Syntactic Type Soundness Results for the Region Calculus

Cristiano Calcagno
Queen Mary, University of London
and DISI, University of Genova
ccris@dcs.qmw.ac.uk

Simon Helsen
Universität Freiburg, Germany
helsen@informatik.uni-freiburg.de

Peter Thiemann
Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

February 20, 2001

Abstract

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 ML without garbage collection. Static region and effect inference can be used to generate annotations from a given ML program. 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 soundness. Following the 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. It is slightly more complicated, but includes operations on references with destructive update. We prove (the pure fragment of) both semantics equivalent to the original evaluation-style formulation by Tofte and Talpin.

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 Smalltalk, Java, ML, and Haskell) perform automatic deallocation by incorporating garbage collection into the runtime system. While the programmer-based solution is immensely error-prone, programs can in principle be tuned for optimal memory use. 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. In consequence, deallocation might happen too late, which can lead to space leaks. In addition, garbage collection takes extra, non-productive time and can cause erratic pauses in the execution of programs. Finally, inter-operability between garbage collected languages, like ML, and non-garbage collected languages, such as C, is difficult.

The region calculus of Tofte and Talpin [14, 15] (TTRC) provides an alternative method of memory management for the functional language ML [11]. It is used as an intermediate language in an ML compiler, the ML-kit [3,4,13–15]. 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 garbage collection (in principle), while guaranteeing safety. In some instances, the region calculus can prove that a pointer is semantically dead, so that the region it points to can be safely deallocated. Standard garbage collectors cannot do this.

1.1 Related Work

The proof of consistency, or type soundness, for the region calculus as it is given by Tofte and Talpin [15] is a complicated proof using rule-based co-induction. The source of the complication is the co-inductive definition of their consistency relation, caused by the loss of information when deleting a region from the store in their particular formulation of the big-step semantics. Recently, alternative type-soundness proofs for the region calculus have been proposed.

1. Crary, Walker, and Morrisett [6] 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 [2] 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. Albeit artificial, 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 π-calculus with name groups. This is shown using a translation from the region calculus to the typed π-calculus with name groups.

4. Helsen and Thiemann [10] 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 [5] 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 the TTRC.

1.2 Contribution and Overview

The present paper is based on the work of Calcagno, Helsen, and Thiemann [1, 5, 10]. After recalling the syntax and the semantics of TTRC in Section 2, Section 3 gives a simplified account of the *store-less region calculus* (SRC) [10], without the standard treatment of polymorphism. The syntactic type soundness results from that work are stated without proofs, which may be found elsewhere [10].

Section 4 introduces a new calculus, the *imperative region calculus* (IRC). This calculus extends TTRC with operations on references. Extending the ideas of the store-less calculus, we provide a small-step operational semantics for IRC based on an explicit store and prove its type soundness, using the syntactic approach due to Wright and Felleisen [16] in a variation pioneered by Harper [8]. The reduction semantics in this paper follows Plotkin's approach to operational semantics [12].

Section 5 contains another novel contribution of the present work. It relates the three calculi by proving the equivalence of TTRC and IRC, as well as the equivalence of IRC and SRC. These technical results are inspired by Calcagno's work [5].

The first technical achievement of this work is the reformulation of the region calculus, so that a standard syntactic soundness proof à la Wright and

Felleisen [16] and Harper [8] can be constructed. We do so using a store-less formulation as well as a store-based one. While the store-less formulation is extremely simple and elegant, the store-based formulation is more complicated but supplies a soundness proof for the calculus with references, as it is actually used in the ML-kit [4].

The proofs themselves use routine inductive techniques and are therefore considerably easier than the co-inductive proofs of Tofte and Talpin.

The second technical result is the statement and proof of equivalence between the different calculi. The equivalence between TTRC and IRC requires to relate a big-step semantics with a small-step semantics. The equivalence between IRC and SRC is a simulation result between small-step semantics. In these equivalences, we do not consider IRC's operations on references.

1.3 Notation

Let s be a map from A to B. Then $\mathrm{dom}(s) = \{x \in A \mid \exists y \in B, s(x) = y\}$ is the domain of s and $\mathrm{ran}(s) = \{y \in B \mid \exists x \in A, s(x) = y\}$. The map $s|_{A'}$ is the *restriction* of s to $A' \subseteq \mathrm{dom}(s)$, defined by $s|_{A'}(a) = s(a)$, if $a \in A'$, and undefined otherwise. If $x \in A$, write $s - x$ for the map $s|_{\mathrm{dom}(s)\setminus\{x\}}$. 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 $\mathrm{ran}(s_1) \cup \mathrm{ran}(s_2)$ is a set of finite maps, then write $s_1 \leq s_2$ (s_2 *extends* s_1) if $\mathrm{dom}(s_1) \subseteq \mathrm{dom}(s_2)$ and for all $r \in \mathrm{dom}(s_1)$, $\mathrm{dom}(s_1(r)) \subseteq \mathrm{dom}(s_2(r))$ and $s_2(r)|_{\mathrm{dom}(s_1(r))} = s_1(r)$.

All mappings 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. Substitution avoids capture of variables by renaming.

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 monomorphic version of the TTRC. The addition of polymorphism is tedious, but standard. It is described elsewhere [5, 10].

2.1 Syntax of TTRC

The following grammar defines the syntax of TTRC.

$$\begin{array}{lcll} \textit{Terms} & e & ::= & x \mid c\ \mathtt{at}\ \varrho \mid \lambda x.\, e\ \mathtt{at}\ \varrho \mid e\,@\,e \mid \mathtt{letregion}\ \varrho\ \mathtt{in}\ e \mid \mathtt{copy}\,[\varrho,\varrho]\,e \\ \textit{Values} & v & ::= & (r, l) \end{array}$$

Moreover, assume that

$$\begin{array}{lcll} x & \in & \mathit{Vname} & \text{variable names} \\ c & \in & \mathit{Cname} & \text{constant symbols} \\ \varrho & \in & \mathit{RegionVars} & \text{region variables} \\ r & \in & \mathit{RegionNames} & \text{region names} \\ l & \in & \mathit{Locations} & \text{store locations} \end{array}$$

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

In comparison to the lambda calculus, constants and lambda abstractions carry a region annotation $\mathtt{at}\ \varrho$, 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 $\mathtt{letregion}\ \varrho\ \mathtt{in}\ e$ allocates a new region of memory, then evaluates e, and finally deallocates the region again. The term $\mathtt{copy}\,[\varrho, \varrho']\, e$ is a simple addition to the original region calculus. It copies a value from one region to another and stands for a prototypical primitive operation.

2.2 Dynamic Semantics of TTRC

This section paraphrases the dynamic semantics of the 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 ϱ 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 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.

2.3 Static Semantics of the Region Calculus

The static semantics deals with the following semantic objects. A *region placeholder*, ρ, ranges —for the moment— over region variables, ϱ. An *effect* φ is a finite set of region place-holders, ρ. The effect of a term, e, indicates the set of regions that may be affected by evaluation of e. *Types*, τ, and *types with place*, μ, are defined by mutual induction:

$$\begin{array}{lcl} \tau & ::= & \mathtt{int} \mid \mu \xrightarrow{\varphi} \mu \\ \mu & ::= & (\tau, \rho) \end{array}$$

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.

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

(b-const)
$$\frac{R(\varrho) = r \quad l \notin \mathrm{dom}(S(r))}{S, \mathit{VE}, R \vdash_b c \texttt{ at } \varrho \Downarrow (r,l), S + \{r \mapsto S(r) + \{l \mapsto \langle c \rangle\}\}}$$

(b-var)
$$\frac{\mathit{VE}(x) = v}{S, \mathit{VE}, R \vdash_b x \Downarrow v, S}$$

(b-abstr)
$$\frac{R(\varrho) = r \quad l \notin \mathrm{dom}(S(r))}{S, \mathit{VE}, R \vdash_b \lambda x.\, e \texttt{ at } \varrho \Downarrow (r,l), S + \{r \mapsto S(r) + \{l \mapsto \langle x, e, \mathit{VE}, R\rangle\}\}}$$

(b-app)
$$\frac{\begin{array}{c} S, \mathit{VE}, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1 \\ S_1(r_1)(l_1) = \langle x, e, \mathit{VE}', R'\rangle \\ S_1, \mathit{VE}, R \vdash_b e_2 \Downarrow v_2, S_2 \\ S_2, \mathit{VE}' + \{x \mapsto v_2\}, R' \vdash_b e \Downarrow v, S' \end{array}}{S, \mathit{VE}, R \vdash_b e_1 \mathbin{@} e_2 \Downarrow v, S'}$$

(b-letregion)
$$\frac{r \notin \mathrm{dom}(S) \quad S + \{r \mapsto [\,]\}, \mathit{VE}, R + \{\varrho \mapsto r\} \vdash_b e \Downarrow v, S'}{S, \mathit{VE}, R \vdash_b \texttt{letregion } \varrho \texttt{ in } e \Downarrow v, S' - r}$$

(b-copy)
$$\frac{\begin{array}{c} S, \mathit{VE}, R \vdash_b e \Downarrow (r,l), S' \\ S'(r)(l) = \langle c \rangle \\ R(\varrho) = r \quad R(\varrho') = r' \\ l' \notin \mathrm{dom}(S'(r')) \end{array}}{S, \mathit{VE}, R \vdash_b \texttt{copy}\,[\varrho, \varrho']\, e \Downarrow (r', l'), S' + \{r' \mapsto S'(r') + \{l' \mapsto \langle c \rangle\}\}}$$

Figure 1: Big-step evaluation relation of TTRC [15]

$$
\text{(var)} \quad \frac{TE(x) = \mu}{TE \vdash_{tt} x : \mu, \emptyset}
$$

$$
\text{(const)} \quad \frac{}{TE \vdash_{tt} c \;\texttt{at}\; \rho : (\texttt{int}, \rho), \{\rho\}}
$$

$$
\text{(abstr)} \quad \frac{TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_{tt} \lambda\, x.\, e \;\texttt{at}\; \rho : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \{\rho\}}
$$

$$
\text{(app)} \quad \frac{TE \vdash_{tt} e_1 : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \varphi_1 \quad TE \vdash_{tt} e_2 : \mu_1, \varphi_2}{TE \vdash_{tt} e_1 \,@\, e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\rho\}}
$$

$$
\text{(letregion)} \quad \frac{TE \vdash_{tt} e : \mu, \varphi \quad \varrho \notin \mathit{frv}\,(TE, \mu)}{TE \vdash_{tt} \texttt{letregion}\; \varrho \;\texttt{in}\; e : \mu, \varphi \setminus \{\varrho\}}
$$

$$
\text{(copy)} \quad \frac{TE \vdash_{tt} e : (\texttt{int}, \rho), \varphi}{TE \vdash_{tt} \texttt{copy}\,[\rho, \rho']\, e : (\texttt{int}, \rho'), \varphi \cup \{\rho, \rho'\}}
$$

Figure 2: Static Semantics of TTRC [15]

A type with place, μ, is a pair of a type, τ, and a region placeholder, ρ. The latter specifies where an object of type τ is stored.

A *typing environment*, TE, is a finite map from variables to types with place. The empty typing environment is written $[\,]$.

Write $\mathit{frv}\,(\mu)$ for the set of region place-holders that occur in μ. Analogously, $\mathit{frv}\,(TE) = \bigcup\{\mathit{frv}\,(TE(x)) \mid x \in \mathrm{dom}(TE)\}$.

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 typing environment TE the term e has type with place μ and effect φ. It differs from the presentation of Tofte and Talpin [15] in two respects: we are omitting polymorphism and effect variables. Adding either is a standard but tedious distraction from our real goal. None is really required to compare the different approaches to region calculus and its type soundness.

3 The Store-less Region Calculus

The store-less region calculus [10] provides a very simple and elegant syntactic soundness proof. It is based on the observation that terms like c `at` ρ cannot be used as values. This is because evaluation of the term c `at` ρ allocates memory, stores the constant c in it, and returns a pointer to the stored constant. In 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 terms, e; values, v; region place-holders, ρ

$$
\begin{array}{lcl}
e & ::= & v \mid x \mid c \;\texttt{at}\; \rho \mid \lambda\, x.\, e \;\texttt{at}\; \rho \mid e \,@\, e \mid \texttt{letregion}\; \varrho \;\texttt{in}\; e \mid \texttt{copy}\,[\rho_1, \rho_2]\, e \\
v & ::= & \langle c \rangle_\rho \mid \langle \lambda\, x.\, e \rangle_\rho \\
\rho & ::= & \varrho \mid \bullet
\end{array}
$$

Reduction rules

$$c \;\texttt{at}\; \varrho \to_s \langle c \rangle_\varrho \tag{1}$$

$$\lambda\, x.\, e \;\texttt{at}\; \varrho \to_s \langle \lambda\, x.\, e \rangle_\varrho \tag{2}$$

$$\texttt{letregion}\; \varrho \;\texttt{in}\; v \to_s v[\varrho \mapsto \bullet] \tag{3}$$

$$\texttt{copy}\,[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} \to_s \langle c \rangle_{\varrho_2} \tag{4}$$

$$\langle \lambda\, x.\, e \rangle_\varrho \,@\, v \to_s e[x \mapsto v] \tag{5}$$

Reduction in context

$$\frac{e \to_s e'}{e \,@\, e'' \to_s e' \,@\, e''} \tag{6}$$

$$\frac{e \to_s e'}{v \,@\, e \to_s v \,@\, e'} \tag{7}$$

$$\frac{e \to_s e'}{\texttt{letregion}\; \varrho \;\texttt{in}\; e \to_s \texttt{letregion}\; \varrho \;\texttt{in}\; e'} \tag{8}$$

$$\frac{e \to_s e'}{\texttt{copy}\,[\rho_1, \rho_2]\, e \to_s \texttt{copy}\,[\rho_1, \rho_2]\, e'} \tag{9}$$

Figure 3: Small-step reduction relation for SRC

before, or a special constant $\bullet$ (*dead region*) that denotes a deallocated region.

The reductions explicitly require a region variable ϱ 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 $v[\varrho \mapsto \bullet]$ replaces all free occurrences of ϱ 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, 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 (see Fig. 2). The difference is that these rules *(use-const)* and *(use-abstr)* provide a typing for pointers. Their effects are empty.

$$\textit{(use-const)} \quad \frac{}{TE \vdash_{tt} \langle c \rangle_\rho : (\mathtt{int}, \rho), \emptyset}$$

$$\textit{(use-abstr)} \quad \frac{TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_{tt} \langle \lambda\, x.\, e \rangle_\rho : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \emptyset}$$

3.2 Type Soundness

This section proves the syntactic type soundness for the small-step transition relation $\to_s$ 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* [7, 16], which states that a well-typed term remains well-typed under the small-step transition relation $\to_s$. 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.

We just state the lemmas and theorems here. Full proofs, including the treatment of polymorphism, may be found elsewhere [10].

Syntactic values have no effect.

Lemma 1 *For all TE, values v, and types μ, if $TE \vdash_{tt} v : \mu, \varphi$ then $\varphi = \emptyset$.*

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

Lemma 2 *If e is closed and $e \to_s e'$, 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 *Suppose $TE + \{x \mapsto \mu\} \vdash_{tt} e : \mu', \varphi'$ and $TE \vdash_{tt} v : \mu, \varphi$, then $TE \vdash_{tt} e[x \mapsto v] : \mu', \varphi'$.*

Typing is preserved under region substitution.

Lemma 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 1 (Typing Preservation) *Suppose $TE \vdash_{tt} e : \mu, \varphi$. If $e \to_s e'$ then $TE \vdash_{tt} e' : \mu, \varphi'$ where $\varphi' \subseteq \varphi$.*

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

Lemma 5 (Canonical Forms) *The following hold:*

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

2. *If* $TE \vdash_{tt} v : (\mathtt{int}, \rho), \varphi$ *then there exist* c *and* ρ *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 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.*

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 $\twoheadrightarrow_s$ be the reflexive and transitive closure of the relation $\rightarrow_s$.

Theorem 1 (Type Soundness) *Suppose that* $[\,] \vdash_{tt} e : \mu, \varphi$ *with* $\bullet \notin \varphi$.
Then, either there exists some value v *with* $e \twoheadrightarrow_s v$ *and* $[\,] \vdash_{tt} v : \mu, \emptyset$ *or, for each* e' *where* $e \twoheadrightarrow_s e'$, *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, `ref` e `at` ρ, dereferencing a reference, `!` e, and updating a reference e `:=` e. In addition, there is a new intermediate term, `region` r `in` e. The binding construct, `letregion` ϱ `in` e, reduces to `region` r `in` $e[\varrho \mapsto r]$ as soon as evaluation has committed to a particular region. Region names, r, are not subject to alpha-conversion.

A value is once again a pointer, that is, a pair of a live (r) or dead ($\bullet$) region and a location $l \in$ *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.

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

Figure 4: Syntactic categories of IRC

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, or 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 `ref` μ of references to objects of type (with places) μ.

4.2 Dynamic Semantics

The dynamic semantics (Figure 5) is defined as a relation on configurations. A configuration s, e is a pair of a store, s, and a closed expression, e.

With some abuse of notation, $e[r \mapsto \bullet]$ denotes the term e after substitution of $\bullet$ for all free occurrences of the constant r in e. Define $\mathit{frv}\,(s) = \bigcup\{\mathit{frv}\,(s(r)(l)) \mid r \in \mathrm{dom}(s), l \in \mathrm{dom}(s(r))\}$. Then $s[r \mapsto \bullet]$ denotes the store s after substitution of $\bullet$ for all $r \in \mathit{frv}\,(s)$.

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 reduces 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 specify a call-by-value semantics for IRC.

4.3 Static Semantics

To state the static semantics, we need two additional finite maps: A *region type*, k, maps locations to types. A *heap type*, h, maps a regions name to a region type, thus providing the typing for locations in the store. It is required to define the following judgments:

Reduction rules

$$s, \texttt{letregion}\ \varrho\ \texttt{in}\ e \rightarrow_i s + \{r \mapsto [\,]\}, \texttt{region}\ r\ \texttt{in}\ e[\varrho \mapsto r] \qquad \text{where } r \notin \mathrm{dom}(s) \tag{10}$$

$$s, \texttt{region}\ r\ \texttt{in}\ v \rightarrow_i s[r \mapsto \bullet] - r, v[r \mapsto \bullet] \tag{11}$$

$$s, c\ \texttt{at}\ r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l) \qquad \text{where } l \notin \mathrm{dom}(s(r)) \tag{12}$$

$$s, \lambda\, x.\, e\ \texttt{at}\ r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}, (r, l) \qquad \text{where } l \notin \mathrm{dom}(s(r)) \tag{13}$$

$$s, (r, l)\ @\ v \rightarrow_i s, e[x \mapsto v] \qquad \text{where } s(r)(l) = \langle \lambda\, x.\, e \rangle \tag{14}$$

$$s, \texttt{copy}\ [r_1, r_2]\ (r_1, l_1) \rightarrow_i s + \{r_2 \mapsto s(r_2) + \{l_2 \mapsto \langle c \rangle\}\}, (r_2, l_2) \qquad \text{where } s(r_1)(l_1) = \langle c \rangle \text{ and } l_2 \notin \mathrm{dom}(s(r_2)) \tag{15}$$

$$s, \texttt{ref}\ v\ \texttt{at}\ r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v \rangle\}\}, (r, l) \qquad \text{where } l \notin \mathrm{dom}(s(r)) \tag{16}$$

$$s, \texttt{!}\ (r, l) \rightarrow_i s, v \qquad \text{where } s(r)(l) = \langle \texttt{ref}\ v \rangle \tag{17}$$

$$s, (r, l)\ \texttt{:=}\ v \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v \rangle\}\}, v \qquad \text{where } s(r)(l) = \langle \texttt{ref}\ v' \rangle \tag{18}$$

Reduction in context

$$\textit{(ctx-region)} \quad \frac{s, e \rightarrow_i s', e'}{s, \texttt{region}\ r\ \texttt{in}\ e \rightarrow_i s', \texttt{region}\ r\ \texttt{in}\ e'}$$

$$\textit{(ctx-app-l)} \quad \frac{s, e_1 \rightarrow_i s', e_1'}{s, e_1\ @\ e_2 \rightarrow_i s', e_1'\ @\ e_2}$$

$$\textit{(ctx-app-r)} \quad \frac{s, e \rightarrow_i s', e'}{s, v\ @\ e \rightarrow_i s', v\ @\ e'}$$

$$\textit{(ctx-copy)} \quad \frac{s, e \rightarrow_i s', e'}{s, \texttt{copy}\ [r_1, r_2]\ e \rightarrow_i s', \texttt{copy}\ [r_1, r_2]\ e'}$$

$$\textit{(ctx-ref)} \quad \frac{s, e \rightarrow_i s', e'}{s, \texttt{ref}\ e\ \texttt{at}\ r \rightarrow_i s', \texttt{ref}\ e'\ \texttt{at}\ r}$$

$$\textit{(ctx-deref)} \quad \frac{s, e \rightarrow_i s', e'}{s, \texttt{!}\ e \rightarrow_i s', \texttt{!}\ e'}$$

$$\textit{(ctx-setref-l)} \quad \frac{s, e_1 \rightarrow_i s', e_1'}{s, e_1\ \texttt{:=}\ e_2 \rightarrow_i s', e_1'\ \texttt{:=}\ e_2}$$

$$\textit{(ctx-setref-r)} \quad \frac{s, e \rightarrow_i s', e'}{s, v\ \texttt{:=}\ e \rightarrow_i s', v\ \texttt{:=}\ e'}$$

Figure 5: Dynamic semantics of IRC

expression typing	$h, TE \vdash e : \mu, \varphi$;
configuration typing	$h \vdash s, e : \mu, \varphi$;
heap typing	$h \vdash s$; and
storables typing	$h \vdash w : \tau$.

Expression typing (Fig. 6) 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 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 1 The set of 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` ϱ `in` e only makes sense if e is pure. In particular, `letregion` binds a region variable, which is subject to alpha-conversion. In contrast, `region` declares that a particular region name is in use and region names cannot be alpha-converted. Furthermore, substitution always inserts pure terms (values (ρ, l)) into pure terms. This explains the difference between the `letregion` and `region`, which have identical typing rules.

There is just one rule for configuration typing: A configuration s, e has type with place μ and effect φ under heap type h, if h is a valid heap type for s and e is a closed expression typeable with heap type h.

$$\textit{(conf)}\quad \frac{h \vdash s \qquad h, [\,] \vdash e : \mu, \varphi}{h \vdash s, e : \mu, \varphi}$$

There is also just one rule for heap typing:

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

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. 7) 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.

(pointer)
$$\frac{h(r)(l) = \tau}{h,\, TE \vdash (r,l) : (\tau, r), \emptyset}$$

(dead)
$$\frac{}{h,\, TE \vdash (\bullet, l) : (\tau, \bullet), \emptyset}$$

(var)
$$\frac{TE(x) = \mu}{h,\, TE \vdash x : \mu, \emptyset}$$

(const)
$$\frac{}{h,\, TE \vdash c \;\texttt{at}\; \rho : (\texttt{int}, \rho), \{\rho\}}$$

(abstr)
$$\frac{h,\, TE + \{x \mapsto \mu_2\} \vdash e : \mu_1, \varphi \qquad \varphi \subseteq \varphi'}{h,\, TE \vdash \lambda\, x.\, e \;\texttt{at}\; \rho : (\mu_2 \xrightarrow{\varphi'} \mu_1, \rho), \{\rho\}}$$

(app)
$$\frac{h,\, TE \vdash e_1 : (\mu_2 \xrightarrow{\varphi} \mu_1, \rho), \varphi_1 \qquad h,\, TE \vdash e_2 : \mu_2, \varphi_2}{h,\, TE \vdash e_1 \mathbin{@} e_2 : \mu_2, \varphi_1 \cup \varphi_2 \cup \varphi \cup \{\rho\}}$$

(copy)
$$\frac{h,\, TE \vdash e : (\texttt{int}, \rho_1), \varphi}{h,\, TE \vdash \texttt{copy}\, [\rho_1, \rho_2]\, e : (\texttt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}}$$

(letregion)
$$\frac{h,\, TE \vdash e : \mu, \varphi \qquad \varrho \notin \mathit{frv}\,(TE, \mu)}{h,\, TE \vdash \texttt{letregion}\; \varrho \;\texttt{in}\; e : \mu, \varphi \setminus \{\varrho\}}$$

(useregion)
$$\frac{h,\, TE \vdash e : \mu, \varphi \qquad r \notin \mathit{frv}\,(TE, \mu)}{h,\, TE \vdash \texttt{region}\; r \;\texttt{in}\; e : \mu, \varphi \setminus \{r\}}$$

(ref)
$$\frac{h,\, TE \vdash e : \mu, \varphi}{h,\, TE \vdash \texttt{ref}\; e \;\texttt{at}\; \rho : (\texttt{ref}\; \mu, \rho), \varphi \cup \{\rho\}}$$

(deref)
$$\frac{h,\, TE \vdash e : (\texttt{ref}\; \mu, \rho), \varphi}{h,\, TE \vdash \texttt{!}\; e : \mu, \varphi \cup \{\rho\}}$$

(setref)
$$\frac{h,\, TE \vdash e_1 : (\texttt{ref}\; \mu, \rho), \varphi_1 \qquad h,\, TE \vdash e_2 : \mu, \varphi_2}{h,\, TE \vdash e_1 \;\texttt{:=}\; e_2 : \mu, \varphi_1 \cup \varphi_2 \cup \{\rho\}}$$

Figure 6: Expression typing

$$\textit{(stored-const)} \qquad \frac{}{h \vdash \langle c \rangle : \texttt{int}}$$

$$\textit{(stored-abstr)} \qquad \frac{h, \{x \mapsto \mu_2\} \vdash e : \mu_1, \varphi \qquad \varphi \subseteq \varphi'}{h \vdash \langle \lambda x.\, e \rangle : \mu_2 \xrightarrow{\varphi'} \mu_1}$$

$$\textit{(stored-ref)} \qquad \frac{h, [\,] \vdash v : \mu, \emptyset}{h \vdash \langle \texttt{ref}\ v \rangle : \texttt{ref}\ \mu}$$

Figure 7: Storable typing

Lemma 6 *If* $h, TE \vdash v : \mu, \varphi$ *then* $\varphi = \emptyset$.

Lemma 7 *If* $h, TE \vdash e : \mu, \varphi$ *then* $h, TE' + TE \vdash e : \mu, \varphi$.

Lemma 8 *If* $h, TE \vdash e : \mu, \varphi$ *and* $h \leq h'$ *then* $h', TE \vdash e : \mu, \varphi$.

Lemma 9 *If* $h, TE \vdash e : \mu, \varphi$ *and* $r \in \mathrm{dom}(h) \setminus \mathit{frv}\,(h, TE, e, \mu, \varphi)$ *then* $h - r, TE \vdash e : \mu, \varphi$.

Lemma 10 (Region Substitution) *Let* θ *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., $\theta_r = \{\varrho \mapsto r\}$*).*

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 11 (Value Substitution)
Suppose $h, TE + \{x \mapsto \mu'\} \vdash e : \mu, \varphi$ *and* $h, TE \vdash v : \mu', \emptyset$.
Then $h, TE \vdash e[x \mapsto v] : \mu, \varphi$.

Typing 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 3 (Typing Preservation)
Suppose $h \vdash s, e : \mu, \varphi$ *and* $s, e \rightarrow_i s', e'$.
Then there exist h' *and* φ' *such that* $h' \vdash s', e' : \mu, \varphi'$ *where* $\varphi' \subseteq \varphi$.

Proof: By induction on the definition of $s, e \rightarrow_i s', e'$.

Case $s, \texttt{letregion}\ \varrho\ \texttt{in}\ e \rightarrow_i s + \{r \mapsto [\,]\}, \texttt{region}\ r\ \texttt{in}\ e[\varrho \mapsto r]$ where $r \notin \mathrm{dom}(s)$. Since $h \vdash s, \texttt{letregion}\ \varrho\ \texttt{in}\ e : \mu, \varphi$ is assumed it must be that $h \vdash s$ and $h, [\,] \vdash \texttt{letregion}\ \varrho\ \texttt{in}\ e : \mu, \varphi$, by rule *(conf)*. By rule *(letregion)*, it must be that $h, [\,] \vdash e : \mu, \varphi'$ and $\varrho \notin \mathit{frv}(\mu)$ and $\varphi = \varphi' \setminus \{\varrho\}$. By Lemma 10, $h[\varrho \mapsto r], [\,][\varrho \mapsto r] \vdash e[\varrho \mapsto r] : \mu[\varrho \mapsto r], \varphi'[\varrho \mapsto r]$. Since $\mathrm{dom}(h) = \mathrm{dom}(s)$, by rule *(heap)* for $h \vdash s$, it follows that $r \notin \mathrm{dom}(h)$. Since furthermore $\varrho \notin \mathit{frv}(\mu)$, the above judgment is equivalent to $h, [\,] \vdash e[\varrho \mapsto r] : \mu, \varphi'[\varrho \mapsto r]$ where r does not occur in μ. These are exactly the assumptions for rule *(useregion)*, hence $h, [\,] \vdash \texttt{region}\ r\ \texttt{in}\ e : \mu, \varphi$. Let $h' = h + \{r \mapsto [\,]\}$. Since $h \leq h'$, Lemma 8 yields that $h', [\,] \vdash \texttt{region}\ r\ \texttt{in}\ e : \mu, \varphi$. Since $h' \vdash s + \{r \mapsto [\,]\}$, rule *(conf)* yields $h' \vdash s + \{r \mapsto [\,]\}, \texttt{region}\ r\ \texttt{in}\ e : \mu, \varphi$.

Case $s, \texttt{region}\ r\ \texttt{in}\ v \rightarrow_i s[r \mapsto \bullet] - r, v[r \mapsto \bullet]$. Since $h \vdash s, \texttt{region}\ r\ \texttt{in}\ v : \mu, \varphi$ is assumed it must be that $h \vdash s$ and $h, [\,] \vdash \texttt{region}\ r\ \texttt{in}\ v : \mu, \varphi$, by rule *(conf)*. By rule *(useregion)*, it must be that $h, [\,] \vdash v : \mu, \varphi'$ and $r \notin \mathit{frv}(\mu)$ and $\varphi = \varphi' \setminus \{r\}$. By Lemma 10, $h[r \mapsto \bullet], [\,][r \mapsto \bullet] \vdash v[r \mapsto \bullet] : \mu[r \mapsto \bullet], \varphi'[r \mapsto \bullet]$ also holds. By Lemma 6, $\varphi' = \emptyset$. Taken together with the assumption on the (non-) occurrence of r in μ it holds that $h[r \mapsto \bullet], [\,] \vdash v[r \mapsto \bullet] : \mu, \emptyset$. Since $r \notin \mathit{frv}(h[r \mapsto \bullet], v[r \mapsto \bullet], \mu)$, Lemma 9 implies that $h', [\,] \vdash v[r \mapsto \bullet] : \mu, \emptyset$ where $h' = h[r \mapsto \bullet] - r$. Since $h' \vdash s[r \mapsto \bullet] - r$, rule *(conf)* yields that $h' \vdash s[r \mapsto \bullet] - r, v[r \mapsto \bullet] : \mu, \emptyset$.

Case $s, c\ \texttt{at}\ r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l)$ where $l \notin \mathrm{dom}(s(r))$. By assumption, $h \vdash s, c\ \texttt{at}\ r : \mu, \varphi$. By rule *(conf)* it must be that $h \vdash s$ and $h, [\,] \vdash c\ \texttt{at}\ r : \mu, \varphi$. By rule *(const)*, it must be that $\mu = (\texttt{int}, r)$ and $\varphi = \{r\}$. Let $h' = h + \{r \mapsto h(r) + \{l \mapsto \texttt{int}\}\}$ and $s' = s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}$. By rules *(stored-const)* and *(heap)*, $h' \vdash s'$. Since $h'(r)(l) = \texttt{int}$ rule *(pointer)* yields $h', [\,] \vdash (r, l) : (\texttt{int}, r), \emptyset$. Applying rule *(conf)* yields $h' \vdash s', (r, l) : (\texttt{int}, r), \emptyset$. This suffices for the claim, since $\emptyset \subseteq \{r\}$.

Case $s, \lambda\, x.\, e\ \texttt{at}\ r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}, (r, l)$ where $l \notin \mathrm{dom}(s(r))$. By assumption, $h \vdash s, \lambda\, x.\, e\ \texttt{at}\ r : \mu, \varphi$. By rule *(conf)* it must be that $h \vdash s$ and $h, [\,] \vdash \lambda\, x.\, e\ \texttt{at}\ r : \mu, \varphi$. By rule *(abstr)*, it must be 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 $l \notin \mathrm{dom}(s(r)) = \mathrm{dom}(h(r))$, 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 $s, (r,l) \mathbin{@} v \to_i s, e[x \mapsto v]$ 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 $h, [\,] \vdash (r,l)@v : \mu, \varphi$. By rule *(app)*, it must be that $h, [\,] \vdash (r,l) : (\mu_2 \xrightarrow{\varphi'} \mu, r), \varphi_1$ and $h, [\,] \vdash v : \mu_2, \varphi_2$ and $\varphi = \varphi_1 \cup \varphi_2 \cup \varphi' \cup \{r\}$. By 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 $h \vdash \langle \lambda\, x.\, e\rangle : \mu_2 \xrightarrow{\varphi'} \mu$. By rule *(stored-abstr)*, it must be that $h, \{x \mapsto \mu_2\} \vdash e : \mu, \varphi''$, for some $\varphi'' \subseteq \varphi'$. By Lemma 11, $h, [\,] \vdash e[x \mapsto v] : \mu, \varphi''$, 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 $s, \texttt{copy}\,[r_1, r_2]\,(r_1, l_1) \to_i s + \{r_2 \mapsto s(r_2) + \{l_2 \mapsto s(r_1, l_1)\}\}, (r_2, l_2)$ where $s(r_1)(l_1) = \langle c\rangle$ and $l_2 \notin \mathrm{dom}(s(r_2))$.
By assumption $h \vdash s, \texttt{copy}\,[r_1, r_2]\,(r_1, l_1) : \mu, \varphi$. By rule *(conf)*, it must be that $h \vdash s$ and $h, [\,] \vdash \texttt{copy}\,[r_1, r_2]\,(r_1, l_1) : \mu, \varphi$. By rule *(copy)*, it must be that $h, [\,] \vdash (r_1, l_1) : (\texttt{int}, r_1), \varphi'$ and $\mu = (\texttt{int}, r_2)$ and $\varphi = \varphi' \cup \{r_1, r_2\}$. By rule *(pointer)*, $h(r_1)(l_1) = \texttt{int}$ and $\varphi' = \emptyset$. By rule *(heap)*, $s(r_1)(l_1) = \langle c\rangle$. Let $h' = h + \{r_2 \mapsto h(r_2) + \{l_2 \mapsto \texttt{int}\}\}$ and $s' = s + \{r_2 \mapsto s(r_2) + \{l_2 \mapsto s(r_1, l_1)\}\}$. By rules *(stored-const)* and *(heap)*, $h' \vdash s'$. By rule *(pointer)* $h', [\,] \vdash (r_2, l_2) : (\texttt{int}, r_2), \emptyset$. Applying *(conf)* concludes (observing that $\emptyset \subseteq \{r_1, r_2\}$).

Case $s, \texttt{ref}\ v\ \texttt{at}\ r \to_i s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v\rangle\}\}, (r,l)$ where $l \notin \mathrm{dom}(s(r))$.
By assumption $h \vdash s, \texttt{ref}\ v\ \texttt{at}\ r : \mu, \varphi$. By rule *(conf)*, it must be that $h \vdash s$ and $h, [\,] \vdash \texttt{ref}\ v\ \texttt{at}\ r : \mu, \varphi$. By rule *(ref)*, it must be that $h, [\,] \vdash v : \mu', \varphi'$ where $\varphi = \varphi' \cup \{r\}$ and $\mu = (\texttt{ref}\ \mu', r)$. Let $h' = h + \{r \mapsto h(r) + \{l \mapsto \texttt{ref}\ \mu'\}\}$ and $s' = s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v\rangle\}\}$. By rule *(heap)*, $h' \vdash s'$ By rule *(pointer)*, $h', [\,] \vdash (r,l) : (\texttt{ref}\ \mu', r), \emptyset$. By rule *(conf)*, $h' \vdash s', (r,l) : (\texttt{ref}\ \mu', r), \emptyset$. Conclude by observing that $\emptyset \subseteq \varphi$.

Case $s, !\,(r,l) \to_i s, v$ where $s(r)(l) = \langle \texttt{ref}\ v\rangle$. By assumption $h \vdash s, !\,(r,l) : \mu, \varphi$. By rule *(conf)*, it must be that $h \vdash s$ and $h, [\,] \vdash !\,(r,l) : \mu, \varphi$. By rule *(deref)*, it must be that $h, [\,] \vdash (r,l) : (\texttt{ref}\ \mu, r), \varphi'$ and $\varphi = \varphi' \cup \{\rho\}$. By rule *(pointer)*, it must be that $\rho = r$. By rule *(heap)*, it must be that $h \vdash s(r)(l) : h(r)(l)$, that is, $h \vdash \langle \texttt{ref}\ v\rangle : \texttt{ref}\ \mu$. By rule *(stored-ref)* $h, [\,] \vdash v : \mu, \emptyset$. Applying rule *(conf)* yields $h \vdash s, v : \mu, \emptyset$. Conclude by observing that $\emptyset \subseteq \varphi$.

Case $s, (r,l) \mathrel{\texttt{:=}} v \to_i s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v\rangle\}\}, v$. By assumption $h \vdash s, (r,l) \mathrel{\texttt{:=}} v : \mu, \varphi$. By rule *(conf)*, it must be that $h \vdash s$ and $h, [\,] \vdash (r,l) \mathrel{\texttt{:=}} v : \mu, \varphi$. By rule *(setref)*, it must be that $h, [\,] \vdash (r,l) : (\texttt{ref}\ \mu, r), \varphi'$ and $h, [\,] \vdash v : \mu, \varphi''$ and $\varphi = \varphi' \cup \varphi'' \cup \{r\}$.

By rule *(stored-ref)*, it holds that $h \vdash \langle \texttt{ref}\ v\rangle : (\texttt{ref}\ \mu, r)$. Hence, with $s' = s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v\rangle\}\}$ this amounts to $h \vdash s'(r)(l) : (\texttt{ref}\ \mu, r)$, 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 $\dfrac{s, e \rightarrow_i s', e'}{s, \texttt{region}\ r\ \texttt{in}\ e \rightarrow_i s, \texttt{region}\ r\ \texttt{in}\ e'}$**.** By assumption, $h \vdash s, \texttt{region}\, r\, \texttt{in}\, e : \mu, \varphi$. By rule *(conf)*, it must be that $h \vdash s$ and $h, [\,] \vdash \texttt{region}\, r\, \texttt{in}\, e : \mu, \varphi$. By rule *(useregion)*, it must be that $h, [\,] \vdash e : \mu, \varphi'$ and $\varphi = \varphi' \setminus \{r\}$.

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

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

Case *(ctx-app-l)***.** analogous.

Case *(ctx-app-r)***.** analogous.

Case *(ctx-copy)***.** analogous.

Case *(ctx-ref)***.** analogous.

Case *(ctx-deref)***.** analogous.

Case *(ctx-setref-l)***.** analogous.

Case *(ctx-setref-r)***.** analogous.

□

Lemma 12 (Canonical Forms) *Suppose* $h \vdash s, (r, l) : \mu, \emptyset$, *then*

1. *If* $\mu = (\texttt{int}, r)$ *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* $\mu = (\texttt{ref}\ \mu_1, r)$ *then* $s(r)(l) = \langle \texttt{ref}\ v \rangle$*, for some* v*.*

Proof: Slightly more complicated as usual due to the indirection through the store.

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

Case 2. Analogous.

Case 3. Analogous. □

Proposition 4 (Progress) *Suppose* $h \vdash s, e : \mu, \varphi$ *where* $\varphi \subseteq$ *RegionNames. Then either*

1. *e is a value; or*

2. *there exist s', e' such that $s, e \to_i s', e'$.*

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

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

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

Proof: By induction on the structure of e:

Case v**.** (Case 1).

Case x**.** Is not closed.

Case c **at** ρ**.** By assumption $h \vdash s, c\ \mathtt{at}\ \rho : \mu, \varphi$ where $\varphi \subseteq \mathit{RegionNames}$. Hence, by *(conf)* and *(const)*, $\mu = (\mathtt{int}, \rho)$ and $\varphi = \{\rho\}$ and $\rho = r$. By reduction rule (12), $s, c\ \mathtt{at}\ r$ is a redex (Case 2).

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

Case $\mathtt{ref}\ e\ \mathtt{at}\ \rho$**.** By assumption $h \vdash s, \mathtt{ref}\ e\ \mathtt{at}\ \rho : \mu, \varphi$ with $\varphi \subseteq \mathit{RegionNames}$. Hence, $\mu = (\mathtt{ref}\ \mu_1, \rho)$ and $\varphi = \varphi' \cup \{\rho\}$ and $\rho = r$. In conclusion, $s, \mathtt{ref}\ e\ \mathtt{at}\ r$ is a redex (Case 2).

Case $\mathtt{copy}\,[\rho_1, \rho_2]\, e$**.** By assumption $h \vdash s, \mathtt{copy}\,[\rho_1, \rho_2]\, e : \mu, \varphi$ with $\varphi \subseteq \mathit{RegionNames}$. Hence, $\mu = (\mathtt{int}, \rho_2)$ and $\varphi = \varphi' \cup \{\rho_1, \rho_2\}$ and $\rho_1 = r_1$ and $\rho_2 = r_2$. By rules *(conf)* and *(copy)*, it must be that $h, [\,] \vdash e : (\mathtt{int}, r_1), \varphi'$ where $\varphi' \subseteq \mathit{RegionNames}$. By rule *(conf)*, it follows that $h \vdash s, e : (\mathtt{int}, r_1), \varphi'$ where $\varphi' \subseteq \mathit{RegionNames}$.

By induction, either e is a value or it can be reduced. If e is a value then by Lemma 12 item 1, $e = (r_1, l)$ and $s(r_1)(l) = \langle c \rangle$ for some c. Hence, $s, \mathtt{copy}\,[r_1, r_2]\,(r_1, l)$ can be reduced by rule (15) (case 2).

If $s, e \to_i s', e'$ then, by *(ctx-copy)*, $s, \mathtt{copy}\,[r_1, r_2]\, e \to_i s', \mathtt{copy}\,[r_1, r_2]\, e'$.

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

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

By induction, either e is a value or it can be reduced. If e is a value then, by Lemma 12 item 3, $e = (r_1, l)$ and $s(r_1)(l) = \langle \mathtt{ref}\ v \rangle$. Hence, $s, !\,(r_1, l)$ is a redex for reduction rule (17) (case 2).

If $s, e \to_i s', e'$ then, by *(ctx-deref)*, $s, !\, e \to_i s', !\, e'$.

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

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 $\varphi_1 \subseteq \mathit{RegionNames}$. 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 *(ctx-app-l)*, $s, e_1 \mathbin{@} e_2 \rightarrow_i s', e_1' \mathbin{@} e_2$.

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

If $s, e_2 \rightarrow_i s', e_2'$ then, by rule *(ctx-app-r)*, $s, v_1 \mathbin{@} e_2 \rightarrow_i s', v_1 \mathbin{@} e_2'$.

If $e_2 = v_2$ is a value, too, then, by Lemma 12, $e_1 = (r, l)$ and $s(r)(l) = \langle \lambda\, x. e \rangle$, for some x and e. Hence, $s, (r, l) \mathbin{@} v_2$ is reduce-able with rule (14) (case 2).

Case e_1 `:=` e_2**.** By assumption $h \vdash s, e_1 \texttt{ := } e_2 : \mu, \varphi$ with $\varphi \subseteq \mathit{RegionNames}$. By rules *(conf)* and *(setref)*, $h, [\,] \vdash e_1 : (\texttt{ref}\ \mu, \rho), \varphi_1$ and $h, [\,] \vdash e_2 : \mu, \varphi_2$ and $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$, so that $\rho = r$ since $\varphi \subseteq \mathit{RegionNames}$.

If e_1 is not a value then applying rule *(conf)* yields $h, [\,] \vdash s, e_1 : (\texttt{ref}\ \mu, r), \varphi_1$ with $\varphi_1 \subseteq \mathit{RegionNames}$. 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 *(ctx-setref-l)*, $s, e_1 \texttt{ := } e_2 \rightarrow_i s', e_1' \texttt{ := } e_2$.

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

Applying induction to e_2 yields two cases. If $e_2 = v_2$ is a value, then by Lemma 12, $e_1 = (r, l)$ and $s(r)(l) = \langle \texttt{ref}\ v \rangle$, for some v. Hence, $s, (r, l) \texttt{ := } v_2$ is a redex for rule (18) (case 2).

Case `letregion` ϱ `in` e**.** This expression is reduce-able with reduction rule (10) (case 2).

Case `region` r `in` e**.** By assumption $h \vdash s, \texttt{region}\ r\ \texttt{in}\ e : \mu, \varphi$ with $\varphi \subseteq \mathit{RegionNames}$. By rules *(conf)* and *(useregion)*, it must be that $h, [\,] \vdash e : \mu, \varphi'$ where $\varphi' \subseteq \varphi \cup \{r\}$. Since $r \in \mathit{RegionNames}$, $\varphi' \subseteq \mathit{RegionNames}$. By rule *(conf)*, it follows that $h \vdash s, e : \mu, \varphi'$ where $\varphi' \subseteq \mathit{RegionNames}$.

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

If $s, e \rightarrow_i s', e'$ then, by *(ctx-region)*, $s, \texttt{region}\, r\ \texttt{in}\, e \rightarrow_i s', \texttt{region}\, r\ \texttt{in}\, e'$.

□

Let $\twoheadrightarrow_i$ 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 2 (Type Soundness)
Suppose $h \vdash s, e : \mu, \varphi$ *and* $\varphi \subseteq \mathit{RegionNames}$. *Then, either*

- *there exist a value* v *and heap type* h' *and store* s' *such that* $s, e \twoheadrightarrow_i s', v$ *and* $h' \vdash s', v : \mu, \emptyset$; *or*
- *for each* s', e' *with* $s, e \twoheadrightarrow_i s', e'$, *there exist* s'', e'' *such that* $s', e' \rightarrow_i s'', e''$.

5 Equivalences between calculi

In this section, we relate the three different calculi TTRC, SRC and IRC to each other. Instead of IRC, we consider pure IRC (PIRC), which excludes the operations on references. First, we establish a relation between TTRC and PIRC and then we relate PIRC with 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.

To this end, we define a relation

$$\begin{aligned} \mathcal{Q} \;\subseteq\;\; & \mathit{TTRC\text{-}Store} \times\; \mathit{TTRC\text{-}ValueEnv} \times\; \mathit{TTRC\text{-}RegionEnv} \times \\ & \mathit{TTRC\text{-}Terms} \times\; \mathit{PIRC\text{-}Store} \times\; \mathit{PIRC\text{-}Terms}, \end{aligned}$$

which relates TTRC configurations to PIRC configurations, an auxiliary relation

$$\mathcal{Q}_v \subseteq\; \mathit{TTRC\text{-}Store} \times\; \mathit{TTRC\text{-}Values} \times\; \mathit{PIRC\text{-}Store} \times\; \mathit{PIRC\text{-}Values},$$

which relates TTRC values to PIRC values, and a relation

$$\mathcal{Q}_s \subseteq\; \mathit{TTRC\text{-}Store} \times\; \mathit{PIRC\text{-}Store},$$

relating TTRC stores to PIRC stores. Figure 8 shows their mutually recursive definition. Note that the relation $\mathcal{Q}$ does not relate intermediate (see remark 1) PIRC terms.

Before formulating the equivalence theorem, we state several technical lemmas needed for the proof.

The following lemma extends lemma 4.1 of [15]:

Lemma 13 *The following hold:*

1. $S_1, \mathit{VE}, R \vdash_b e \Downarrow v, S_2$ *implies* $S_1 \leq S_2$ *and* $\mathrm{dom}(S_1) = \mathrm{dom}(S_2)$;
2. *Suppose* e_1 *is a pure term (see remark 1). Then* $s_1, e_1 \twoheadrightarrow_i s_2, e_2$ *implies* $s_1 \leq s_2$. *Moreover, if* $e_2 \in \mathit{Values}$ *then* $\mathit{dom}(s_1) = \mathit{dom}(s_2)$.

Lemma 14 *If* $S_1 \leq S_2$ *and* $s_1 \leq s_2$ *and* $\mathrm{dom}(S_2) = \mathrm{dom}(s_2)$ *then*

$$\mathcal{Q}_s(S,s) \Leftrightarrow \mathrm{dom}(S) = \mathrm{dom}(s) \wedge (\forall r \in \mathrm{dom}(S))\ \mathrm{dom}(S(r)) = \mathrm{dom}(s(r))$$

$$\begin{array}{rcl}
\mathcal{Q}_v(S,(r,l),s,(r,l)) & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge r \in \mathrm{dom}(S) \wedge l \in \mathrm{dom}(S(r)) \quad \wedge \\
& & (S(r(l)) = \langle c \rangle = s(r(l)) \quad \vee \\
& & (S(r(l)) = \langle x, e, VE, R \rangle \wedge s(r(l)) = \langle \lambda\, x.\, e' \rangle \wedge \\
& & \ \mathcal{Q}(S, VE - x, R, e, s, e')) \\
\mathcal{Q}_v(S,(r,l),s,(\bullet,l)) & \Leftrightarrow & \mathcal{Q}_s(S,s)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{Q}(S, VE, R, c \texttt{ at } \varrho, s, c \texttt{ at } \varrho) & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge \varrho \notin \mathrm{dom}(R) \\
\mathcal{Q}(S, VE, R, c \texttt{ at } \varrho, s, c \texttt{ at } r) & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge R(\varrho) = r \\
\mathcal{Q}(S, VE, R, c \texttt{ at } \varrho, s, c \texttt{ at } \bullet) & \Leftrightarrow & \mathcal{Q}_s(S,s) \\
\mathcal{Q}(S, VE, R, x, s, v) & \Leftrightarrow & \mathcal{Q}_v(S, VE(x), s, v) \\
\mathcal{Q}(S, VE, R, x, s, x) & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge x \notin \mathrm{dom}(VE) \\
\mathcal{Q}(S, VE, R, \lambda\, x.\, e \texttt{ at } \varrho, s, \lambda\, x.\, e' \texttt{ at } \varrho) & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge \varrho \notin \mathrm{dom}(R) \wedge \\
& & \mathcal{Q}(S, VE - x, R, e, s, e') \\
\mathcal{Q}(S, VE, R, \lambda\, x.\, e \texttt{ at } \varrho, s, \lambda\, x.\, e' \texttt{ at } r) & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge R(\varrho) = r \wedge \\
& & \mathcal{Q}(S, VE - x, R, e, s, e') \\
\mathcal{Q}(S, VE, R, \lambda\, x.\, e \texttt{ at } \varrho, s, \lambda\, x.\, e' \texttt{ at } \bullet) & \Leftrightarrow & \mathcal{Q}_s(S,s) \\
\mathcal{Q}(S, VE, R, e_1 \mathbin{@} e_2, s, e_1' \mathbin{@} e_2') & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge \mathcal{Q}(S, VE, R, e_1, s, e_1') \wedge \\
& & \mathcal{Q}(S, VE, R, e_2, s, e_2') \\
\mathcal{Q}(S, VE, R, \texttt{letregion } \varrho \texttt{ in } e, & \Leftrightarrow & \mathcal{Q}_s(S,s) \wedge \mathcal{Q}(S, VE, R - \varrho, e, s, e') \\
\quad s, \texttt{letregion } \varrho \texttt{ in } e') & &
\end{array}$$

Figure 8: Relation between TTRC and PIRC terms

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: Case analysis on the definitions of $\mathcal{Q}$ and $\mathcal{Q}_v$. $\square$

The next lemma deals with the substitution of a value for a variable. It is used in the proof case for beta reduction.

Lemma 15 *If $\mathcal{Q}(S, VE - x, R, e, s, e')$ and $\mathcal{Q}_v(S, v, s, v')$ then $\mathcal{Q}(S, VE + \{x \mapsto v\}, R, e, s, e'[x \mapsto v'])$.*

Proof: By induction on e, the only interesting case being the one for variables. All other cases go through by appealing to the induction hypothesis.

Case x'. Suppose that $\mathcal{Q}(S, VE - x, R, x', s, e')$ and $\mathcal{Q}_v(S, v, s, v')$. If $x' \neq x$ then the claim is immediate. If $x = x'$ then $e' = x$ so that $e'[x \mapsto v'] = v'$, and $\mathcal{Q}(S, VE + \{x \mapsto v\}, R, x, s, v')$ holds. $\square$

The next lemma deals with region allocation.

Lemma 16 *The following hold:*

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

2. *If $\mathcal{Q}_s(S, s)$ then $\mathcal{Q}_s(S + \{r \mapsto [\,]\}, s + \{r \mapsto [\,]\})$.*

Proof: Item 1: By induction on e. The assumption for the store is always a simple appeal to item 2, which is left implicit.

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

Subcase $x \notin \mathrm{dom}(VE)$. In this case, $e' = x$ and $\mathcal{Q}_s(S, s)$. By item 2, $\mathcal{Q}_s(S + \{r \mapsto [\,]\}, s + \{r \mapsto [\,]\})$. The claim holds, since VE does not change.

Subcase $VE(x) = V$. $\mathcal{Q}_v(S, V, s, v)$, for some v, and $e' = v$. The claim holds independent of R.

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

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

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

Subcase $e' = c \texttt{ at } r'$. Hence, $(R' - \varrho)(\varrho') = r'$, so that $\mathcal{Q}(S + \{r \mapsto [\,]\}, VE, R + \{\varrho \mapsto r\}, c \texttt{ at } \varrho', s + \{r \mapsto [\,]\}, c \texttt{ at } r')$.

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

Case $\lambda x.\, e$ `at` ϱ. Analogous to the previous case, plus appealing to the induction hypothesis.

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

Case `letregion` ϱ' `in` e. Since $\mathcal{Q}(S, VE, R - \varrho, \texttt{letregion } \varrho' \texttt{ in } e, s, e')$, it must be that $e' = \texttt{letregion } \varrho' \texttt{ in } e''$ 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 definition of $\mathcal{Q}$ yields that $\mathcal{Q}(S + \{r \mapsto [\,]\}, VE, R + \{\varrho \mapsto r\}, \texttt{letregion } \varrho' \texttt{ in } e, s + \{r \mapsto [\,]\}, \texttt{letregion } \varrho' \texttt{ in } e''[\varrho \mapsto r])$.

Case `copy` $[\varrho_1, \varrho_2]\, e$. Immediate by appealing to the induction hypothesis.

Item 2: Immediate by definition. □

The next lemma deals with region deallocation.

Lemma 17 *The following hold:*

1. *If* $\mathcal{Q}(S, VE, R, e, s', e')$ *then* $\mathcal{Q}(S - r_0, VE, R, e, s'[r_0 \mapsto \bullet] - r_0, e'[r_0 \mapsto \bullet])$.
2. *If* $\mathcal{Q}_s(S, s')$ *then* $\mathcal{Q}_s(S - r_0, s'[r_0 \mapsto \bullet] - r_0)$.
3. *If* $\mathcal{Q}_v(S, V, s', v')$ *then* $\mathcal{Q}_v(S - r_0, V, s'[r_0 \mapsto \bullet] - r_0, v'[r_0 \mapsto \bullet])$.

Proof: **Item** 1: By induction on e.

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

Subcase $x \in \mathrm{dom}(VE)$. In this case $V = VE(x)$ and it must be that $e' = v'$ and $\mathcal{Q}_v(S, V, s', v')$. There are two cases.

If $v' = (\bullet, l)$ then the claim holds.

Suppose now that $v' = (r, l)$. If $r \neq r_0$ then the claim holds. If $r = r_0$ then $v'[r_0 \mapsto \bullet] = (\bullet, l)$ and the claim holds, too.

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

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

Subcase $e' = c$ `at` ϱ. So, $\varrho \notin \mathrm{dom}(R)$ and $\mathcal{Q}_s(S, s')$.

Since $c \texttt{ at } \varrho[r_0 \mapsto \bullet] = c \texttt{ at } \varrho$, it is immediate that $\mathcal{Q}(S - r_0, VE, R', c \texttt{ at } \varrho, s'[r_0 \mapsto \bullet] - r_0, c \texttt{ at } \varrho)$.

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

If $r \neq r_0$ then the claim holds. If $r = r_0$ then $c \texttt{ at } r[r_0 \mapsto \bullet] = c \texttt{ at } \bullet$ and the claim holds, too.

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

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

Case $e_1 \mathbin{@} e_2$. Simply appeals to the induction hypothesis for e_1 and e_2.

Case $\texttt{letregion}\ \varrho\ \texttt{in}\ e$. Since $\mathcal{Q}(S, VE, R, \texttt{letregion}\ \varrho\ \texttt{in}\ e, s', e')$ it must be that $e' = \texttt{letregion}\ \varrho\ \texttt{in}\ e''$ and $\mathcal{Q}_s(S, s')$ and $\mathcal{Q}(S, VE, R - \varrho, e, s', e'')$. By item 2 $\mathcal{Q}_s(S - r_0, s'[r_0 \mapsto \bullet] - r_0)$. By induction $\mathcal{Q}(S - r_0, VE, R - \varrho, e, s'[r_0 \mapsto \bullet] - r_0, e''[r_0 \mapsto \bullet])$, so that $\mathcal{Q}(S - r_0, VE, R, \texttt{letregion}\ \varrho\ \texttt{in}\ e, s'[r_0 \mapsto \bullet] - r_0, \texttt{letregion}\ \varrho\ \texttt{in}\ e''[r_0 \mapsto \bullet])$.

Case $\texttt{copy}\,[\varrho_2, \varrho_2]\, e$. Immediate from the induction hypothesis.

Item 2: Immediate.
Item 3: Suppose that $\mathcal{Q}_v(S, V, s', v')$. Then it holds that $\mathcal{Q}_s(S, s')$ and item 2 yields the assumption about the substituted store. There are two cases for the value.

If $V = (r, l)$ then $v' = (r, l)$. If $r_0 \neq r$ then $(r, l)[r_0 \mapsto \bullet] = (r, l)$, so that $\mathcal{Q}_v(S - r_0, (r, l), s'[r_0 \mapsto \bullet] - r_0, (r, l))$ holds. If $r_0 = r$ then $(r, l)[r_0 \mapsto \bullet] = (\bullet, l)$, so that $\mathcal{Q}_v(S - r_0, (r, l), s'[r_0 \mapsto \bullet] - r_0, (\bullet, l))$ holds.

If $v' = (\bullet, l)$ then v' is unaffected by the substitution. Hence, the claim holds. $\square$

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

An IRC configuration s, e *becomes stuck* iff $s, e \twoheadrightarrow_i s', e'$ and s', e' is stuck.

Since there is no notion of stuck term in a big-step semantics, we extend the TTRC semantics with error reductions in figure 9; the canonical rules for error propagation are omitted.

The following lemma relates the notions of stuck term and error.

Lemma 18 *If $\mathcal{Q}(S, VE, R, e, s, e')$ and $S, VE, R \vdash_b e \Downarrow \mathsf{err}$ then s, e' becomes stuck.*

Proof: Easy induction on the derivation of $S, VE, R \vdash_b e \Downarrow \mathsf{err}$. $\square$

Now, we can formulate the equivalence theorem:

Theorem 3 *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 $s, e' \twoheadrightarrow_i s', v'$ and $\mathcal{Q}_v(S', v, s', v')$.*

2. *If $s_1, e' \twoheadrightarrow_i s_2, v'$ and $\mathcal{Q}(S_1, VE, R, e, s_1, e')$, then $S_1, VE, R \vdash_b e \Downarrow v, S_2$ for which $\mathcal{Q}_v(S_2, v, s_2, v')$.*

$$\textit{(b-const-err)} \qquad \frac{\varrho \notin \mathit{dom}(R)}{S, \mathit{VE}, R \vdash_b c \; \texttt{at} \; \varrho \Downarrow \mathsf{err}}$$

$$\textit{(b-var-err)} \qquad \frac{x \notin \mathit{dom}(\mathit{VE})}{S, \mathit{VE}, R \vdash_b x \Downarrow \mathsf{err}}$$

$$\textit{(b-abstr-err)} \qquad \frac{\varrho \notin \mathit{dom}(R)}{S, \mathit{VE}, R \vdash_b \lambda x.\, e \; \texttt{at} \; \varrho \Downarrow \mathsf{err}}$$

$$\textit{(b-app-err)} \qquad \frac{\begin{array}{c} S, \mathit{VE}, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1 \;\; \mathit{and} \\ (r \notin \mathrm{dom}(S_1) \;\; \mathit{or} \;\; S_1(r_1)(l_1) \neq \langle x, e, \mathit{VE}', R' \rangle) \end{array}}{S, \mathit{VE}, R \vdash_b e_1 \,@\, e_2 \Downarrow \mathsf{err}}$$

$$\textit{(b-copy-err)} \qquad \frac{\begin{array}{c} \{\varrho, \varrho'\} \not\subseteq \mathrm{dom}(R) \;\; \mathit{or} \\ (S, \mathit{VE}, R \vdash_b e \Downarrow (r, l), S' \;\; \mathit{and} \;\; (r' \notin \mathrm{dom}(S') \;\; \mathit{or} \;\; S'(r)(l) \neq \langle c \rangle)) \end{array}}{S, \mathit{VE}, R \vdash_b \mathsf{copy}\, [\varrho, \varrho']\, e \Downarrow \mathsf{err}}$$

Figure 9: Extension of TTRC big-step evaluation with errors

Part 1 of theorem 3 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 untypeable expressions, the TTRC semantics is slightly less deterministic than the PIRC semantics. Consider the following example:

$$\begin{array}{l} \texttt{letregion}\; \varrho \;\texttt{in} \\ \quad \texttt{let}\; f = \texttt{letregion}\; \varrho_1 \;\texttt{in}\; \lambda x.\, (5 \;\texttt{at}\; \varrho_1) \;\texttt{at}\; \varrho \;\texttt{in} \\ \qquad \texttt{letregion}\; \varrho_2 \;\texttt{in}\; f \,@\, (3 \;\texttt{at}\; \varrho_2) \end{array}$$

This term definitely gets stuck in PIRC since the region allocated for ϱ_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 ϱ_1 in the definition of f is re-allocated to ϱ_2 in the last line.

This subtle difference between the two semantics becomes irrelevant for well-typed terms as can be seen from the following corollary of theorem 2 and theorem 3:

Corollary 1 *Suppose* $[\,] \vdash_{tt} e : \mu, \varphi$. *If* $[\,], [\,], [\,] \vdash_b e \Downarrow v, S$ *and* $\mathcal{Q}([\,], [\,], [\,], e, [\,], e')$ *then there exist some* s, v' *such that* $[\,], e' \twoheadrightarrow_i s, v'$ *and* $\mathcal{Q}_v(S, v, s, v')$.

Observe that if e is a TTRC term, then e is also a pure PIRC term such that $\mathcal{Q}([\,], [\,], [\,], e, [\,], e)$ holds.

A second corollary of theorem 2 and theorem 3 and lemma 18 implies type safety of TTRC.

Corollary 2 $[\,] \vdash_{tt} e : \mu, \varphi$ *implies* $[\,], [\,], [\,] \vdash_b e \not\Downarrow \mathsf{err}$.

Although IRC is more deterministic, TTRC is closer to what happens in an actual implementation.

Proof: **(part 1 of theorem 3)** By induction on the derivation of $S, VE, R \vdash_b e \Downarrow v, S'$.

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

Subcase $e' = c \texttt{ at } r$. Then $\mathcal{Q}_s(S, s)$, so that $l \notin \mathrm{dom}(s(r))$. In consequence, $s, c \texttt{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r,l)$. It also holds that $\mathcal{Q}_v(S + \{r \mapsto S(r) + \{l \mapsto \langle c \rangle\}\}, (r,l), s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r,l))$.

Subcase $e' = c \texttt{ at } \bullet$. This term is stuck.

Case $\dfrac{VE(x) = V}{S, VE, R \vdash_b x \Downarrow V, S}$. Let s, e' be such that $\mathcal{Q}(S, VE, R, x, s, e')$. It must be that $e' = v'$ where $\mathcal{Q}_v(S, V, s, v')$, which proves the claim.

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

Subcase $e'' = \lambda\, x.\, e' \texttt{ at } r$. Then $\mathcal{Q}(S, VE - x, R', e, s, e')$ and $\mathcal{Q}_s(S, s)$. Hence, $l \notin \mathrm{dom}(s(r))$ so that $s, \lambda\, x.\, e' \texttt{ at } r \rightarrow_i s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}, (r,l)$. From the assumptions it is immediate that $\mathcal{Q}_v(S + \{r \mapsto S(r) + \{l \mapsto \langle x, e, VE, R \rangle\}\}, (r,l), s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}, (r,l))$.

Subcase $e'' = \lambda\, x.\, e' \texttt{ at } \bullet$. This term is stuck.

Case $\dfrac{\begin{array}{c} S, VE, R \vdash_b e_1 \Downarrow (r_1, l_1), S_1 \\ S_1(r_1)(l_1) = \langle x, e, VE', R' \rangle \\ S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2 \\ S_2, VE' + \{x \mapsto v_2\}, R' \vdash_b e \Downarrow v, S' \end{array}}{S, VE, R \vdash_b e_1 \mathbin{@} e_2 \Downarrow v, S'}$.
Let s, e'' be such that $\mathcal{Q}(S, VE, R, e_1 \mathbin{@} e_2, s, e'')$. Hence, $\mathcal{Q}_s(S, s)$ and $\mathcal{Q}(S, VE, R, e_1, s, e_1')$ and $\mathcal{Q}(S, VE, R, e_2, s, e_2')$ and $e'' = e_1' \mathbin{@} e_2'$.

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 $s, e_1' \mathbin{@} e_2'$ becomes also stuck) or there exist s_1', v_1' such that $s, e_1' \twoheadrightarrow_i s_1', v_1'$ and $\mathcal{Q}_v(S_1, (r_1, l_1), s_1', v_1')$.

By induction on $S_1, VE, R \vdash_b e_2 \Downarrow v_2, S_2$ (using lemmas 13 and 14 to establish $\mathcal{Q}$) either s_1', e_2' becomes stuck (in which case $s_1', v_1' \mathbin{@} e_2'$ becomes also stuck) or there exist s_2', v_2' such that $s_1', e_2' \twoheadrightarrow_i s_2', v_2'$ and $\mathcal{Q}_v(S_2, v_2, s_2', v_2')$.

From the definition of $\mathcal{Q}_v(S_1, (r_1, l_1), s_1, v_1')$, there are two cases for v_1'. Either $v_1' = (\bullet, l_1)$ and then the IRC configuration $s_2', v_1' \mathbin{@} v_2'$ is stuck; or $v_1' =$

(r_1, l_1) and since $S_1(r_1)(l_1) = \langle x, e, \mathit{VE}', R'\rangle$ it must be that $s'_1(r_1)(l_1) = \lambda\, x.\, e'$ where $\mathcal{Q}(S_1, \mathit{VE}' - x, R', e, s'_1, e')$. Hence, $s'_2, v'_1 \mathbin{@} v'_2 \to_i s'_2, e'[x \mapsto v'_2]$. By lemmas 13 and 14 it holds that $\mathcal{Q}(S_2, \mathit{VE}' - x, R', e, s'_2, e')$. By Lemma 15 it holds that $\mathcal{Q}(S_2, \mathit{VE}' + \{x \mapsto v_2\}, R', e, s'_2, e'[x \mapsto v'_2])$.

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

Putting the parts together, either $s, e'_1 \mathbin{@} e'_2$ becomes stuck or $s, e'_1 \mathbin{@} e'_2 \twoheadrightarrow_i s', v'$ where $\mathcal{Q}_v(S', v, s', v')$.

Case $\dfrac{r \notin \mathrm{dom}(S) \quad S + \{r \mapsto [\,]\}, \mathit{VE}, R + \{\varrho \mapsto r\} \vdash_b e \Downarrow v, S'}{S, \mathit{VE}, R \vdash_b \texttt{letregion}\ \varrho\ \texttt{in}\ e \Downarrow v, S' - r}$. Let s, e'' be such that $\mathcal{Q}(S, \mathit{VE}, R, \texttt{letregion}\,\varrho\,\texttt{in}\,e, s, e'')$. Hence, $e'' = \texttt{letregion}\,\varrho\,\texttt{in}\,e'$ and $\mathcal{Q}_s(S, s)$ and $\mathcal{Q}(S, \mathit{VE}, R - \varrho, e, s, e')$. Hence, $s, \texttt{letregion}\ \varrho\ \texttt{in}\ e' \to_i s + \{r \mapsto [\,]\}, \texttt{region}\ r\ \texttt{in}\ e'[\varrho \mapsto r]$. By Lemma 16, it holds that $\mathcal{Q}(S + \{r \mapsto [\,]\}, \mathit{VE}, R + \{\varrho \mapsto r\}, e, s + \{r \mapsto [\,]\}, e'[\varrho \mapsto r])$.

By induction, either $s + \{r \mapsto [\,]\}, e'[\varrho \mapsto r]$ becomes stuck (in which case $s + \{r \mapsto [\,]\}, \texttt{region}\,r\,\texttt{in}\,e'[\varrho \mapsto r]$ becomes stuck, too) or there exist s', v' such that $s + \{r \mapsto [\,]\}, e'[\varrho \mapsto r] \twoheadrightarrow_i s', v'$ and $\mathcal{Q}_v(S', v, s', v')$. By examination of evaluation contexts, $s + \{r \mapsto [\,]\}, \texttt{region}\ r\ \texttt{in}\ e'[\varrho \mapsto r] \twoheadrightarrow_i s', \texttt{region}\,r\,\texttt{in}\,v'$. But the last configuration is a redex: $s', \texttt{region}\,r\,\texttt{in}\,v' \to_i s'[r \mapsto \bullet] - r, v'[r \mapsto \bullet]$. By Lemma 17, 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 3**) By induction on the number of steps in $s_1, e' \twoheadrightarrow_i s_2, v'$. We consider only some non-trivial cases.

Case $s_1, \texttt{letregion}\ \varrho\ \texttt{in}\ e' \to_i s_1 + \{r \mapsto [\,]\}, \texttt{region}\ r\ \texttt{in}\ e[\varrho \mapsto r] \twoheadrightarrow_i s_2, v'$ where $r \notin \mathrm{dom}(s_1)$.

Then $s_1 + \{r \mapsto [\,]\}, e[\varrho \mapsto r] \twoheadrightarrow_i s_3, v'_1$ where $s_3[r \mapsto \bullet] - r = s_2$ and $v'_1[r \mapsto \bullet] = v'$, so that $s_3, \texttt{region}\ r\ \texttt{in}\ v'_1 \to_i s_2, v'$ is the last step.

Suppose $\mathcal{Q}(S_1, \mathit{VE}, R, \texttt{letregion}\ \varrho\ \texttt{in}\ e, s_1, \texttt{letregion}\ \varrho\ \texttt{in}\ e')$, then lemma 16 implies $\mathcal{Q}(S_1 + \{r \mapsto [\,]\}, \mathit{VE}, R + \{\varrho \mapsto r\}, e, s_1 + \{r \mapsto [\,]\}, e'[\varrho \mapsto r])$.

By induction hypothesis, $S_1 + \{r \mapsto [\,]\}, \mathit{VE}, R + \{\varrho \mapsto r\} \vdash_b e \Downarrow v_1, S_3$ for which $\mathcal{Q}_v(S_3, v_1, s_3, v'_1)$.

By rule *(b-letregion)* $S_1, \mathit{VE}, R \vdash_b \texttt{letregion}\ \varrho\ \texttt{in}\ e \Downarrow v_1, S_3 - r$, and $\mathcal{Q}_v(S_3 - r, v_1, s_2, v')$ follows from lemma 17.

Case $s_1, e'_1 \mathbin{@} e'_2 \twoheadrightarrow_i s_4, v'$. Then $s_1, e'_1 \mathbin{@} e'_2 \twoheadrightarrow_i s_2, v'_1 \mathbin{@} e'_2 \twoheadrightarrow_i s_3, v'_1 \mathbin{@} v'_2 \to_i s_3, e'[x \mapsto v'_2] \twoheadrightarrow_i s_4, v'$, where $v'_1 = (r, l)$ and $s_3(r)(l) = \langle \lambda\, x.\, e'\rangle$ and $s_1, e'_1 \twoheadrightarrow_i s_2, v'_1$ and $s_2, e'_2 \twoheadrightarrow_i s_3, v'_2$. Note that all the intermediate $\twoheadrightarrow_i$ have strictly fewer steps than $s_1, e'_1 \mathbin{@} e'_2 \twoheadrightarrow_i s_4, v'$.

Suppose $\mathcal{Q}(S_1,\, VE, R, e_1 \,@\, e_2, s_1, e'_1 \,@\, e'_2)$, then $\mathcal{Q}(S_1,\, VE, R, e_i, s_1, e'_i)$ for $i = 1, 2$.

By induction hypothesis, $S_1,\, VE, R \vdash_b e_1 \Downarrow v_1, S_2$ for which $\mathcal{Q}_v(S_2, v_1, s_2, v'_1)$.

Since $S_1 \leq S_2$ and $s_1 \leq s_2$ (by lemma 13) and $\mathrm{dom}(S_2) = \mathrm{dom}(s_2)$, lemma 14 gives $\mathcal{Q}(S_2,\, VE, R, e_2, s_2, e'_2)$.

By induction hypothesis, $S_2,\, VE, R \vdash_b e_2 \Downarrow v_2, S_3$ for which $\mathcal{Q}_v(S_3, v_2, s_3, v'_2)$.

By lemma 14, $\mathcal{Q}_v(S_3, v_1, s_3, v'_1)$, hence we have $S_3(r)(l) = \langle x, e,\, VE_1, R_1\rangle$ with $\mathcal{Q}(S_3,\, VE_1 - x, R_1, e, s_3, e')$.

Lemma 15 implies $\mathcal{Q}(S_3,\, VE_1 + \{x \mapsto v_2\}, R_1, e, s_3, e'[x \mapsto v'_2])$.

By induction hypothesis, $S_3, VE_1 + \{x \mapsto v_2\}, R_1 \vdash_b e \Downarrow v, S_4$ for which $\mathcal{Q}_v(S_4, v, s_4, v')$.

Finally, rule *(b-app)* yields $S_1,\, VE, R \vdash_b e_1 \,@\, e_2 \Downarrow v, S_4$.

□

5.2 Equivalence between PIRC and SRC

Proving equivalence between IRC and SRC is considerably simpler because we are dealing with two small-step semantics. The proof boils down to a simple bisimulation argument 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 *not* alpha-convertible. The explicit store 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 IRC configurations and SRC terms via a relation

$$\mathcal{R}_\alpha \subseteq (\mathit{PIRC\text{-}Store} \times \; \mathit{PIRC\text{-}Terms})_\alpha \times \; \mathit{SRC\text{-}Terms}$$

where $\alpha : \mathit{RegionVars} \to \mathit{RegionNames}$ is an injective map and $(\mathit{PIRC\text{-}Store} \times \mathit{PIRC\text{-}Terms})_\alpha$ is the set of configurations s, e such that:

- $\mathrm{dom}(s) = \mathrm{ran}(\alpha)$
- $\mathit{frn}\,(s, e) \subseteq \mathrm{ran}(\alpha)$ with $\mathit{frn}\,(-)$ the set of free region names.

The required relation $\mathcal{R}_\alpha$ is defined by the axioms listed in Figure 10.

We formulate a few lemmas:

Lemma 19 *If* $\mathcal{R}_{\alpha+\{\varrho\mapsto r\}}(s, e, e')$ *then* $\mathcal{R}_\alpha(s[r \mapsto \bullet] - r, e[r \mapsto \bullet], e'[\varrho \mapsto \bullet])$.

Lemma 20 *If* $\mathcal{R}_\alpha(s, e, e')$ *and* $\varrho \notin \mathrm{dom}(\alpha)$ *and* $r \notin \mathrm{ran}(\alpha)$ *then* $\mathcal{R}_{\alpha+\{\varrho\mapsto r\}}(s, e, e')$.

$$
\begin{array}{lcl}
\mathcal{R}_\alpha(s,(r,l),\langle c\rangle_\varrho) & \Leftrightarrow & s(r)(l)=\langle c\rangle \wedge \alpha(\varrho)=r \\
\mathcal{R}_\alpha(s,(\bullet,l),\langle c\rangle_\bullet) & & \\
\mathcal{R}_\alpha(s,(r,l),\langle \lambda\, x.\, e\rangle_\varrho) & \Leftrightarrow & s(r)(l)=\langle \lambda\, x.\, e'\rangle \wedge \mathcal{R}_\alpha(s,e',e) \wedge \alpha(\varrho)=r \\
\mathcal{R}_\alpha(s,(\bullet,l),\langle \lambda\, x.\, e\rangle_\bullet) & & \\
\mathcal{R}_\alpha(s,c\ \mathtt{at}\ r, c\ \mathtt{at}\ \varrho) & \Leftrightarrow & \alpha(\varrho)=r \\
\mathcal{R}_\alpha(s,c\ \mathtt{at}\ \bullet, c'\ \mathtt{at}\ \bullet) & & \\
\mathcal{R}_\alpha(s,\lambda\, x.\, e\ \mathtt{at}\ r, \lambda\, x.\, e'\ \mathtt{at}\ \varrho) & \Leftrightarrow & \mathcal{R}_\alpha(s,e,e') \wedge \alpha(\varrho)=r \\
\mathcal{R}_\alpha(s,\lambda\, x.\, e\ \mathtt{at}\ \bullet, \lambda\, x.\, e'\ \mathtt{at}\ \bullet) & & \\
\mathcal{R}_\alpha(s,x,x) & & \\
\mathcal{R}_\alpha(s,e_1 \mathbin{@} e_2, e_1' \mathbin{@} e_2') & \Leftrightarrow & \mathcal{R}_\alpha(s,e_1,e_1') \wedge \mathcal{R}_\alpha(s,e_2,e_2') \\
\mathcal{R}_\alpha(s,\mathtt{letregion}\ \varrho\ \mathtt{in}\ e, \mathtt{letregion}\ \varrho\ \mathtt{in}\ e') & \Leftrightarrow & \mathcal{R}_{\alpha+\{\varrho\mapsto r\}}(s+\{r\mapsto [\,]\}, e[\varrho\mapsto r], e') \\
 & & \quad \textit{for each } r \notin \mathrm{ran}(\alpha) \\
\mathcal{R}_\alpha(s,\mathtt{region}\ r\ \mathtt{in}\ e, \mathtt{letregion}\ \varrho\ \mathtt{in}\ e') & \Leftrightarrow & \alpha(\varrho)=r \wedge \mathcal{R}_\alpha(s,e,e')
\end{array}
$$

Figure 10: Relation between PIRC and SRC terms

Lemma 21 *If $\mathcal{R}_\alpha(s,e_1,e_1')$ and $\mathcal{R}_\alpha(s,e_2,e_2')$ then $\mathcal{R}_\alpha(s,e_1[x\mapsto e_2],e_1'[x\mapsto e_2'])$.*

We write $\rightarrow_i^{1,2}$ to mean one or two reduction steps.

The equivalence between SRC and IRC is a result of the following theorem

Theorem 4 *Suppose $\mathcal{R}_\alpha(s_1,e_1,e_1')$.*

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

Proof: The proof is a case analysis on the definition of $\rightarrow_i$ and $\rightarrow_s$. □

6 Conclusions

We have considered three different operational semantics of "the region calculus": the original one [15], a store-less definition [10], and a novel store-based definition, which extends the others with operations on references. Both the store-less and the store-based definitions are given as small-step operational semantics. We have shown that all formulations are essentially equivalent. We have further shown that the store-less as well as the store-based operational semantics admit a syntactic type soundness result with respect to the original static semantics.

The paper by Calcagno [5] gives yet another approach to a type soundness result, using a store-less big-step operational semantics, which is parametric in a set φ of currently allocated regions. In that setting, the approach is essentially equivalent to using a special constant $\bullet$ for dead regions: in fact it is possible to relate the two semantics by renaming the regions outside φ 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. Only the static semantics prevents TTRC from doing so. Hence, TTRC is closer to an actual implementation.

References

[1] *Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'01)*, Imperial College, London, January 2001.

[2] 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*, Trento, Italy, July 1999. IEEE Computer Society Press.

[3] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, to appear, 1999.

[4] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 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.

[5] 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*, London, England, January 2001. ACM Press.

[6] Karl Crary, 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.

[7] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume I. North Holland, 1958.

[8] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [9].

[9] Robert Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See [8].

[10] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *HOOTS '00 Higher Order Operational Techniques in Semantics*, Electronic Notes in Theoretical Computer Science, Montreal, Canada, September 2000. Elsevier Science.

[11] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[12] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[13] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.

[14] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-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, 15 November 1994.

[17] S. Dal Zilio and Andrew Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of MFCS '00*, Lecture Notes in Computer Science, 2000.