e'_2 \rightarrow_i s_1, v'_1 @ e'_2 \rightarrow_i s_2, v'_1 @ v'_2 \rightarrow_i s_2, e'[x \mapsto v'_2] \rightarrow_i s', v'$ , where  $v'_1 = (r, l)$  and  $s_2(r)(l) = \langle \lambda x. e' \rangle$  and  $s, e'_1 \rightarrow_i s_1, v'_1$  and  $s_1, e'_2 \rightarrow_i s_2, v'_2$ . Note that all the intermediate  $\rightarrow_i$  have strictly fewer steps than  $s, e'_1 @ e'_2 \rightarrow_i s', v'$ .

Suppose now that  $\mathcal{Q}(S, VE, R, e_1 @ e_2, s, e'_1 @ e'_2)$ , then  $\mathcal{Q}(S, VE, R, e_i, s, e'_i)$  for  $i = 1, 2$ . By the induction hypothesis,  $S, VE, R \vdash_b e_1 \Downarrow v_1, S_1$  for which  $\mathcal{Q}_v(S_1, v_1, s_1, v'_1)$ .

Since  $S \leq S_1$  and  $s \leq s_1$  (by lemmas 2.1 and 4.7) and because  $\mathcal{Q}_s(S_1, s_1)$ , lemma 5.1 gives  $\mathcal{Q}(S_1, VE, R, e_2, s_1, e'_2)$ . By the induction hypothesis,  $S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2$  for which  $\mathcal{Q}_v(S_2, v_2, s_2, v'_2)$ .

So, again by lemma 5.1, we have that  $\mathcal{Q}_v(S_2, v_1, s_2, v'_1)$ . Therefore  $S_2(r)(l) = \langle x, e, VE', R' \rangle$  with  $\mathcal{Q}(S_2, VE' - x, R', e, s_2, e')$ . Lemma 5.2 then implies  $\mathcal{Q}(S_2, VE' + \{x \mapsto v_2\}, R', e, s_2, e'[x \mapsto v'_2])$ . By the induction hypothesis, we derive that  $S_2, VE' + \{x \mapsto v_2\}, R' \vdash_b e \Downarrow v, S'$  for which  $\mathcal{Q}_v(S', v, s', v')$ . Finally, application of reduction rule (*b-app*) yields  $S, VE, R \vdash_b e_1 @ e_2 \Downarrow v, S'$ . ■

Before we refine theorem 5.1 to typed terms, we need to formalize how we *translate* typed TTRC terms to IRC-terms.

DEFINITION 5.2.

1. We say that  $(S, R, e, \mu, \varphi)$  is a *typed start configuration* for a TTRC program  $e$  iff  $\{\} \vdash_{tt} e : \mu, \varphi$  and  $S = \{r \mapsto \{\} \mid r \in \text{ran}(R)\}$ .

2. The tuple  $(S, R, e, \mu, \varphi, s, e')$  is a *typed translation* iff  $(S, R, e, \mu, \varphi)$  is a typed start configuration,  $e' = R(e)$  and  $s = \{r \mapsto \{ \} \mid r \in \text{ran}(R)\}$ .

We have the following property for typed translations:

LEMMA 5.5. *If  $(S, R, e, \mu, \varphi, s, e')$  is a typed translation, then  $\mathcal{Q}(S, \{ \}, R, e, s, e')$ .*

The following theorem is the typed variant of theorem 5.1:

THEOREM 5.2. *Suppose a typed translation  $(S, R, e, \mu, \varphi, s, e')$ . Then*

1. *If  $S, \square, R \vdash_b e \Downarrow v, S'$  then there exist  $v'$  and  $s'$  such that  $s, e' \rightarrow_i s', v'$  where  $\mathcal{Q}_v(S', v, s', v')$ .*

2. *If  $s, e' \rightarrow_i s', v'$  then there exist  $v$  and  $S'$  such that  $S, \square, R \vdash_b e \Downarrow v, S'$  where  $\mathcal{Q}_v(S', v, s', v')$ .*

*Proof.* Part 2. is a special case of theorem 5.1, part 2.

For part 1., take a heap type  $H = \{r \mapsto \{ \} \mid r \in \text{ran}(R)\}$ . Now, obviously,  $H \vdash s, e : \mu, \varphi$  and therefore by lemma 4.5 we have  $R(H) \vdash R(s), R(e) : R(\mu), R(\varphi)$ . Hence,  $H \vdash s, e' : R(\mu), R(\varphi)$  where  $R(\varphi) \subseteq \text{RegionNames}$  and  $\text{fn}(R(\mu)) \subseteq \text{ran}(R)$  ( $= \text{dom}(H)$ ). So, we can use theorem 4.1 to conclude that the configuration  $s, e'$  cannot become stuck. Because of lemma 5.5 we have  $\mathcal{Q}(S, \{ \}, R, e, s, e')$  and so by theorem 5.1 we know that there exist  $v'$  and  $s'$  such that  $s, e' \rightarrow_i s', v'$  and  $\mathcal{Q}_v(S', v, s', v')$ . ■

We conclude this section by proving type soundness for TTRC. Since there is no notion of a stuck term in a big-step semantics, we extend the TTRC evaluation relation with error reductions, which can be found in figure 10. The canonical rules for error propagation are omitted.

The following lemma relates error reductions in TTRC with stuck terms in IRC.

LEMMA 5.6. *If  $\mathcal{Q}(S, VE, R, e, s, e')$  and  $S, VE, R \vdash_b e \Downarrow \text{err}$  then  $s, e'$  becomes stuck.*

*Proof.* By induction on the evaluation of  $S, VE, R \vdash_b e \Downarrow \text{err}$ . All error propagation cases are straightforward applications of the induction hypothesis. The base cases are easily shown using theorem 5.1, we illustrate this with the case for application:

*Case  $S, VE, R \vdash_b e_1 @ e_2 \Downarrow \text{err}$*

So  $S, VE, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1$ . Since  $\mathcal{Q}(S, VE, R, e_1, s, e'_1)$  and using theorem 5.1 we have either that  $s, e'_1$  is stuck from which we also have that  $s, e'_1 @ e'_2$  is stuck following reduction rule (20).

Alternatively, there exist  $s_1, v'_1$  such that  $s, e'_1 \rightarrow_i s_1, v'_1$  where  $\mathcal{Q}_v(S_1, (r_1, l_1), s_1, v'_1)$ . We also have  $S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2$  and  $\mathcal{Q}(S, VE, R, e_2, s, e'_2)$ . By lemmas 2.1, 4.7 and 5.1, we have that  $\mathcal{Q}(S_1, VE, R, e_2, s_1, e'_2)$ . Again, we can apply theorem 5.1 and have either that:

- $s_1, e'_2$  becomes stuck and therefore, by reduction rules (20) and (21), that  $s, e'_1 @ e'_2$  also becomes stuck; or

$$\begin{array}{c}
(b\text{-const-err}) \quad \frac{\varrho \notin \text{dom}(R)}{S, VE, R \vdash_b c \text{ at } \varrho \Downarrow \text{err}} \\
\\
(b\text{-var-err}) \quad \frac{x \notin \text{dom}(VE)}{S, VE, R \vdash_b x \Downarrow \text{err}} \\
\\
(b\text{-abstr-err}) \quad \frac{\varrho \notin \text{dom}(R)}{S, VE, R \vdash_b \lambda x. e \text{ at } \varrho \Downarrow \text{err}} \\
\\
(b\text{-app-err}) \quad \frac{S, VE, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1 \wedge S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2 \wedge (r_1 \notin \text{dom}(S_2) \vee S_2(r_1)(l_1) \neq \langle x, e, VE', R' \rangle)}{S, VE, R \vdash_b e_1 @ e_2 \Downarrow \text{err}} \\
\\
(b\text{-copy-err}) \quad \frac{\{\varrho, \varrho'\} \not\subseteq \text{dom}(R) \vee (S, VE, R \vdash_b e \Downarrow (r, l), S' \wedge (r \notin \text{dom}(S') \vee S'(r)(l) \neq \langle c \rangle))}{S, VE, R \vdash_b \text{copy}[\varrho, \varrho'] e \Downarrow \text{err}}
\end{array}$$

FIG. 10. Extension of TTRC big-step evaluation with errors

• there exists a configuration  $s_2, v'_2$  such that  $s_1, e_2 \rightarrow_i s_2, v'_2$  with  $\mathcal{Q}_v(S_2, v_2, s_2, v'_2)$ . Again by lemmas 2.1, 4.7 and 5.1 we have that  $\mathcal{Q}_v(S_2, (r_1, l_1), s_2, v'_1)$ .

Following reduction (*b-app-err*), we have now two cases:

– Case  $r_1 \notin \text{dom}(S_2)$ . Then  $v'_1 = (\bullet, l_1)$  and therefore, by rules (14), (20) and (21), the configuration  $s, e'_1 @ e'_2$  becomes stuck.

– Case  $S_2(r_1)(l_1) \neq \langle x, e, VE', R' \rangle$ . Then, if  $v'_1 = (\bullet, l_1)$ , see previous case. Otherwise,  $S_2(r_1)(l_1) = \langle c \rangle = s_2(r_1)(l_1)$ . Hence, again by rules (14), (20) and (21),  $s, e'_1 @ e'_2$  becomes stuck.

■

**THEOREM 5.3 (Type Soundness of TTRC).** *If  $(S, R, e, \mu, \varphi)$  is a typed start configuration, then  $S, \{ \}, R \vdash_b e \Downarrow \text{err}$ .*

*Proof.* Take  $s$  and  $e'$  such that  $(S, R, e, \mu, \varphi, s, e')$  is a typed translation. By analogous reasoning as in the proof for theorem 5.2, we know that  $s, e'$  does not become stuck. But since  $\mathcal{Q}(S, \{ \}, R, e, s, e')$  by lemma 5.5, we can conclude that  $S, \{ \}, R \vdash_b e \Downarrow \text{err}$  as a consequence of lemma 5.6. ■

Note that this soundness result is slightly stronger than the original theorem of Tofte and Talpin [14] since it proves soundness for terminating as well as non-terminating programs.

$$\begin{array}{lcl}
\mathcal{R}(s, (r, l), e', \alpha) & \Leftrightarrow & (e' \equiv \langle c \rangle_{\varrho} \wedge s(r)(l) = \langle c \rangle \wedge \alpha(\varrho) = r) \vee \\
& & (e' \equiv \langle \lambda x. e'' \rangle_{\varrho} \wedge s(r)(l) = \langle \lambda x. e \rangle \wedge \\
& & \mathcal{R}(s, e, e'', \alpha) \wedge \alpha(\varrho) = r) \\
\mathcal{R}(s, (\bullet, l), e', \alpha) & \Leftrightarrow & (e' \equiv \langle c \rangle_{\bullet}) \vee (e' \equiv \langle \lambda x. e'' \rangle_{\bullet}) \\
\mathcal{R}(s, c \text{ at } r, e', \alpha) & \Leftrightarrow & e' \equiv c \text{ at } \varrho \wedge \alpha(\varrho) = r \\
\mathcal{R}(s, c \text{ at } \bullet, e', \alpha) & \Leftrightarrow & e' \equiv c \text{ at } \bullet \\
\mathcal{R}(s, \lambda x. e \text{ at } r, e', \alpha) & \Leftrightarrow & e' \equiv \lambda x. e'' \text{ at } \varrho \wedge \mathcal{R}(s, e, e'', \alpha) \wedge \alpha(\varrho) = r \\
\mathcal{R}(s, \lambda x. e \text{ at } \bullet, e', \alpha) & \Leftrightarrow & e' \equiv \lambda x. e'' \text{ at } \bullet \\
\mathcal{R}(s, x, e', \alpha) & \Leftrightarrow & e' \equiv x \\
\mathcal{R}(s, e_1 @ e_2, e', \alpha) & \Leftrightarrow & e' \equiv e'_1 @ e'_2 \wedge \mathcal{R}(s, e_1, e'_1, \alpha) \wedge \mathcal{R}(s, e_2, e'_2, \alpha) \\
\mathcal{R}(s, \text{letregion } \varrho \text{ in } e, e', \alpha) & \Leftrightarrow & e' \equiv \text{letregion } \varrho \text{ in } e'' \wedge r \notin \text{dom}(s) \wedge \\
& & \mathcal{R}(s + \{r \mapsto \{\}\}, e[\varrho \mapsto r], e'', \alpha + \{\varrho \mapsto r\}) \\
\mathcal{R}(s, \text{region } r \text{ in } e, e', \alpha) & \Leftrightarrow & e' \equiv \text{letregion } \varrho \text{ in } e'' \wedge \alpha(\varrho) = r \wedge \\
& & \mathcal{R}(s, e, e'', \alpha)
\end{array}$$

FIG. 11. Relation between PIRC and SRC terms

## 5.2. Equivalence between PIRC and SRC

Proving equivalence between PIRC and SRC is considerably simpler because we are dealing with two small-step semantics. The proof boils down to a simple induction on the two transition relations.

The most important difference between PIRC and SRC is the handling of region constants. Region variables play a dual role in SRC. On the one hand, they are alpha-convertible to avoid conflicting uses, but on the other hand, they substitute for region names and thus turn up in addresses (values). In contrast, the region names in PIRC are bound by the store which guarantees that a region name is not in use at the point where a new region is created. In the relation  $\mathcal{R}$ , we cater for this apparent mismatch with an explicit region environment that maps region variables (in SRC) to region names (in PIRC).

We relate PIRC configurations and SRC terms via a relation

$$\mathcal{R} \subseteq \text{PIRC-Store} \times \text{PIRC-Terms} \times \text{SRC-Terms} \times (\text{RegionVars} \rightarrow \text{RegionNames})$$

defined by the equivalences in figure 11. The map  $\alpha \in \text{RegionVars} \rightarrow \text{RegionNames}$  relates allocated region variables in SRC with actual regions in PIRC. In all equivalences, we implicitly assume that  $\alpha$  is injective and  $\text{dom}(s) = \text{ran}(\alpha)$ .

In contrast to the relation  $\mathcal{Q}$  of section 5.1, we can define  $\mathcal{R}$  as the least fixed point of the associated functional. In fact, it is easy to see that  $\mathcal{R}$  is well-founded and that cycles are a non-issue since it is not possible to create them in SRC.

The relation  $\mathcal{R}$  is trivially non-empty. Moreover, we have the following proposition:

PROPOSITION 5.1. *If  $e$  is a pure PIRC-term,  $\alpha$  an injective region environment where  $\text{dom}(\alpha) = \text{frv}(e)$  and  $s = \{r \mapsto \{\} \mid r \in \text{ran}(\alpha)\}$ , then  $\mathcal{R}(\alpha(s), \alpha(e), e, \alpha)$ .*

Before we can state the equivalence theorem, we need a few lemmas which are all proven by simple inductions:

LEMMA 5.7. *Suppose  $\mathcal{R}(s, e, e', \alpha + \{\varrho \mapsto r\})$ .*

Then  $\mathcal{R}(s[r \mapsto \bullet] - r, e[r \mapsto \bullet], e'[q \mapsto \bullet], \alpha)$ .

LEMMA 5.8. *Suppose  $\mathcal{R}(s, e, e', \alpha)$ ,  $q \notin \text{dom}(\alpha)$  and  $r \notin \text{ran}(\alpha)$ . Then  $\mathcal{R}(s, e, e', \alpha + \{q \mapsto r\})$ .*

LEMMA 5.9. *Suppose  $\mathcal{R}(s, e_1, e'_1, \alpha)$  and  $\mathcal{R}(s, e_2, e'_2, \alpha)$ . Then  $\mathcal{R}(s, e_1[x \mapsto e_2], e'_1[x \mapsto e'_2], \alpha)$ .*

We write  $\rightarrow_i^{1,2}$  to mean one or two reduction steps. The equivalence between SRC and PIRC is a result of the following theorem:

THEOREM 5.4. *Suppose  $\mathcal{R}(s_1, e_1, e'_1, \alpha)$ .*

1. *If  $s_1, e_1 \rightarrow_i s_2, e_2$  then there exist  $e'_2$  and  $\alpha'$  such that  $\mathcal{R}(s_2, e_2, e'_2, \alpha')$  and either  $e'_1 \rightarrow_s e'_2$  or  $e'_1 = e'_2$ .*

2. *If  $e'_1 \rightarrow_s e'_2$  then there exist  $s_2, e_2, \alpha'$  such that  $s_1, e_1 \rightarrow_i^{1,2} s_2, e_2$  and  $\mathcal{R}(s_2, e_2, e'_2, \alpha')$ .*

*Proof.* The proof is a case analysis on the definition of  $\rightarrow_i$  and  $\rightarrow_s$ , making use of lemmas 5.7, 5.8 and 5.9. ■

## 6. CONCLUSIONS

We have considered three different operational semantics of “the region calculus”: the original evaluation-style formulation [15], a store-less transition semantics [9], and a novel store-based variation, which extends the other two with operations on references. We have proven type soundness for both the small-step semantics with respect to the original region and effect system, by using the standard syntactic approach [16, 7]. The resulting proofs are simple inductions, considerably easier than the co-inductive formulation of Tofte and Talpin. Additionally, we have shown that all formulations are essentially equivalent in their typed subsets. As a direct consequence, we have given an alternative type soundness proof for the original region calculus semantics.

The store-less big-step semantics by Calcagno [4] is very similar to our store-less formulation. His high-level semantics is parameterized by a set  $\varphi$  of currently allocated regions. This is essentially equivalent to using a special constant  $\bullet$  for dead regions: it is possible to relate the two semantics by renaming the regions outside  $\varphi$  to  $\bullet$ . However, this clear relation is not maintained when moving to a semantics with an explicit store.

The approach with  $\bullet$  has the appealing property that all dangling pointers are explicitly marked with  $\bullet$ . As a consequence, IRC enables to cleanly and simply reason about *re-use* of a region after deallocating it. In contrast, an approach like TTRC without  $\bullet$  necessarily involves dangling pointers to old regions. These regions may be deallocated, and then re-used for different purposes. In the case of a re-used region, the dynamic semantics of TTRC would use the overwritten contents of the store, just like a real implementation. However, the static semantics prevents TTRC from doing so.

## ACKNOWLEDGMENTS

We thank the anonymous referees for useful comments on the initial draft of this paper.

## REFERENCES

1. Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proc. of the 14th Annual IEEE symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, July 1999. IEEE Computer Society Press.
2. Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 58:299–392, 2001.
3. Lars Birkedal, Mads Tofte, and Magnus Vejlsturp. From region inference to von Neumann machines via region representation inference. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, Fla., January 1996. ACM Press.
4. Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In Hanne Riis Nielson, editor, *Proc. 28th Annual ACM Symposium on Principles of Programming Languages*, pages 155–165, London, England, January 2001. ACM Press.
5. Karl Cray, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In Alex Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, USA, January 1999. ACM Press.
6. H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume I. North Holland, 1958.
7. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [8].
8. Robert Harper. A note on: “A simplified account of polymorphic references”. *Information Processing Letters*, 57(1):15–16, January 1996. See also [7].
9. Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *HOOTS '00 Higher Order Operational Techniques in Semantics*, volume 41 of *Electronic Notes in Theoretical Computer Science*, Montreal, Canada, September 2000. Elsevier Science.
10. Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
11. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.
12. Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.
13. Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
14. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value  $\lambda$ -calculus using a stack of regions. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, OG, January 1994. ACM Press.
15. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
16. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
17. S. Dal Zilio and Andrew Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of MFCS '00*, volume 1893 of *Lecture Notes in Computer Science*, pages 1–20, 2000.