

**REGIONS AND CONTROL**

by

**MILEY EDWARD SEMMELROTH**

**A DISSERTATION**

**Presented to the Department of Computer  
and Information Science  
and the Graduate School of the University of Oregon  
in partial fulfillment of the requirements  
for the degree of  
Doctor of Philosophy**

**March 2003**

“Regions and Control,” a dissertation prepared by Miley Edward Semmelroth in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:



---

Dr. Zena M. Ariola, Chair of the Examining Committee

3-10-2003

Date

Committee in charge:

Dr. Zena M. Ariola, Chair  
Dr. Andrzej Proskurowski  
Dr. Marie Vitulli  
Dr. Michal Young

Accepted by:



---

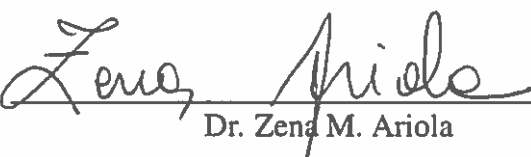
Dean of the Graduate School

An Abstract of the Dissertation of  
Miley Edward Semmelroth for the degree of Doctor of Philosophy  
in the Department of Computer and Information Science

to be taken

March 2003

Title: REGIONS AND CONTROL

Approved:   
Dr. Zena M. Ariola

Region analysis provides a conservative approximation of the lifetimes of objects in higher-order programs. Stemming from earlier work on type and effect calculi, the region system proposed by Tofte and Talpin has led to an implementation of Standard ML based entirely on static memory management.

Extensive research has focused on the optimization and extension of region systems, as well as the subtle problem of formalizing their correctness. Although the classical region calculus of Tofte and Talpin scales suitably to most of the constructs of modern functional languages, relatively little is known about its relation to control mechanisms such as exception handling and the manipulation of reified continuations.

We present a simple and scalable operational framework for proving the correctness of region systems based on a variant of the Tofte-Talpin calculus. We then define two extensions to this framework which serve to both clarify and formalize the interaction between region systems and constructs for generative exceptions and first-class continuations.

## CURRICULUM VITA

NAME OF AUTHOR: Miley Edward Semmelroth

PLACE OF BIRTH: Washington, D.C.

DATE OF BIRTH: January 17, 1970

### GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon  
Western Michigan University

### DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science,  
2003, University of Oregon  
Master of Science in Computer Science,  
1995, Western Michigan University  
Bachelor of Science in Mathematics,  
1993, Western Michigan University

### AREAS OF SPECIAL INTEREST:

Programming Language Semantics  
Type Theory

### PROFESSIONAL EXPERIENCE:

Research and Teaching Assistant,  
Department of Computer Information Science,  
University of Oregon, Eugene, 1995-2002  
  
Teaching Assistant,  
Department of Computer Science,  
Western Michigan University, Kalamazoo, 1994-95

## ACKNOWLEDGEMENTS

I owe many thanks both to my advisor, Zena, and to my parents, Carl and Sara, for their patience and support during my graduate-student life.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION . . . . .	1
History and Background . . . . .	2
Summary of Contributions . . . . .	4
II. TYPES AND EFFECTS . . . . .	6
Milner's System . . . . .	6
Imperative Features . . . . .	18
Typing References with Effects . . . . .	24
III. REGIONS . . . . .	32
Static Memory Management . . . . .	32
The Region Calculus . . . . .	33
The Language $\mathcal{RL}$ . . . . .	38
Dynamic Semantics . . . . .	42
Static Semantics . . . . .	45
Discussion . . . . .	50
IV. EXCEPTIONS . . . . .	53
Exceptions in the ML-Kit . . . . .	53
Type and Effect Rules for Exceptions . . . . .	56
The Language $\mathcal{RE}$ . . . . .	58
Dynamic Semantics . . . . .	61
Static Semantics . . . . .	64
V. SOUNDNESS OF $\mathcal{RE}$ . . . . .	70
Basic Properties . . . . .	70
Preservation and Progress . . . . .	75
VI. CONTINUATIONS . . . . .	98
Programming with Continuations . . . . .	98
The Problem . . . . .	99

	Page
Effects vs. Capabilities . . . . .	100
Typing Continuations . . . . .	107
The Language $\mathcal{RC}$ . . . . .	109
Dynamic Semantics . . . . .	110
Static Semantics . . . . .	113
 VII. SOUNDNESS OF $\mathcal{RC}$ . . . . .	 118
Basic Properties . . . . .	118
Preservation and Progress . . . . .	120
 BIBLIOGRAPHY . . . . .	 137

## LIST OF FIGURES

Figure	Page
1. Transition Rules for Expression Language . . . . .	9
2. Milner Calculus . . . . .	14
3. Transition Rules for References . . . . .	19
4. Typing Rules for References . . . . .	20
5. Selected Typing Rules for Machine States . . . . .	23
6. Talpin-Jouvelot Calculus (Part 1) . . . . .	27
7. Talpin-Jouvelot Calculus (Part 2) . . . . .	28
8. Tofte-Talpin Calculus Variant . . . . .	36
9. Transition Rules for $\mathcal{RL}$ . . . . .	43
10. Static Semantics for $\mathcal{RL}$ (Part 1) . . . . .	46
11. Static Semantics for $\mathcal{RL}$ (Part 2) . . . . .	47
12. Static Semantics for $\mathcal{RL}$ (Part 3) . . . . .	48
13. Static Semantics for $\mathcal{RL}$ (Part 4) . . . . .	49
14. ML-Kit Example . . . . .	54
15. Type and Effect Rules for Exceptions . . . . .	57
16. Transition Rules for $\mathcal{RE}$ (Part 1) . . . . .	62
17. Transition Rules for $\mathcal{RE}$ (Part 2) . . . . .	63
18. Static Semantics for $\mathcal{RE}$ (Part 1) . . . . .	65
19. Static Semantics for $\mathcal{RE}$ (Part 2) . . . . .	67
20. Static Semantics for $\mathcal{RE}$ (Part 3) . . . . .	68
21. Static Semantics for $\mathcal{RE}$ (Part 4) . . . . .	69



	Page
22. Static Semantics for $\mathcal{RE}$ (Part 5) . . . . .	69
23. SMI/NJ Example . . . . .	99
24. Simple Effect-Based Region System . . . . .	102
25. Simple Capability-Based Region System . . . . .	103
26. Transition Rules for $\mathcal{RC}$ (Part 1) . . . . .	111
27. Transition Rules for $\mathcal{RC}$ (Part 2) . . . . .	112
28. Static Semantics for $\mathcal{RC}$ (Part 1) . . . . .	114
29. Static Semantics for $\mathcal{RC}$ (Part 2) . . . . .	115
30. Static Semantics for $\mathcal{RC}$ (Part 3) . . . . .	116
31. Static Semantics for $\mathcal{RC}$ (Part 4) . . . . .	117

## CHAPTER I

### INTRODUCTION

The study of type systems has become a ubiquitous aspect of programming language theory. Fundamentally, these systems provide structure by assigning *types* as static descriptions to terms which serve as the building blocks for a language. Although type systems are often understood as a means of language definition which can eliminate potentially ill-behaved programs, typing disciplines have found a number of other uses in compiler construction.

An important family of augmented type systems has been based on embellishing type descriptions with *effects*. Just as types can be seen as classifiers of terms, effects provide more detailed classifications by conservatively approximating the potential for program fragments to perform imperative computations. Systems for assigning both type and effect descriptions to terms are typically cast in a functional language setting with imperative features such as mutable state, exceptions, first-class continuations, and concurrency. Type and effect descriptions are often mutually defined with the inter-related concept of *regions*. Regions may be regarded as static names for partitions of unbounded size within the run-time store.

Several variants of type, effect, and region systems have been proposed with differing goals. In this chapter, we provide context for the present work with a brief survey of the history of these systems. We then summarize the contributions of this thesis and provide a road-map for the remaining chapters.

## History and Background

Most of the essential concepts underlying type and effect systems first appeared in the work of Lucassen and Gifford on the FX programming language[34]. FX is based on a variant of the second-order lambda-calculus enriched with store operations. The type and effect system for FX is used to detect shared-state dependencies among program fragments for the purpose of efficient automatic parallelization. Although powerful, this system requires that programmers explicitly annotate their code with type information making programs syntactically unruly and complex. Later research therefore focused on the difficult problem of automatically inferring type and effect annotations [27, 42, 48, 55].

The ability to approximate the side-effects performed by program fragments has proven helpful for a variety of program optimizations such as dead code elimination, common subexpression elimination, and hoisting of loop invariants. Effect systems have been incorporated in the TIL and Standard ML/NJ compilers for this purpose[45, 44]. An earlier proposal for a *control effect* system was aimed at optimizing the implementation of first-class continuations[26]. The well-known problem of soundly incorporating effects into the framework of Milner-style polymorphism has also been approached with systems based on effects[13, 42, 43, 57].

As hypothesized by Wadler, effects systems can be alternately cast with a *monadic phrasing* [29, 40, 55]. This analogy has been exploited to provide an elegant framework for incorporating mutable state into the purely functional language Haskell[30, 31]. This analogy also underlies some frameworks for effect dependent optimizing transformations for ML-like languages[6, 53].

Arguably one of the most visible successes in applying type and effect systems

has been in the implementation of static memory management[48, 50, 51]. In contrast to garbage collectors which rely on *run-time* examinations of the heap, static memory management is based on an analysis that makes all decisions about when memory will be freed at *compile-time*. Region analysis, the process of statically assigning regions to heap-allocated objects in a program, was formalized in the seminal work of Tofte and Talpin[52]. The Tofte-Talpin region calculus is a type and effect system which describes how ML-like programs may safely use regions. This framework, when composed with supplementary analyses, scales to a practical implementation of Standard ML[49].

The work of Tofte and Talpin has led to an abundance of research on region systems. An early extension proposed by Aiken *et al.* was aimed at relaxing the requirement that regions are managed in a strictly last-in-first-out (LIFO) manner[1]. The idealized framework has been further supplemented by analyses such as multiplicity inference and region-representation inference[7, 54]. More recent work has explored the benefits of combining region-based memory management with a conventional copying garbage collector[17]. A region-based system has also been employed in implementation of Cyclone[16], a type-safe dialect of the C language.

In addition to the applied research just outlined, a number of theoretical aspects of region systems have been studied. For example, because the original proof of correctness for the region calculus relied on a complex co-inductive argument, alternative approaches to proving soundness have been proposed[8, 21]. The work of Walker and Watkins explores the combination of regions with linear types[56]. Other authors have explored the relationship between region systems and frameworks such as the polymorphic lambda-calculus, the pi-calculus extended with groups, and monadic state[4, 40, 60]. Finally, the calculus of capabilities, proposed by Cray *et al.*, provides

a means for implementing a type-safe continuation-passing-style (CPS) transformation on the Tofte-Talpin language[11].

### Summary of Contributions

The Tofte-Talpin calculus was originally cast in an idealized setting including only the essential constructs of ML-like languages. Extending this core framework to handle other important features of ML such as references and recursive datatypes can be done with relative ease[52]. Although these language features interact benignly with region inference, a more subtle situation arises with *control* constructs such as Standard ML's generative exceptions, and Standard ML of New Jersey's first-class continuations[3, 36]. Operationally, exceptions and continuations allow "jumps" that are at odds with region inference. Previous work on region systems has not addressed the use of control constructs in relation to the Tofte-Talpin calculus.

We study the interaction between region analysis and control constructs in the setting of a higher-order polymorphic language. The primary contributions of this work are as follows:

1. We define a variant of the Tofte-Talpin calculus and an operational semantics which capture the essential features of region analysis and the intended execution model. Building on previous work on syntactic type-soundness [11, 21, 59], our abstract machine provides a scalable framework for investigating control features.
2. We extend this framework with idealized constructs for generative exceptions similar to those defined by Wright and Felleisen[59]. Although the extension of our region system is simple and intuitive from a typing perspective, verifying its correctness is not. We present a formal proof of soundness for this extension which,

for the first time, demonstrates the correctness of a higher-order region system augmented with a control feature.

3. We show how a direct-style variant of the calculus of capabilities can be employed to accommodate first-class continuations within the region-based execution model[11]. By building again on our operational framework, we formally prove soundness for an extension with ML-like constructs for capturing and invoking continuations.

The remainder of this thesis is organized as follows: In CHAPTER II, we provide a pedagogic presentation of the essential concepts related to types and effects while focusing on the technical evolution of region analysis. In CHAPTER III, we discuss a close variant of the Tofte-Talpin calculus. We then motivate and define the language  $\mathcal{RL}$  which serves as the basis for our study of control constructs. In CHAPTER IV, we define the language  $\mathcal{RE}$ , and extension of  $\mathcal{RL}$  with constructs for generative exceptions. CHAPTER V contains a detailed proof of soundness  $\mathcal{RE}$ . In CHAPTER VI, we contrast approaches based on effects and capabilities and motivate our solution to the problem of typing continuations based on the latter. We then define the language  $\mathcal{RC}$ , an extension of  $\mathcal{RL}$  with constructs for manipulating first-class continuation. Finally, a soundness proof for  $\mathcal{RC}$  is provided in CHAPTER VII.

## CHAPTER II

## TYPES AND EFFECTS

We now introduce the fundamental technical notions underlying the current work and trace the early evolution of the region calculi studied in later chapters. Historically, this evolution is best understood by exploring the properties of Milner's system in relation to computational effects, and in particular, by discussion of the type and effect discipline of Talpin and Jouvelot [43].

Milner's System

At the core of typed functional languages such the ML and its dialects is the Hindley-Milner typing discipline for polymorphic functions[35]. We will discuss the general concepts related to this system in the context of a minimal functional language. Consider the following syntax where  $f$  and  $x$  range over a set of term variables,  $n$  ranges over the integers, and  $op \in \{+, -, =\}$  is a binary operator symbol:

$$\begin{aligned}
 v & ::= x \mid \text{rec } f(x) \Rightarrow e \mid \underline{n} \mid \text{true} \mid \text{false} \\
 e & ::= v \mid e_1 e_2 \mid \text{let } x \text{ be } e_1 \text{ in } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid e_1 \text{ op } e_2 \mid ! e
 \end{aligned}$$

For convenience, the expressions of our language, ranged over by  $e$ , are defined using a separate syntactic class of values, ranged over by  $v$ . Expressions of the form  $\text{rec } f(x) \Rightarrow e$  represent recursive functional abstractions in which the variables  $f$  and  $x$  are bound within  $e$ . The free and bound variables of an expression have standard inductive definitions. For non-recursive functions, that is, whenever  $f$  does not occur free

in  $e$ , we use the abbreviated notation  $\text{fn } x \Rightarrow e$ . Function applications are indicated by juxtaposition, written  $e_1 e_2$ , where  $e_1$  is an expression yielding a function to be applied to the argument  $e_2$ . Expressions of the form  $\text{let } x \text{ be } e_1 \text{ in } e_2$  represent local declarations in which the variable  $x$  is bound within  $e_2$  but not  $e_1$ . Following common notational conventions, we identify terms up to consistent renaming of bound variables and we will freely make use of parentheses to clarify the syntactic structure of expressions. For example, consider the following expression:

$$((\text{fn } x \Rightarrow x) (\text{fn } x \Rightarrow x)) (\underline{2} + \underline{3})$$

Striping is used in order to distinguish numerals, a syntactic notion, from numbers, a semantic notion. The expression above consists of an application of the identity function to itself, the result of which is applied to the argument  $\underline{2} + \underline{3}$ . For brevity, we include only four primitive operations: integer addition and subtraction, integer equality, and logical negation, written  $! e$ . Our language also includes constants for boolean values and a standard form of conditional expression. We note in passing that other common arithmetic and boolean operations can be encoded using the given constructs.

### Dynamic Semantics

Throughout the present work, we consider only *call-by-value* semantics for functions. In other words, we assume that function arguments are evaluated prior to applications. In order to formalize the meanings of expressions, we will make use of a syntactic style of semantics called *transition semantics*. The key idea is to regard the expressions themselves as the states of an abstract machine. A transition relation, defined on the states, provides a deterministic means of making progress in the computation by evalu-



ating a particular subexpression. Although the bare syntax of our expression language is presently sufficient, we shall see later that special syntax is often required to represent machine states. One then distinguishes between *surface expressions*, relevant to the programmer, and *computational expressions*, relevant to the intermediate states of evaluation.

In order to identify a unique subexpression, transition rules will be defined using the auxiliary notion of an *evaluation context*[14]. Evaluation contexts, ranged over by  $E$ , are expressions containing a hole, written  $[\cdot]$ , corresponding to a missing subexpression. We define evaluation contexts for our expression language as follows:

$$E ::= [\cdot] \mid E e \mid v E \mid \text{let } x \text{ be } E \text{ in } e \mid \\ \text{if } E \text{ then } e_1 \text{ else } e_2 \mid E \text{ op } e \mid v \text{ op } E \mid ! E$$

The given contexts will serve to specify a *left-most outer-most* evaluation strategy. Filling the hole of a context  $E$  with an expression  $e$  yields a new expression written  $E[e]$ .

The single-step transition relation,  $\mapsto$ , is defined in FIGURE 1 where  $e[x := v]$  denotes the capture-avoiding substitution of the value  $v$  for the free occurrences of  $x$  in  $e$  and  $\equiv$  is used for syntactic identity. We define the multi-step transition relation,  $\mapsto^*$ , as the reflexive and transitive closure of  $\mapsto$ . Finally, we define an evaluator as a partial function from expressions to values as follows:

$$\text{eval}(e) = v \text{ if and only if } e \mapsto^* v$$

Partiality arises for two reasons. Firstly, the application of a recursive function may result in an infinite reduction sequence. In that case, we say that such an expression *diverges*. A second possibility is that a non-value state may be reached from which no

---


$$E[ (\text{rec } f(x) \Rightarrow e) v ] \mapsto E[ (e[x := v])[f := \text{rec } f(x) \Rightarrow e] ]$$

$$E[ \text{let } x \text{ be } v \text{ in } e ] \mapsto E[ e[x := v] ]$$

$$E[ \text{if true then } e_1 \text{ else } e_2 ] \mapsto E[ e_1 ]$$

$$E[ \text{if false then } e_1 \text{ else } e_2 ] \mapsto E[ e_2 ]$$

$$E[ \underline{n}_1 + \underline{n}_2 ] \mapsto E[ \underline{n}_1 + \underline{n}_2 ]$$

$$E[ \underline{n}_1 - \underline{n}_2 ] \mapsto E[ \underline{n}_1 - \underline{n}_2 ]$$

$$E[ \underline{n}_1 = \underline{n}_2 ] \mapsto E[ \text{true} ] \text{ if } \underline{n}_1 \equiv \underline{n}_2$$

$$E[ \underline{n}_1 = \underline{n}_2 ] \mapsto E[ \text{false} ] \text{ if } \underline{n}_1 \not\equiv \underline{n}_2$$

$$E[ ! \text{true} ] \mapsto E[ \text{false} ]$$

$$E[ ! \text{false} ] \mapsto E[ \text{true} ]$$


---

FIGURE 1. Transition Rules for Expression Language

transition is defined. We shall have more to say about the latter situation shortly.

As an example of the consequences of our semantics, we may define a sequential expression, written  $e_1 ; e_2$ , using *syntactic sugar* as follows:

$$e_1 ; e_2 \triangleq \text{let } x \text{ be } e_1 \text{ in } e_2$$

where  $x$  does not occur free in  $e_2$ . The intention is that first  $e_1$  will be evaluated and its value discarded, then  $e_2$  will be evaluated yielding the value of the whole expression.

Semantically, let-expressions may be regarded as a special form of function application. Informally, the equivalence can be written as follows:

$$\text{let } x \text{ be } e_1 \text{ in } e_2 \approx (\text{fn } x \Rightarrow e_2) e_1$$

However, for the purposes of typing, let-expressions will play a special rôle in allowing the introduction of *polymorphic* functions. Parametric polymorphism is a hallmark of functional programming. Intuitively, parametric polymorphism allows the same functional algorithm to be applied to different types of data. As a trivial example, the following expression binds the polymorphic identity function to the variable  $f$  which is then applied to both boolean and integer constants:

$$\text{let } f \text{ be } (\text{fn } x \Rightarrow x) \text{ in } (f \text{ true} ; f \underline{5})$$

We say that the function bound to  $f$  above is used *heterogeneously* on both integer and boolean values. Perhaps a more convincing example is the following higher-order

function:

$$\text{fn } f \Rightarrow (\text{fn } x \Rightarrow (f \ x))$$

Although a formal discussion of typing is yet to follow, it should be intuitively clear that for applications of the function above to make sense, some appropriate relationship must exist between the function argument named  $f$  and the argument named  $x$ . This relationship must be captured in the type of the expression as a whole.

### Program Errors

The fundamental purpose of a type system is to prevent *type errors*. Consider the following expression:

$$(\text{fn } x \Rightarrow x + \underline{1}) \text{ true}$$

With respect to the semantics introduced earlier, the expression results in a *stuck state*. In other words, after a single transition, the state  $(\text{true} + \underline{1})$  is reached which is neither a value, nor reducible by any transition rule. Thus, our formally defined semantics says nothing about the meaning of this expression. In order to avoid unpredictable behavior in the case of such programs, language systems have historically used two main approaches to this problem: *Dynamic typing*, used for example in the language Scheme and its dialects, seeks to detect and report such situations at *run-time*[28]. *Static typing*, an alternative approach used to some extent in most modern programming languages, employs an analysis to reject programs at *compile-time* if they may cause certain errors.

The dynamic approach has the advantage of affording the greatest flexibility to the programmer; programs are assumed to make sense until an error is actually encountered during execution. Considerable debate has focused on whether this constitutes a virtue

of dynamic typing. For example, consider:

$$\text{if } e \text{ then } \underline{5} \text{ else } (\underline{1} + \text{true})$$

where  $e$  is some unspecified expression. In a dynamically typed system, the expression above will result in a run-time error only when  $e$  evaluates to false. In a statically typed system, the expression above would be rejected outright as nonsensical, even though it may execute and terminate normally. Indeed, it should be clear that even in our simple setting, it is undecidable whether an arbitrary expression will produce an error in the sense of resulting in a stuck state. Therefore the design of a type system involves a tradeoff; one may want to be as permissive as possible while retaining the algorithmic decidability of the analysis.

Pragmatically, one can often side-step computational intractability by requiring programmers to provide type information to guide the analysis. In the context of polymorphic functional languages, a variety of systems with and without explicit type information have been studied [5]. Although type annotations may be regarded positively as a form of documentation, they may also be considered too cumbersome for programmers. *Type-inference* is the process of determining the type, if one exists, of an un-annotated expression based on the rules of a type system.<sup>1</sup> The work of Hindley, Milner, and Damas led to the development of a system for typing higher-order polymorphic functions which provides an appealing degree of expressive power while still allowing a practical algorithm for type-inference [12, 23, 35]. This system has served as the backbone of the statically typed functional languages Haskell and ML[25, 37].

---

<sup>1</sup>Note that this problem subsumes the question of whether an expression is well-formed.

## Static Semantics

We now turn to the formalization of a Milner-style type system. Expressions will be assigned (mono-) types, ranged over by  $\tau$ , of the following forms where  $\alpha$  ranges over a set of type variables:

$$\tau ::= \alpha \mid \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2$$

Base types, `int` and `bool`, will play an obvious rôle. Functions will be assigned arrow types, written  $\tau_1 \rightarrow \tau_2$ , where  $\tau_1$  and  $\tau_2$  are the domain and codomain types respectively. Type variables are place-holders for an unspecified type. Type schemes, ranged over by  $\pi$ , represent a set of types obtained by substitution for a sequence of quantified type variables:

$$\pi ::= \tau \mid \forall \vec{\alpha}. \tau$$

For convenience, types schemes are defined to include monotypes. The free and bound type variables of a type scheme have standard inductive definitions and we write  $ftv(\pi)$  for the set of free type variables occurring in  $\pi$ . Type schemes are identified up to consistent renaming and reordering of bound type variables. Throughout the present work, we will use vector notation, such as  $\vec{\alpha}$ , to represent sequences of *distinct* bound variables.

We write  $\tau[\alpha_1 := \tau_1, \dots, \alpha_k := \tau_k]$  for the simultaneous substitution of  $\tau_i$  for  $\alpha_i$  in  $\tau$  for each  $1 \leq i \leq k$ . We say that the type scheme  $\pi$  subsumes or generalizes the type  $\tau$ , written  $\pi \succ \tau$ , if and only if  $\pi \equiv \tau$  or  $\pi \equiv \forall \vec{\alpha}. \tau'$  and  $\tau$  can be obtained from  $\tau'$  via substitution for zero or more type variables occurring in  $\vec{\alpha}$ .

Type environments, ranged over by  $\Gamma$ , are finite maps from term variables to type

---


$$\frac{\Gamma(x) \succ \tau}{\Gamma \triangleright x : \tau}$$

$$\Gamma \triangleright \underline{n} : \text{int}$$

$$\Gamma \triangleright \text{true} : \text{bool}$$

$$\Gamma \triangleright \text{false} : \text{bool}$$

$$\frac{\Gamma \langle f : \tau_1 \rightarrow \tau_2 \rangle \langle x : \tau_1 \rangle \triangleright e : \tau_2}{\Gamma \triangleright \text{rec } f(x) \Rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \triangleright e_2 : \tau_1}{\Gamma \triangleright e_1 e_2 : \tau_2}$$

$$\frac{\{\bar{\alpha}\} \cap \text{ftv}(\Gamma) = \emptyset \quad \Gamma \triangleright e_1 : \tau_1 \quad \Gamma \langle x : \forall \bar{\alpha}. \tau_1 \rangle \triangleright e_2 : \tau_2}{\Gamma \triangleright \text{let } x \text{ be } e_1 \text{ in } e_2 : \tau_2}$$

$$\frac{\Gamma \triangleright e_1 : \text{bool} \quad \Gamma \triangleright e_2 : \tau \quad \Gamma \triangleright e_3 : \tau}{\Gamma \triangleright \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

$$\frac{\Gamma \triangleright e_1 : \text{int} \quad \Gamma \triangleright e_2 : \text{int}}{\Gamma \triangleright e_1 \text{ op } e_2 : \tau_b}$$

where  $\tau_b \equiv \text{int}$  if  $\text{op} \in \{+, -\}$  and  $\tau_b \equiv \text{bool}$  otherwise

$$\frac{\Gamma \triangleright e : \text{bool}}{\Gamma \triangleright !e : \text{bool}}$$


---

FIGURE 2. Milner Calculus

schemes. We write  $dom(\Gamma)$  for the domain of  $\Gamma$  and use the notation  $\Gamma\langle x : \pi \rangle$  for functional updates. We use  $\langle \rangle$  for the empty map. Type environments are used to carry assumptions about the types of the free term variables occurring in expressions. We write  $ftv(\Gamma)$  for the set of free type variables occurring in all of the type schemes in the range of  $\Gamma$ .

The type system is defined by *judgments* of the form  $\Gamma \triangleright e : \tau$  asserting that the expression  $e$  has type  $\tau$  given the assumptions in  $\Gamma$ . A judgment is valid if it can be derived from the axioms and inference rules defined in FIGURE 2. Each of the phrases of our language has a corresponding rule in the type system. Axioms, that is, typing rules without premises, are given for numerals and boolean constants. Term variables may be assigned any type which is subsumed by the assumption given the type environment. Each of the remaining rules involve premises corresponding to the subexpressions of the given phrase. The rules for function applications, conditional expressions, and the primitive operations are straightforward. For each of the binding constructs, functions and let-expressions, the type environment is extended with assumptions about the types of the bound variables.

In order for a function to be used heterogeneously, it must be bound to a term variable. The key idea underlying Milner's system is to allow only let-bound variables to be used this way. Thus, the typing rule for functions allows only monotypes to be added to the typing environment. In contrast, the rule for let-expressions allows for the type of the let-bound expression to be generalized. The criterion for generalizability requires that the quantified type variables not occur free in the type schemes of any assumptions in the typing environment.<sup>2</sup> To clarify this condition, it may be helpful to

---

<sup>2</sup>This proviso should be familiar from universal quantification in first-order logic.



consider the following (incorrect) derivation:

$$\frac{\langle g : \alpha \rightarrow \alpha \rangle \not\vdash \text{let } f \text{ be } g \text{ in } (f \text{ true}; f \underline{5}) : \text{int}}{\langle \rangle \not\vdash \text{fn } g \Rightarrow (\text{let } f \text{ be } g \text{ in } (f \text{ true}; f \underline{5})) : (\alpha \rightarrow \alpha) \rightarrow \text{int}}$$

Without the restriction on generalization, the let-bound identifier  $f$  above could be assigned the type scheme  $\forall \alpha. \alpha \rightarrow \alpha$  thus making the expressions above typable. However, such a derivation, if allowed, could be used to construct a larger derivation for the following erroneous expression:

$$\text{let } h \text{ be } (\text{fn } g \Rightarrow (\text{let } f \text{ be } g \text{ in } (f \text{ true}; f \underline{5}))) \text{ in } h (\text{fn } x \Rightarrow x + 1)$$

An expression is *open* if it contains free term variables and *closed* otherwise. A *program* is a well-typed and closed expression. It is easy to show that for any valid judgment, the type environment must contain an entry for every free term variable of the expression and that any additional entries are unnecessary. Programs may therefore be typed in an empty environment and we use the abbreviated judgment form  $\triangleright e : \tau$  to assert that the program  $e$  has type  $\tau$ .

Technically, the given arrangement of type schemes in relation to bound variables results in so-called *prenex-predicative* polymorphism. Prenex refers to the fact that our type schemes are *shallow*, that is, nested quantifiers are not allowed. Predicativity refers to the fact that our domain of quantification is limited to monotypes, and therefore does not include types schemes themselves. As an example of the expressive limitations of this system, consider that the expression  $(\text{fn } x \Rightarrow (x \ x))$  which involves the self-application of the function bound variable  $x$  is not typable. In a more general system,

one can imagine assigning this expression the following non-shallow type:

$$(\forall\alpha.\alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\alpha \rightarrow \alpha)$$

### Type Soundness

Having defined a type system, or *static semantics*, and an evaluator, or *dynamic semantics*, the question now arises as to whether the system is sound in the sense of avoiding any possibility of type errors. Although we have been informal about what exactly constitutes a type error, we claim that all reasonable means of proceeding with evaluation have been accounted for in our formal semantics and we therefore identify errors with stuck states. In a more complete setting, the question of what potential error situations can and should be avoided is more subtle.<sup>3</sup>

We say that the type system is sound if all programs either diverge or produce a value of the expected type. In other words, the partiality of our semantic function should be the result only of *infinite loops* in our programs. In the vernacular of type-theorists, we may formalize Milner’s slogan — “programs don’t go wrong” — as follows:

#### Proposition 2.1 (Type Soundness)

If  $\triangleright e : \tau$  then either  $e$  diverges or  $\text{eval}(e) = v$  and  $\triangleright v : \tau$ .

The difficulties involved in proving type soundness are to some extent dependent on the style of dynamic semantics. The syntactic style of semantics used throughout the present work leads to a particularly simple and scalable proof technique based on the seminal work of Wright and Felleisen[59]. It will be instructive to sketch the main steps

---

<sup>3</sup>Cardelli’s work provides a detailed taxonomy of program errors [9].

of such a proof in the simple setting of our expression language.<sup>4</sup>

The syntactic proof technique advocated by Wright and Felleisen consists of two main steps. The first is to show that any transition from a well-formed state produces a well-formed state with the same type.

Proposition 2.2 (Type Preservation)

If  $\triangleright e : \tau$  and  $e \mapsto e'$  then  $\triangleright e' : \tau$ .

In the context of a non-deterministic reduction relation, this property is normally phrased relative to open expressions and referred to as *subject reduction*. We prefer the term type preservation to emphasize the deterministic nature of our transition relation. The proof of Proposition 2.2 is by case analysis on the transition rules and requires a number of auxiliary properties which we will not detail here. The second step is to show that evaluation progress can be made from any well-formed state which is not already a value.

Proposition 2.3 (Progress)

If  $\triangleright e : \tau$  then either  $e$  is a value, or there exists  $e'$  such that  $e \mapsto e'$ .

The proof is by induction in the structure of  $e$ . Type soundness is now an easy consequence of Proposition 2.2 and Proposition 2.3.

### Imperative Features

The purely functional system presented so far is a simple and elegant means of enforcing type safety without the need for cumbersome type annotations. Subtle problems

---

<sup>4</sup>Wright and Felleisen distinguish between strong type soundness, as formalized by Proposition 2.1, and weak type soundness in which the type of the value produced is unconstrained.

---


$$\begin{aligned}
(\mu, E[\text{new } v]) &\longmapsto (\mu\langle l \mapsto v \rangle, E[l]) \text{ if } l \notin \text{dom}(\mu) \\
(\mu, E[\text{get } l]) &\longmapsto (\mu, E[v]) \text{ if } \mu(l) = v \\
(\mu, E[\text{set } l v]) &\longmapsto (\mu\langle l \mapsto v \rangle, E[v]) \text{ if } l \in \text{dom}(\mu)
\end{aligned}$$


---

FIGURE 3. Transition Rules for References

arise, however, when extending this framework with the common imperative features of modern functional languages. We will explore this issue by defining a naïve extension for mutable references. Let  $l$  range over a set of location names which we will regard as (computational) values. We extend the syntax of our expression language as follows:

$$\begin{aligned}
v &::= \dots \mid l \\
e &::= \dots \mid \text{new } e \mid \text{get } e \mid \text{set } e_1 e_2
\end{aligned}$$

These operations represent respectively the creation, dereferencing, and assignment of reference cells. We extend the definition of evaluation contexts as follows:

$$E ::= \dots \mid \text{new } E \mid \text{get } E \mid \text{set } E e \mid \text{set } v E$$

A *store*, ranged over by  $\mu$ , is a finite map from location names to (closed) values. We write  $\text{dom}(\mu)$  for the domain of  $\mu$  and use the notation  $\mu\langle l \mapsto v \rangle$  for functional updates. Our machine states will have the form  $(\mu, e)$  consisting of a store and an expression. The transition rules for our reference operations are defined in FIGURE 3. Each of the other transition rules from FIGURE 1 may be adapted *mutatis mutandis*.

We will delay our discussion of the typing rules for machine states and instead fo-

---


$$\frac{\Gamma \triangleright e : \tau}{\Gamma \triangleright \text{new } e : \text{ref } \tau}$$

$$\frac{\Gamma \triangleright e : \text{ref } \tau}{\Gamma \triangleright \text{get } e : \tau}$$

$$\frac{\Gamma \triangleright e_1 : \text{ref } \tau \quad \Gamma \triangleright e_2 : \tau}{\Gamma \triangleright \text{set } e_1 e_2 : \tau}$$


---

FIGURE 4. Typing Rules for References

cus on surface expressions. We extend our syntax of monotypes with a type constructor for references as follows:

$$\tau ::= \dots \mid \text{ref } \tau$$

The typing rules for our operations, given in FIGURE 4, are essentially those employed by Standard ML. However, when used in conjunction with the type system defined so far, they are unsound. The following program shows the well-known problem with the naïve rules:

```
let r be new (fn x ⇒ x) in set r (fn x ⇒ x + 1); (get r) true
```

Operationally, the program above first allocates a new reference cell containing the polymorphic identity function. The expression bound to  $r$  above may be assigned the type  $\text{ref } (\alpha \rightarrow \alpha)$  which may in turn be generalized resulting in the type scheme  $\forall \alpha. (\text{ref } (\alpha \rightarrow \alpha))$  being associated with the let-bound variable  $r$ . Within the body of the let-expression, the newly allocated cell is then assigned to the successor function.

This use of the bound variable  $r$  is justified by the following subsumption:

$$\forall\alpha.(\text{ref } (\alpha \rightarrow \alpha)) \succ \text{ref } (\text{int} \rightarrow \text{int})$$

Finally, the location is dereferenced and its contents applied to a boolean constant. This second use of the variable  $r$  is similarly justified:

$$\forall\alpha.(\text{ref } (\alpha \rightarrow \alpha)) \succ \text{ref } (\text{bool} \rightarrow \text{bool})$$

Thus, although the program above is well-formed according to our typing rules, its evaluation results in the stuck state  $\text{true} + \underline{1}$ . Intuitively, because the reference cell originally contained a polymorphic function, a more appropriate type for the variable  $r$  may be  $\text{ref } (\forall\alpha.\alpha \rightarrow \alpha)$ . Unfortunately, such non-shallow types cannot be accommodated within the confines of Milner's system.

The problem of soundly incorporating imperative operations into Milner's system has received extensive attention[24, 32, 33, 43, 46, 47, 57]. Indeed, similar problems arise with the naïve inclusion of first-class continuations, or equivalently, with the combination of references and exceptions[19]. Earlier versions of Standard ML adopted a solution based on the work of Tofte[47]. The key intuition underlying Tofte's proposal is that one should avoid generalizing type variables which would occur free in the types of locations in the store. This phenomenon should be clearly visible in our example above. Thus, a distinction is enforced between *imperative* and *applicative* type variables with only the later variety being subject to generalization.

Based on the observations of Wright, Standard ML currently solves this problem by enforcing a simple value restriction on let-bound expressions[58]. By allowing gen-

eralization only for the types of syntactic values, examples like our program above are ruled out. Non-value expressions may still be let-bound, however, the corresponding variable may not be used heterogeneously. To clarify, consider:

$$\text{let } f \text{ be } (\text{fn } x \Rightarrow x) (\text{fn } x \Rightarrow x) \text{ in } f \underline{5}; f \text{ false}$$

Although the let-bound expression above is harmless, this expression is rejected since it requires a polymorphic assumption for the let-bound variable  $f$ . Wright observed that in the case of purely functional expressions, this limitation could be side-stepped via the mechanism of eta-expansion. For example, the let-bound expression above could be replaced with the following operationally equivalent value:

$$\text{fn } y \Rightarrow (((\text{fn } x \Rightarrow x) (\text{fn } x \Rightarrow x)) y)$$

Using the syntactic proof technique introduced earlier, one can show that the typing rules for references given in FIGURE 4 are sound if type generalization is restricted to syntactic values. Although we will not prove type soundness in detail, it will be instructive to see how the static semantics of our modified abstract machine could be arranged for such a proof.

The typing rules given in FIGURE 5 employ three new judgment forms. The judgment  $\triangleright (\mu, e) : \tau$  asserts that the machine state  $(\mu, e)$  is well-formed and may be assigned type  $\tau$ . The judgment  $\triangleright \mu : \Sigma$  asserts that the store  $\mu$  is well-formed with store type  $\Sigma$  where  $\Sigma$  ranges over finite maps from location names to monotypes. As usual, we write  $ftv(\Sigma)$  for the set of free type variables occurring in all of the types in the range of  $\Sigma$ . The judgment  $\Sigma; \Gamma \triangleright e : \tau$  asserts that the expressions  $e$  has type  $\tau$  relative

---


$$\begin{array}{c}
\frac{\triangleright \mu : \Sigma \quad \Sigma; \langle \rangle \triangleright e : \tau}{\triangleright (\mu, e) : \tau} \\
\\
\frac{\Sigma; \langle \rangle \triangleright \mu(x) : \Sigma(x) \quad (\forall x \in \text{dom}(\Gamma)) \quad \text{dom}(\mu) = \text{dom}(\Sigma)}{\triangleright \mu : \Sigma} \\
\\
\frac{\Sigma(l) = \tau}{\Sigma; \Gamma \triangleright l : \tau} \\
\\
\frac{\Sigma; \Gamma \triangleright v : \tau_1 \quad \Sigma; \Gamma \langle x : \forall \vec{\alpha}. \tau_1 \rangle \triangleright e : \tau_2 \quad \{\vec{\alpha}\} \cap (\text{ftv}(\Sigma) \cup \text{ftv}(\Gamma)) = \emptyset}{\Sigma; \Gamma \triangleright \text{let } x \text{ be } v \text{ in } e : \tau_2} \\
\\
\frac{\Sigma; \Gamma \triangleright e_1 : \tau_1 \quad \Sigma; \Gamma \langle x : \tau_1 \rangle \triangleright e : \tau_2}{\Sigma; \Gamma \triangleright \text{let } x \text{ be } e_1 \text{ in } e_2 : \tau_2}
\end{array}$$


---

FIGURE 5. Selected Typing Rules for Machine States

to both a store type  $\Sigma$  and a typing environment  $\Gamma$ . The nature of mutable references leads to the possibility of mutual recursion among the values stored in reference cells. Consequently, these values are typed relative to the type of the entire store in which they reside. With the exception of the rule for let-expressions given in FIGURE 2, each of the rules from FIGURE 2 and FIGURE 4 may be adapted directly to this new judgment form by including an arbitrary store type. The remaining rules account for dynamically created locations, and for our new convention of restricting type-generalization to apply only to the types of let-bound values. The usual proviso for type generalization is also modified to account for free type variables occurring in the store type. One can now establish typability preservation and progress properties for the modified system.



### Typing References with Effects

Although the value restriction provides a simple and elegant means for imperative operations to safely coexist with Milner-style polymorphism, a number of more flexible proposals have been based on the concept of an *effect system*[13, 42, 43, 57]. In this section we discuss a variant of one of the more sophisticated proposals based on the work of Talpin and Jouvelot.

We have already seen two approaches to incorporating references in Milner's system. The following example reveals a shortcoming of each:

$$\text{let } f \text{ be get (new (fn } x \Rightarrow x)) \text{ in } f \underline{5}; f \text{ false}$$

Evaluation of the let-bound expression above results in the harmless creation and dereferencing of a location containing the identity function. Both the imperative type discipline of Tofte, and the value-restricted system advocated by Wright would reject the program above. Thus, the intuition that one should avoid generalizing type variables which occur free in the types of locations is in some cases overly restrictive. Moreover, because the bound expression is not purely functional, the simple device of eta-expansion would not yield an operationally equivalent program. Intuitively, what makes programs such as this harmless is the fact that the reference manipulated by the let-bound expression is entirely local to its evaluation. The type and effect discipline of Talpin and Jouvelot can be understood as exploiting this intuition by detecting the locality of reference cells.

The system we will now explore assigns both type and effect descriptions to the phrases of our expression language extended with store operations. Effects are defined using the concept of *regions* which should be regarded as unbounded partitions within

a store. Let  $\rho$  range over an infinite set of region variables which will serve as static names for regions. Let  $\epsilon$  range over an infinite set of effect variables. Atomic effects, written  $\text{eff}(\rho, \tau)$ , represent the manipulation of a reference residing in region  $\rho$  which holds a value of type  $\tau$ .<sup>5</sup> Effects, ranged over by  $\varphi$ , are finite sets of atomic effects and effect variables.

In order to track effects, the types for our language are redefined as follows:

$$\tau ::= \text{int} \mid \text{bool} \mid \alpha \mid \text{ref } \rho \tau \mid \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle$$

No confusion should arise from our repeated reuse of metavariables such as  $\tau$  throughout the present work. Our type constructor for references now carries an extra argument representing the region in which the reference resides. Arrow types are decorated with *latent effects* which represent a conservative approximation of the effects incurred when applying a function. Following common practice, we will not distinguish between effects, defined as sets, and their syntactic representations as they appear in types. Type schemes may now be quantified with respect to region, effect, and type variables:

$$\pi ::= \tau \mid \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau$$

We retain earlier notational conventions for typing environments and for the identification of type schemes. We use the term *constructor variable* to refer uniformly to all region, effect, and type variables. We write  $\text{fcv}(\pi)$ ,  $\text{fcv}(\varphi)$ , and  $\text{fcv}(\Gamma)$  for the set of free constructor variables occurring respectively in a type scheme, effect, and typing

---

<sup>5</sup>Our presentation differs from that of Talpin and Jouvelot in that we use only a single variety of atomic effect, rather than one for each reference operation.

environment. Similarly, we write  $frv(\cdot)$  and  $fev(\cdot)$  for free region and effect variables. Substitutions on types now come in three varieties: types may be substituted for type variables, effects for effect variables, and region variables for region variables.<sup>6</sup> Substitution is defined as  $\pi \succ \tau$  if and only if  $\pi \equiv \tau$  or  $\pi \equiv \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau'$  and  $\tau$  can be obtained from  $\tau'$  via substitutions for zero or more of the constructor variables occurring in  $\vec{\rho}$ ,  $\vec{\epsilon}$ , and  $\vec{\alpha}$ .

The type and effect rules given in FIGURE 6 and FIGURE 7 employ judgments of the form  $\Gamma \triangleright e : \tau, \varphi$  asserting that the expression  $e$  has type  $\tau$  and effect  $\varphi$  relative to typing environment  $\Gamma$ . The rules in FIGURE 6 might informally be called *propagation* rules; in addition to doing the job of their counterparts from FIGURE 2, they serve to propagate effect information through typing derivations. In contrast, the rules given in FIGURE 7 either generate or remove effects at some point in a derivation.

Syntactic values require no evaluation and thus cannot generate effects. Each of the rules for values in FIGURE 6 therefore assigns the empty effect along with the usual type to the given value. The effects incurred by an application consist of those generated by evaluating both the function and argument, as well as the latent effect of applying the function. Similar reasoning applies to conditionals and primitive operations where in each case, the effects of evaluating each subexpression are accumulated. The latent effect attributed to a function is simply the effect of evaluating its body.

The typing rule for let-expressions now allows for quantification of any constructor variable according (almost) to the usual criterion. In addition to avoiding the generalization of variables occurring free in the typing environment, the proviso also includes variables occurring in the effect of the let-bound expression. Intuitively, this is exactly

---

<sup>6</sup>For now, we omit region constants which will become relevant only when defining dynamic semantics for region systems in later chapters.

---


$$\frac{\Gamma(x) \succ \tau}{\Gamma \triangleright x : \tau, \emptyset}$$

$$\frac{}{\Gamma \triangleright \underline{n} : \text{int}, \emptyset}$$

$$\frac{}{\Gamma \triangleright \text{true} : \text{bool}, \emptyset}$$

$$\frac{}{\Gamma \triangleright \text{false} : \text{bool}, \emptyset}$$

$$\frac{\Gamma\{f : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle\}[x : \tau_1] \triangleright e : \tau_2, \varphi}{\Gamma \triangleright \text{rec } f(x) \Rightarrow e : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle, \emptyset}$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \rightarrow \langle \varphi_1, \tau_2 \rangle, \varphi_2 \quad \Gamma \triangleright e_2 : \tau_1, \varphi_3}{\Gamma \triangleright e_1 e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

$$\frac{\{ \vec{\rho}, \vec{\epsilon}, \vec{\alpha} \} \cap (\text{fcv}(\Gamma) \cup \text{fcv}(\varphi_1)) = \emptyset \quad \Gamma \triangleright e_1 : \tau_1, \varphi_1 \quad \Gamma(x : \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau_1) \triangleright e_2 : \tau_2, \varphi_2}{\Gamma \triangleright \text{let } x \text{ be } e_1 \text{ in } e_2 : \tau_2, \varphi_1 \cup \varphi_2}$$

$$\frac{\Gamma \triangleright e_1 : \text{bool}, \varphi_1 \quad \Gamma \triangleright e_2 : \tau, \varphi_2 \quad \Gamma \triangleright e_3 : \tau, \varphi_3}{\Gamma \triangleright \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau, \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

$$\frac{\Gamma \triangleright e_1 : \text{int}, \varphi_1 \quad \Gamma \triangleright e_2 : \text{int}, \varphi_2}{\Gamma \triangleright e_1 \text{ op } e_2 : \tau_b, \varphi_1 \cup \varphi_2}$$

where  $\tau_b \equiv \text{int}$  if  $\text{op} \in \{+, -\}$  and  $\tau_b \equiv \text{bool}$  otherwise

$$\frac{\Gamma \triangleright e : \text{bool}, \varphi}{\Gamma \triangleright !e : \text{bool}, \varphi}$$


---

FIGURE 6. Talpin-Jouvelot Calculus (Part 1)

---


$$\frac{\Gamma \triangleright e : \tau, \varphi}{\Gamma \triangleright \text{new } e : \text{ref } \rho \tau, \varphi \cup \{\text{eff}(\rho, \tau)\}}$$

$$\frac{\Gamma \triangleright e : \text{ref } \rho \tau, \varphi}{\Gamma \triangleright \text{get } e : \tau, \varphi \cup \{\text{eff}(\rho, \tau)\}}$$

$$\frac{\Gamma \triangleright e_1 : \text{ref } \rho \tau, \varphi_1 \quad \Gamma \triangleright e_2 : \tau, \varphi_2}{\Gamma \triangleright \text{set } e_1 e_2 : \tau, \varphi_1 \cup \varphi_2 \cup \{\text{eff}(\rho, \tau)\}}$$

$$\frac{\Gamma \triangleright e : \tau, \varphi_1 \quad \varphi_1 \subseteq \varphi_2}{\Gamma \triangleright e : \tau, \varphi_2}$$

$$\frac{\Gamma \triangleright e : \tau, \varphi \quad \rho \notin \text{fru}(\Gamma) \quad \rho \notin \text{fru}(\tau)}{\Gamma \triangleright e : \tau, \varphi \setminus \{\text{eff}(\rho, \tau)\}}$$

$$\frac{\Gamma \triangleright e : \tau, \varphi \quad \epsilon \notin \text{fev}(\Gamma) \quad \epsilon \notin \text{fev}(\tau)}{\Gamma \triangleright e : \tau, \varphi \setminus \{\epsilon\}}$$


---

FIGURE 7. Talpin-Jouvelot Calculus (Part 2)

the observation made by Tofte: type variables occurring in the types of locations, or equivalently in the effect description of a let-bound expression, should not be generalized. However, a key aspect of the expressiveness of the type and effect system is that in some cases, localized effects can be *masked*. We will discuss this feature shortly.

Turning to the typing rules in FIGURE 7, one can see that each reference operation generates a new effect and that newly created references may be placed in an arbitrary region. Note, however, that the references created by programs cannot always be placed in different regions. Consider, for example, the following conditional expression where  $e$  is arbitrary:

$$\text{if } e \text{ then (new } \underline{5} \text{) else (new } \underline{7} \text{)}$$

Because the branches of a conditional are required to have the same type, the regions assigned to the locations above are forced to be *unified*. Similarly, the next rule, called effect subsumption, allows for effect descriptions to be made less precise. Consider

$$\text{if } e \text{ then (fn } x \Rightarrow \underline{5} \text{) else (let } r \text{ be new } \underline{5} \text{ in (fn } x \Rightarrow \text{get } r \text{))}$$

Effect subsumption is necessary to attribute an appropriate latent effect to the pure constant function above.

Finally, the last two rules of FIGURE 7 allow for the masking of effects which are purely local to a given expression. Intuitively, if a region variable does not occur free in any assumption in the typing environment, then the region need not exist prior to the evaluation of an expression. Furthermore, if a region variable does not occur free in the type of an expression, for example, as a latent effect, then it will not be accessed again after evaluation. Similar reasoning applies to effect variables. This operational reading

of the effect masking criteria will have greater significance in the coming chapters.

To appreciate the utility of polymorphism with respect to regions and effects, consider the following expression:

$$\text{fn } f \Rightarrow (\text{fn } x \Rightarrow (\text{new } (f \ x)))$$

A *principal type scheme* for an expression is one which subsumes every type which could legally be assigned to it. The existence of principal types is an important aspect of the computational tractability of type-inference. Relative to Milner's system, the expression above has the following principal type scheme:

$$\forall \alpha_1, \alpha_2. (\alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha_1 \rightarrow \text{ref } \alpha_2)$$

Relative to the type and effect system, note that applying the expression above to a single argument should incur no effect, but the argument itself could be a function with any latent effect whatsoever. Applying it to a second argument results in the allocation within an arbitrary region, as well as the latent effects of the first argument. However, we have already seen that effect information can always be made less precise through the effect subsumption rule. Thus we arrive at the following type scheme:

$$\forall \rho, \epsilon_1, \epsilon_2, \epsilon_3, \alpha_1, \alpha_2. (\alpha_1 \rightarrow \langle \{\epsilon_1\}, \alpha_2 \rangle) \rightarrow \langle \{\epsilon_2\}, (\alpha_1 \rightarrow \langle \{\epsilon_1, \epsilon_3, \text{eff}(\rho, \alpha_2)\}, \text{ref } \rho \ \alpha_2 \rangle) \rangle$$

The system presented here was shown to be sound relative to a *big-step* operational semantics[43]. Talpin and Jouvelot also provided an algorithm for inferring the principal type and minimal effect of an expression. Wadler later pointed out certain infelicities in

the proof of correctness for the inference algorithm while adapting it to a closely related setting[55].



## CHAPTER III

### REGIONS

An important prerequisite to the success of higher-order programming languages such as Java and Standard ML has been freeing programmers from the burden of memory management. Extensive research has been devoted to improving the technologies underlying automatic garbage collectors. An alternative approach, static memory management, has been developed in the work of Tofte, Talpin, and Birkedal among others [1, 7, 11, 48, 50, 51, 52, 54]. The key formalism underlying this approach is the Tofte-Talpin region calculus, a type and effect system which describes how annotations may be statically added to programs which safely determine their interaction with memory. This approach has been implemented in the ML-Kit with Regions[49].

#### Static Memory Management

Static memory management is based on an execution model in which the heap is organized as a stack of regions. Allocations can potentially be made into any region currently on the stack and are guided by annotations inserted into the program by region analysis. New (empty) regions are pushed onto the stack and later deallocated in a last-in-first-out order, again following program annotations. It is important that although each region can grow indefinitely during its lifetime, deallocation of an entire region takes only constant time.

The essential features of region analysis can be clarified by a simple example. Returning to the expression language introduced in CHAPTER II, consider a program

containing the fragment  $(\dots((\text{fn } x \Rightarrow x) \underline{5})\dots)$ . The region inference process might produce the following annotated program:

$$\text{letreg } \rho \text{ in } (\dots(\text{letreg } \rho' \text{ in } (\text{fn } x \Rightarrow x) \text{ at } \rho' (\underline{5} \text{ at } \rho))\dots)$$

Every value in the source program is annotated to indicate into which run-time region it will be allocated. Regions are introduced with the lexically scoped construct  $\text{letreg } \rho \text{ in } e$  which binds the region variable  $\rho$  within  $e$ . These expressions are evaluated by first pushing an empty region onto the stack, then evaluating  $e$ , and finally popping the top region from the stack. In our example, the closure for the function  $(\text{fn } x \Rightarrow x)$  is allocated in the inner region,  $\rho'$ , and its argument is allocated in the outer region,  $\rho$ . When the inner region is popped, it is safe to reclaim the closure allocated in  $\rho'$ , but not so for its argument which may yet be needed in the context of the outer region.

### The Region Calculus

The region calculus formalizes how programs may be safely annotated. Most of the technical features of the region calculus can easily be seen as stemming from the type and effect framework of Tofte and Talpin discussed in CHAPTER II. The central ideas are to incorporate the *implicit* effects of the intended implementation framework into the typing rules, and to use the effect-masking criteria to guide the introduction and collection of memory regions.

We will now define in detail a slightly simplified variant of the original Tofte-Talpin calculus.<sup>1</sup> We have already introduced most of the required notational conven-

---

<sup>1</sup>The main simplification is related to the omission of arrow-effect labels which are relevant only to the algorithmic inference process. Cray *et al.* define an explicitly-typed variant similar to ours[11].

tions in CHAPTER II including the meaning of meta-variables such as  $x$ ,  $f$ ,  $\rho$ ,  $\alpha$ , and  $\epsilon$ .

The syntax of expressions is defined as follows:

$$e ::= x \mid \underline{n} \text{ at } \rho \mid \text{fn } x \Rightarrow e \text{ at } \rho \mid e_1 e_2 \mid \text{letrec } f \langle \bar{\rho} \rangle (x) \text{ at } \rho \text{ be } e_1 \text{ in } e_2 \mid \\ f \langle \bar{\rho} \rangle \text{ at } \rho \mid \text{letreg } \rho \text{ in } e$$

Variables and applications appear in their usual form. In our example above, we have already seen the use of region annotations on numerals and functions and the use of the let-region construct itself. Although numerals provide a representative base type, for brevity we will henceforth omit arithmetic operations from our formal presentation.

The remaining constructs relate to the introduction and application of region polymorphic functions. In contrast to the systems discussed in CHAPTER II, the presence of region annotations requires that polymorphism with respect to regions be made *explicit*, that is, region-polymorphic functions are passed actual arguments representing the regions they will act on.

The expression  $(\text{letrec } f \langle \bar{\rho} \rangle (x) \text{ at } \rho \text{ be } e_1 \text{ in } e_2)$  allocates a function named  $f$  with body  $e_1$  in the region named  $\rho$  and then delivers the result of evaluating  $e_2$ . The function  $f$  has formal parameters  $\bar{\rho}$  and  $x$  which are bound within the function body. Because the function may be recursive, the variable  $f$  is bound both in  $e_1$  and  $e_2$ .

In order to apply such a function, actual regions arguments must first be provided via the expression  $(f \langle \bar{\rho} \rangle \text{ at } \rho)$ . Operationally, this expression dereferences the region-polymorphic function represented by  $f$  and applies it to the actual arguments given by  $\bar{\rho}$ . This results in an ordinary function closure which is then allocated in region  $\rho$ . Recall that the notation  $\bar{\rho}$  stands for a sequence of distinct region variables so that the formal region parameters of function are distinct. In contrast, the notation  $\bar{\rho}$  refers to

any sequence of region variables so that the actual arguments may overlap. We write  $|\vec{\rho}|$  and  $|\vec{\rho}'|$  for the length of these sequences. We will often use notation such as  $[\vec{\rho}' := \vec{\rho}]$  for the simultaneous substitution of constructors for constructor variables. This notation is only defined when the given sequences have the same length.

Types will be defined mutually with a new syntactic category of *types with places*, ranged over by  $\mu$ :

$$\mu ::= (\tau, \rho)$$

The reader may consider types with places as corresponding closely to region annotated reference types of the form  $(\text{ref } \rho \tau)$ . Indeed, in the remainder of this work, reference types will supplant the former. The definition of types is modified so that functions are assumed to receive and produce allocated values:

$$\tau ::= \text{int} \mid \alpha \mid \mu_1 \rightarrow \langle \varphi, \mu_2 \rangle$$

For the remainder of this work we represent atomic effects with bare regions. Therefore we now assume effects, ranged over by  $\varphi$ , are finite sets of region and effect variables.

We use  $\sigma$  to range over type schemes which quantify region, effect, and type variables over types with places:

$$\sigma ::= \mu \mid \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \mu$$

The subsumption relation,  $\succ$ , is defined as usual. The systems discussed in CHAPTER II allowed only values with mono-types to be allocated. We have already seen that this restriction is relaxed in the Tofte-Talpin calculus by allocations of region-polymorphic function closures. Thus, we require an additional syntactic category of type schemes

---


$$\frac{\Gamma(x) = \mu}{\Gamma \triangleright x : \mu, \emptyset}$$

$$\Gamma \triangleright \underline{n} \text{ at } \rho : (\text{int}, \rho), \{\rho\}$$

$$\frac{\Gamma(x : \mu_1) \triangleright e : \mu_2, \varphi_1 \quad \varphi_2 \supseteq \varphi_1}{\Gamma \triangleright \text{fn } x \Rightarrow e \text{ at } \rho : (\mu_1 \rightarrow \langle \varphi_2, \mu_2 \rangle, \rho), \{\rho\}}$$

$$\frac{\Gamma \triangleright e_1 : (\mu_1 \rightarrow \langle \varphi_1, \mu_2 \rangle, \rho), \varphi_2 \quad \Gamma \triangleright e_2 : \mu_1, \varphi_3}{\Gamma \triangleright e_1 e_2 : \mu_2, \varphi_1 \cup \varphi_2 \cup \varphi_3 \cup \{\rho\}}$$

$$\frac{\begin{array}{l} \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\} \cap (\text{fcv}(\Gamma) \cup \{\rho\}) = \emptyset \\ \Gamma \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. (\tau, \rho) \rangle \triangleright \text{fn } x \Rightarrow e_1 \text{ at } \rho : \tau, \{\rho\} \\ \Gamma \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. (\tau, \rho) \rangle \triangleright e_2 : \mu, \varphi \end{array}}{\Gamma \triangleright \text{letrec } f \langle \bar{\rho} \rangle (x) \text{ at } \rho \text{ be } e_1 \text{ in } e_2 : \mu, \varphi \cup \{\rho\}}$$

$$\frac{\Gamma(f) = (\forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau, \rho) \quad \forall \bar{\epsilon}, \bar{\alpha}. \tau[\bar{\rho} := \bar{\rho}] \succ \tau'}{\Gamma \triangleright f \langle \bar{\rho} \rangle \text{ at } \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

$$\frac{\Gamma \triangleright e : \tau, \varphi \quad \rho \notin \text{frv}(\Gamma) \quad \rho \notin \text{frv}(\tau)}{\Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}}$$


---

FIGURE 8. Tofte-Talpin Calculus Variant

with places, ranged over by  $\pi$ :

$$\pi ::= (\pi, \rho)$$

The type and effect rules for the Tofte-Talpin system are defined in FIGURE 8 using judgments of the form  $\Gamma \triangleright e : \mu, \varphi$  asserting that the expression  $e$  has type with place  $\mu$  and effect  $\varphi$  relative to typing environment  $\Gamma$ . The reader may wish to read these rules in contrast to the Talpin-Jouvelot system defined in FIGURES 6 and 7 of CHAPTER II. Just as before, the type of a variable is determined by the typing context and variables incur no effect. In contrast, numerals and functional abstractions now

generate an effect equivalent to the region annotation they are assigned. The proviso  $\varphi_2 \supseteq \varphi_1$  allows a less-precise latent effect to be assigned and thus plays the rôle of the effect subsumption rule in the Talpin-Jouvelot system. Just as functions are always allocated in some region, the typing rule for applications incorporates the implicit effect of dereferencing the closure for  $e_1$ .

The rules for declaring and applying polymorphic functions are more complex. Generalization of type, region, and effect variables is controlled by the usual proviso preventing the quantification of constructors occurring free in the typing context. This proviso is extended to include the region into which the closure is being allocated which is assumed to be bound outside the expression. The instantiation of a polymorphic type occurs only when region arguments are supplied in the typing rule for  $(f \langle \bar{\rho} \rangle \text{ at } \rho)$ . This instantiation is dependent, in part, on the actual region arguments supplied.

In relation to the Milner-style let-construct seen previously, the letrec-construct in the Tofte-Talpin system has an interesting and important twist. Note that the body of the recursive function declared by a letrec is typed relative to a polymorphic assumption about the type of the function itself. Intuitively, *polymorphic recursion* is the ability for a recursive function to make heterogeneous use of itself within its own body. Polymorphic recursion in types leads, in general, to the undecidability of type-inference[22]. For practical reasons, the Tofte-Talpin calculus permits polymorphic recursion in regions and effects only.

Finally, we have already discussed the operational intuition behind the masking of region variables in the Talpin-Jouvelot system in CHAPTER II. The Tofte-Talpin system takes this intuition more seriously by using the same criteria to guide the typing of let-region expressions.

A typical example of the use of region polymorphism is when the value returned by a function is stored in some region provided as an argument. Consider the following program fragment:

$$\text{letrec } f(\rho)(x) \text{ at } \rho' \text{ be } (\underline{5} \text{ at } \rho) \text{ in } \dots$$

The function  $f$  requires a single region argument named by formal parameter  $\rho$ , and an ordinary argument named  $x$  whose type is unconstrained. The function itself is allocated in region  $\rho'$  and delivers an integer allocated in  $\rho$ . The variable  $f$  could be assigned the following type scheme with place in the body of the expression above:

$$(\forall \rho, \epsilon, \alpha. \alpha \rightarrow \langle \epsilon \cup \{\rho\}, (\text{int}, \rho) \rangle, \rho')$$

Although the latent effect of applying the function  $f$  could be just the set  $\{\rho\}$ , we have already seen that one gains greater flexibility by allowing  $f$  to take on larger effect sets as well. This can be achieved by instantiating the quantified effect variable  $\epsilon$  above.

### The Language $\mathcal{RL}$

Our goal is to capture the essential aspects of region systems such as the Tofte-Talpin calculus in way that facilitates both simple and scalable proofs of soundness. We have already seen the close analogy between types with places in the Tofte-Talpin system, and the reference types of the Talpin-Jouvelot system defined in CHAPTER II. For clarity and simplicity, we choose to work with only explicit effects and their associated reference types.

For ease of presentation, we will dispense with any separation of surface terms from computational terms, choosing instead to view our language as already embedded

in an abstract machine. As a first step, we assume  $\xi$  ranges over a new set of region names which will serve as the runtime counterparts to region variables. Region identifiers, ranged over by  $\gamma$ , are defined as either region variables or names:

$$\gamma ::= \rho \mid \xi$$

We use the notation  $frn(\cdot)$  for the set of region names occurring in a type or expression, even though region names cannot be bound in the sense of constructor variables.

Our expressions will be mutually defined with syntactic values:

$$v ::= x \mid \underline{n} \mid \text{fn } x \Rightarrow e \mid \text{rec } f\langle \vec{\rho} \rangle(x) \Rightarrow e \mid (\xi, l)$$

In addition to variables, numerals, and functional abstractions, our language includes a construct for introducing recursive region-polymorphic functions. The two-dimensional nature of memory in our execution model leads to the use of pairs of the form  $(\xi, l)$  to represent a reference to location  $l$  in region  $\xi$ . Expressions are defined as follows:

$$e ::= v \mid e \langle \vec{\gamma} \rangle \mid e_1 e_2 \mid \text{new } \gamma e \mid \text{get } e \mid \text{letreg } \rho \text{ in } e$$

where  $\vec{\gamma}$  is a sequence of region names which serve as arguments for region-polymorphic functions. Our language includes operations for allocating and dereferencing locations. The operation  $(\text{new } \gamma e)$  generalizes the region annotations defined in the Tofte-Talpin language by allowing the value of any expression to be allocated. Finally, regions are introduced by  $(\text{letreg } \rho \text{ in } e)$  which binds  $\rho$  in  $e$  and will be subject to the same operational reading previously introduced.



In contrast to approaches based on evaluation contexts which were discussed in CHAPTER II, our abstract machine will maintain an explicit representation of the *continuation* for the current program expression. Continuations, ranged over by  $K$ , are defined as follows:

$$K ::= [\cdot] \mid K \diamond ([\cdot] \langle \bar{\gamma} \rangle) \mid K \diamond ([\cdot] e) \mid K \diamond (v [\cdot]) \mid K \diamond (\text{pop}; [\cdot])$$

Empty continuations, indicated with a hole,  $[\cdot]$ , are extended to the right by continuation frames. Each of these frames contains another hole where, intuitively, the value delivered by the next frame to the right should fit. In analogy to the sequential operation discussed in CHAPTER II, frames of the form  $(\text{pop}; [\cdot])$  represent the operation of receiving a value in the given hole, then popping the region stack, and then delivering the value unchanged.

Our theoretical study of regions will be based on more general framework which is not limited to *shallow* types in the sense introduced in CHAPTER II. Thus, we define types as a single syntactic category which includes quantified types:

$$\tau ::= \text{int} \mid \alpha \mid \text{ref } \gamma \tau \mid \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle \mid \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$$

The only modifications we require from earlier definitions relate to the fact that region identifiers, not just region variables, can occur in types and effects. For the remainder of this work, we assume effects, ranged over by  $\varphi$ , are finite sets of region identifiers and region variables. In analogy to the  $\bar{\rho}$  notation already mentioned, we will use  $\bar{\gamma}$ ,  $\bar{\varphi}$ , and  $\bar{\tau}$  for sequences of region identifiers, effects, and types which in each case, need not be distinct. Constructor substitutions now map region variables to region identifiers,

effect variables to effects, and type variables to types. We will often write simultaneous constructor substitutions as  $[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ .

Memory regions, ranged over by  $R$ , are finite maps from locations,  $l$ , to values. In order to accurately capture the region-based execution model, we will view memories as finite *sequences* of binding between distinct region names and regions. The resulting region stacks, written  $\bar{s}$ , have the form

$$\xi_1 \mapsto R_1, \dots, \xi_l \mapsto R_k$$

With some abuse, we will treat these sequences dually as finite maps from region names to regions by using typical notations such as  $\bar{s}(\xi \mapsto R)$  for functional updates. We also write  $\bar{s}(\xi)(l)$  to obtain the value  $v$  such that  $\bar{s}(\xi) = R$  and  $R(l) = v$ . The operation  $\bar{s} @ \bar{s}'$  appends two sequences. In order for our treatment of sequences as maps to make sense, this operation is only defined when  $dom(\bar{s}) \cap dom(\bar{s}') = \emptyset$ . We write  $\bullet$  for the empty sequence and assume  $\bar{s} @ \bullet = \bullet @ \bar{s} = \bar{s}$ . Our region stacks will grow to the right. Thus, pushing a new region  $R$  named  $\xi$  on to the stack  $\bar{s}$  will be expressed as  $\bar{s} @ \xi \mapsto R$ .

Region types, ranged over by  $\Sigma$ , are finite maps from locations to types. Memory types, written  $\bar{\tau}$ , are finite sequences of binding between region names and region types. Thus, memory types have the form

$$\xi_1 \mapsto \Sigma_1, \dots, \xi_l \mapsto \Sigma_k$$

and are subject to all of the notational conventions discussed above for memories.

Machine states, ranged over by  $M$  are triples consisting of a region stack, a con-

tinuation, and an expression:

$$M ::= (\bar{s}, K, e)$$

Informally, we may regard *initial* states of computation as having the form  $(\bar{s}, [\cdot], e)$  where  $\bar{s}$  contains the global regions to be used by the program  $e$ .

### Dynamic Semantics

The transition rules for  $\mathcal{RL}$  are defined in FIGURE 9. In the remainder of this work, we will continue to use  $\mapsto^*$  for the reflexive and transitive closure of a transition relation  $\mapsto$ . For clarity, the rules are grouped into three sets. The decomposition rules are *administrative* in the sense that they do not correspond to normal computational steps in evaluation. Instead, they serve to explicitly build and maintain a syntactic representation of the continuation for the current program expression. The computational rules express the normal operations associated with function applications and references. Finally, the region management rules express the pushing and popping of memory regions.

A simple example will help to clarify the use of continuations. Consider the following initial machine state:

$$(\bullet, [\cdot], \text{letreg } \rho \text{ in } ((\text{fn } x \Rightarrow (\text{get } x)) (\text{new } \rho \underline{5})))$$

The first evaluation step is given by (T10) which requires that we choose a run-time name for the region  $\rho$ . For technical reasons, this name must be chosen to be distinct from any region names occurring in the current program expression and region stack. The notation  $frn(\bar{s})$  refers to the set of all region names occurring in values stored

## [Decomposition]

$$(T1) \quad (\bar{s}, K, e \langle \bar{\gamma} \rangle) \mapsto (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), e)$$

$$(T2) \quad (\bar{s}, K, e_1 e_2) \mapsto (\bar{s}, K \diamond ([\cdot] e_2), e_1)$$

$$(T3) \quad (\bar{s}, K \diamond ([\cdot] e), v) \mapsto (\bar{s}, K \diamond (v [\cdot]), e)$$

$$(T4) \quad (\bar{s}, K, \text{new } \gamma e) \mapsto (\bar{s}, K \diamond (\text{new } \gamma [\cdot]), e)$$

$$(T5) \quad (\bar{s}, K, \text{get } e) \mapsto (\bar{s}, K \diamond (\text{get } [\cdot]), e)$$

## [Computation]

$$(T6) \quad (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), \text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e) \\ \mapsto \\ (\bar{s}, K, (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e)][\bar{\rho} := \bar{\gamma}])$$

$$(T7) \quad (\bar{s}, K \diamond ((\text{fn } x \Rightarrow e) [\cdot]), v) \mapsto (\bar{s}, K, e[x := v])$$

$$(T8) \quad (\bar{s}, K \diamond (\text{new } \xi [\cdot]), v) \mapsto (\bar{s} \langle \xi \mapsto R \langle l \mapsto v \rangle \rangle, K, (\xi, l)) \\ \text{if } \bar{s}(\xi) = R \text{ and } l \notin \text{dom}(R)$$

$$(T9) \quad (\bar{s}, K \diamond (\text{get } [\cdot]), (\xi, l)) \mapsto (\bar{s}, K, v) \\ \text{if } \bar{s}(\xi)(l) = v$$

## [Region Management]

$$(T10) \quad (\bar{s}, K, \text{letreg } \rho \text{ in } e) \mapsto (\bar{s} @ \xi \mapsto \langle \rangle, K \diamond (\text{pop}; [\cdot]), e[\rho := \xi]) \\ \text{if } \xi \notin \text{frn}(\bar{s}) \cup \text{frn}(e)$$

$$(T11) \quad (\bar{s} @ \xi \mapsto R, K \diamond (\text{pop}; [\cdot]), v) \mapsto (\bar{s}, K, v)$$

FIGURE 9. Transition Rules for  $\mathcal{RL}$

in  $\bar{3}$ . Choosing  $\xi$  for the name of our new region yields

$$(\xi \mapsto \langle \rangle, [\cdot] \diamond (\text{pop}; [\cdot]), (\text{fn } x \Rightarrow (\text{get } x)) (\text{new } \xi \underline{5}))$$

In an approach based on evaluation contexts, one would identify the program expression above as having the form  $E[\text{new } \xi \underline{5}]$  where  $E \equiv (\text{fn } x \Rightarrow (\text{get } x)) [\cdot]$ . Instead, our machine explicitly extends the current continuation with a frame equivalent to the context  $E$ . Thus, the application of rules (T2) and (T3) yields

$$(\xi \mapsto \langle \rangle, [\cdot] \diamond (\text{pop}; [\cdot]) \diamond ((\text{fn } x \Rightarrow (\text{get } x)) [\cdot]), \text{new } \xi \underline{5})$$

Because expressions of the form  $(\text{new } \xi e)$  could, in general, require further evaluation of  $e$ , our machine continues with the decomposition step given by (T4) yielding

$$(\xi \mapsto \langle \rangle, [\cdot] \diamond (\text{pop}; [\cdot]) \diamond ((\text{fn } x \Rightarrow (\text{get } x)) [\cdot]) \diamond (\text{new } \xi [\cdot]), \underline{5})$$

At this point, the program expression is a syntactic value so that our next step is dependent on the top-most continuation frame. Applying rule (T8) results in an allocation in the region named  $\xi$  producing

$$(\xi \mapsto \langle l \mapsto \underline{5} \rangle, [\cdot] \diamond (\text{pop}; [\cdot]) \diamond ((\text{fn } x \Rightarrow (\text{get } x)) [\cdot]), (\xi, l))$$

Once again, the resulting program expression is a syntactic value. Looking again at the top-most continuation frame, we see that this value is the argument for a function and

we apply rule (T7) to arrive at

$$(\xi \mapsto \langle l \mapsto \underline{\mathfrak{v}} \rangle, [\cdot] \diamond (\text{pop}; [\cdot]), \text{get}(\xi, l))$$

Evaluation now leads to the dereferencing of the location  $(\xi, l)$  via the rules (T5) and (T9) producing

$$(\xi \mapsto \langle l \mapsto \underline{\mathfrak{v}} \rangle, [\cdot] \diamond (\text{pop}; [\cdot]), \underline{\mathfrak{v}})$$

At this point, the let-region expression of our original program has been completely evaluated. Noting that the region stack has the form  $\bullet @ \xi \mapsto R$ , we apply (T11) yielding the final program state

$$(\bullet, [\cdot], \underline{\mathfrak{v}})$$

Although our initial and final states have empty region stacks, this need not be the case in general. Any number of global regions may be assumed in an initial configuration. Because these regions would not have corresponding continuation frames of the form  $(\text{pop}; [\cdot])$ , they would appear in the final state as well.

### Static Semantics

The static semantics for  $\mathcal{RL}$  is defined to facilitate a soundness proof based on typability preservation as discussed in CHAPTER II. Each of the syntactic categories we have defined — values, expression, continuations, regions, memories, and machine states — has a corresponding judgment form in the static semantics which will be discussed in turn.

The main departure from the systems studied so far is the use of a constructor environment to control type generalization. Recall that type, effect, and region variables,

---

(var)	$\frac{\Gamma(x) = \tau}{\bar{\zeta}; \Delta; \Gamma \triangleright x : \tau}$
(num)	$\bar{\zeta}; \Delta; \Gamma \triangleright \underline{n} : \text{int}$
(abs)	$\frac{\bar{\zeta}; \Delta; \Gamma \langle x : \tau_1 \rangle \triangleright e : \tau_2, \varphi_1 \quad \varphi_1 \subseteq \varphi_2 \quad fcv(\tau_1) \cup fcv(\varphi_2) \subseteq \Delta \quad x \notin \text{dom}(\Gamma)}{\bar{\zeta}; \Delta; \Gamma \triangleright \text{fn } x \Rightarrow e : \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle}$
(rec)	$\frac{\bar{\zeta}; \Delta \uplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Gamma \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau \rangle \triangleright \text{fn } x \Rightarrow e : \tau \quad fcv(\forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau) \subseteq \Delta \quad f \notin \text{dom}(\Gamma)}{\bar{\zeta}; \Delta; \Gamma \triangleright \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau}$
(loc-live)	$\frac{\bar{\zeta}(\xi)(l) = \tau \quad fcv(\tau) = \emptyset}{\bar{\zeta}; \Delta; \Gamma \triangleright (\xi, l) : \text{ref } \xi \tau}$
(loc-dead)	$\frac{\xi \notin \text{dom}(\bar{\zeta}) \quad fcv(\tau) = \emptyset}{\bar{\zeta}; \Delta; \Gamma \triangleright (\xi, l) : \text{ref } \xi \tau}$

---

FIGURE 10. Static Semantics for  $\mathcal{RL}$  (Part 1)

ranged over by  $\alpha$ ,  $\epsilon$ , and  $\rho$  respectively, are gathered into a single class of constructor variables. Constructor environments, ranged over by  $\Delta$ , are finite sets of constructor variables. We write  $\Delta \uplus \Delta'$  for the union of disjoint constructor environments.

The typing rules for values are given in FIGURE 10 using judgments of the form  $\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau$  asserting that the value  $v$  has type  $\tau$  relative to the given memory type, constructor environment, and typing environment. Because values incur no computational effects, no effect description is assigned.

The rules (abs) and (rec) enforce that any new type added to the typing context must be well-formed relative to the constructor environment. That is, the constructor environment,  $\Delta$ , should carry all of the free constructor variables of  $\Gamma$ . The quantifica-

---


$$\begin{array}{l}
\text{(val)} \quad \frac{\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau}{\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau, \emptyset} \\
\text{(papp)} \quad \frac{\bar{\zeta}; \Delta; \Gamma \triangleright e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau, \varphi}{\bar{\zeta}; \Delta; \Gamma \triangleright e \langle \bar{\gamma} \rangle : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}], \varphi} \\
\text{(app)} \quad \frac{\bar{\zeta}; \Delta; \Gamma \triangleright e_1 : \tau_1 \rightarrow \langle \varphi_1, \tau_2 \rangle, \varphi_2 \quad \bar{\zeta}; \Delta; \Gamma \triangleright e_2 : \tau_1, \varphi_3}{\bar{\zeta}; \Delta; \Gamma \triangleright e_1 e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi_3} \\
\text{(new)} \quad \frac{\bar{\zeta}; \Delta; \Gamma \triangleright e : \tau, \varphi \quad \text{frv}(\gamma) \subseteq \Delta}{\bar{\zeta}; \Delta; \Gamma \triangleright \text{new } \gamma e : \text{ref } \gamma \tau, \varphi \cup \{\gamma\}} \\
\text{(get)} \quad \frac{\bar{\zeta}; \Delta; \Gamma \triangleright e : \text{ref } \gamma \tau, \varphi}{\bar{\zeta}; \Delta; \Gamma \triangleright e : \tau, \varphi \cup \{\gamma\}} \\
\text{(letreg)} \quad \frac{\bar{\zeta}; \Delta \boxplus \{\rho\}; \Gamma \triangleright e : \tau, \varphi \quad \rho \notin \text{frv}(\tau)}{\bar{\zeta}; \Delta; \Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}}
\end{array}$$


---

FIGURE 11. Static Semantics for  $\mathcal{RL}$  (Part 2)

tion of constructor variables in the rule (*rec*) is allowed only for variables not already occurring in  $\Delta$ . The reader should understand these conditions as being equivalent to the usual proviso which restricts generalization in each of the systems previously studied. Note also that the rule (*rec*) admits full polymorphic recursion in types, regions, and effects. Locations are typed via the rules (*loc-live*) and (*loc-dead*). Because only closed values are allocated, the store type may only supply closed assumptions about their types which is seen in the proviso  $\text{fcv}(\tau) = \emptyset$  for each rule. We will discuss the necessity of the rule (*loc-dead*) shortly.

The typing rules for expressions are given in FIGURE 11 using judgments of the form  $\bar{\zeta}; \Delta; \Gamma \triangleright e : \tau, \varphi$  asserting that the expression  $e$  has type  $\tau$  and effect  $\varphi$  relative to the given memory type, constructor environment, and typing environment. Values



---


$$\begin{array}{l}
(k\text{-empty}) \quad \frac{\text{frn}(\tau) \subseteq \text{dom}(\bar{\zeta}) \quad \text{fcv}(\tau) = \emptyset}{\bar{\zeta} \triangleright [\cdot] \rightsquigarrow \tau} \\
(k\text{-papp}) \quad \frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]}{\bar{\zeta} \triangleright K \circ ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau} \\
(k\text{-app-l}) \quad \frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau_2 \quad \bar{\zeta}; \emptyset; \langle \rangle \triangleright e : \tau_1, \varphi_1 \quad \varphi_1 \cup \varphi_2 \subseteq \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \circ ([\cdot] e) \rightsquigarrow \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle} \\
(k\text{-app-r}) \quad \frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau_2 \quad \bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle \quad \varphi \subseteq \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \circ (v [\cdot]) \rightsquigarrow \tau_1} \\
(k\text{-new}) \quad \frac{\bar{\zeta} \triangleright K \rightsquigarrow \text{ref } \gamma \tau \quad \gamma \in \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \circ (\text{new } \gamma [\cdot]) \rightsquigarrow \tau} \\
(k\text{-get}) \quad \frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau \quad \gamma \in \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \circ (\text{get } [\cdot]) \rightsquigarrow \text{ref } \gamma \tau} \\
(k\text{-pop}) \quad \frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau}{\bar{\zeta} @ \xi \mapsto \Sigma \triangleright K \circ (\text{pop}; [\cdot]) \rightsquigarrow \tau}
\end{array}$$


---

FIGURE 12. Static Semantics for  $\mathcal{RL}$  (Part 3)

are accommodated via the rule (*val*) assigning them the empty effect. Polymorphic instantiation, given by the rule (*papp*), occurs only when expressions are applied to region arguments in a manner similar to the Tofte-Talpin system. The rules (*app*), (*new*), and (*get*) are similar to their counterparts in the Talpin-Jouvelot system of CHAPTER II and have been adapted here to the new judgment form. Similarly, the rule (*letreg*) differs from the construct defined in FIGURE 8 only in using the constructor environment to enforce the generalizability of the bound region variable.

The typing rules for continuations are given in FIGURE 12 using judgments of the

---


$$\begin{array}{c}
\text{(region)} \quad \frac{\text{dom}(R) = \text{dom}(\Sigma) \quad \bar{\tau}; \emptyset; \langle \rangle \triangleright R(l) : \Sigma(l) \quad (\forall l \in \text{dom}(R))}{\bar{\tau} \triangleright R : \Sigma} \\
\\
\text{(stack)} \quad \frac{\begin{array}{l} \bar{s} = \xi_1 \mapsto R_1, \dots, \xi_k \mapsto R_k \\ \bar{\tau} = \xi_1 \mapsto \Sigma_1, \dots, \xi_k \mapsto \Sigma_k \\ \bar{\tau} \triangleright \bar{s}(\xi) : \bar{\tau}(\xi) \quad (\forall \xi \in \text{dom}(\bar{s})) \end{array}}{\triangleright \bar{s} : \bar{\tau}} \\
\\
\text{(prog)} \quad \frac{\begin{array}{l} \triangleright \bar{s} : \bar{\tau} \quad \bar{\tau} \triangleright K \rightsquigarrow \tau \\ \bar{\tau}; \emptyset; \langle \rangle \triangleright e : \tau, \varphi \quad \varphi \subseteq \text{dom}(\bar{\tau}) \end{array}}{\triangleright (\bar{s}, K, e)}
\end{array}$$


---

FIGURE 13. Static Semantics for  $\mathcal{RL}$  (Part 4)

form  $\bar{\tau} \triangleright K \rightsquigarrow \tau$ . Intuitively,  $\tau$  is the type for the value expected by the right-most continuation frame. The memory type,  $\bar{\tau}$ , plays a special rôle in typing continuations because of the presence of frames of the form  $(\text{pop}; [\cdot])$ . The rule (*k-empty*) allows empty continuations to take on any closed type. In this rule, the memory type is assumed to be the type of the global regions which can never be freed by  $(\text{pop}; [\cdot])$  frames occurring to the right. The condition  $\text{frn}(\tau) \subseteq \text{dom}(\bar{\tau})$  enforces that the top-level type of the program can only contain these global regions. Therefore programs cannot produce, say, a function with latent effects in some dead region. The rule (*k-pop*) allows an additional region to be added to the memory type for the frames to the right. Each of the other rules enforce that the given frame will produce a value of the appropriate type, and that the effects associated with that computation involve only the regions currently in memory.

The typing rules for regions, memories, and machine states are defined in FIGURE 13 with each requiring only a single rule. Regions are typed with the judgment

form  $\bar{\tau} \triangleright R : \Sigma$  asserting that the region  $R$  has type  $\Sigma$  relative to the memory type  $\bar{\tau}$ . Each of the values bound in  $R$  must be closed and must have the type given by  $\Sigma$ . The domains of these maps must also coincide. Because the values stored in a particular region could contain references to any other region in the stack, each value is typed relative to the entire memory type. Memories themselves are typed via the judgment form  $\triangleright \bar{\tau} : \bar{\tau}$  asserting simply that the memory  $\bar{\tau}$  has type  $\bar{\tau}$ . Each region in  $\bar{\tau}$  must have the corresponding region type given in  $\bar{\tau}$ . These sequences must also bind the same region names in the same order.

Finally, the judgment  $\triangleright M$  asserts that the machine state  $M$  is well-formed. The machine state must consist of a well-formed memory whose type is consistent with the continuation stack. The program expression must have the type expected by the continuation, and an effect involving only regions currently in memory.

### Discussion

The design of our abstract machine draws on three primary sources: the Tofte-Talpin calculus as a canonical region-based system and execution model[52], the static semantics of the calculus of capabilities defined by Crary *et al.* [11], and the use of explicit continuations given by Harper[18]. The language  $\mathcal{RL}$  differs considerably, however, from each of these settings.

The expressive power of region systems such as the Tofte-Talpin calculus allows regions to be freed despite the presence of *dangling pointers*. This well-known feature of region systems leads to potential problems for soundness proofs based on preservation arguments. For example, the evaluation of a well-formed program could result in an

intermediate state of the form

$$(\bar{s} @ \xi \mapsto R, K \diamond (\text{pop}; [\cdot]), \text{fn } x \Rightarrow ((\text{fn } y \Rightarrow \underline{s}) (\xi, l)))$$

Subsequent evaluation will free the region named  $\xi$  leaving the dangling reference  $(\xi, l)$  embedded in the program expression. The rule (*loc-dead*) in FIGURE 10 is needed to type locations such as  $(\xi, l)$  which may occur in *dead code*.

Our treatment of dead locations is similar to that used in the capability language of Cray et al. [11]. However, preservation in both settings becomes dependent on restricting run-time name generation as well. This can be seen in the rule (*T10*) of FIGURE 9 which prevents the use of new region names for which dead locations already occur. Helsen and Thiemann solve this problem with an alternative approach based on the substitution of a distinguished dead-region identifier for all occurrences of region names such as  $\xi$  above[21]. Thus, both approaches require a slight departure from the idealized semantics of region systems. Moggi suggests that this phenomenon is an inherent limitation of approaches based on *small-step* operational semantics[38].

The semantics of our abstract machine has been defined so that the following key properties hold:

Proposition 3.1 (*RL Typability Preservation*)

If  $\triangleright M$  and  $M \mapsto M'$  then  $\triangleright M'$

Proposition 3.2 (*RL Progress*)

If  $\triangleright M$  then either  $M \mapsto M'$  or  $M \equiv (\bar{s}, [\cdot], v)$ .

We will not prove these properties here as the technical details involved are subsumed

by proofs given for the language  $\mathcal{RE}$  introduced in CHAPTER IV. The soundness of  $\mathcal{RL}$  is an easy consequence of Propositions 3.1 and 3.2:

Proposition 3.3 ( $\mathcal{RL}$  Soundness)

If  $\triangleright M$  then either  $M$  diverges or  $M \mapsto^* (\bar{\sigma}, [\cdot], v)$ .

Although soundness, as stated above, guarantees that regions are used safely, it does not guarantee that programs produce values of the expected type. By modifying the static semantics of continuations and machine states for  $\mathcal{RL}$ , one could state and prove a stronger form of soundness equating the type of the initial and final machine states. However, in the extensions we will study next, strong soundness will be either less meaningful or more awkward to prove.

## CHAPTER IV

### EXCEPTIONS

Exception handling constructs are a staple of high-level languages such as Java and Standard ML. Although often associated with error handling, exceptions more generally provide a means of “jumping” outside the normal flow of execution. In this chapter, we provide the simple intuition behind type and effect rules for idealized exception constructs. This intuition is then formalized in the language  $\mathcal{RE}$ .

#### Exceptions in the ML-Kit

Standard ML provides constructs for locally declaring, raising, and handling constructed exceptions[36]. The fact that exceptions can be locally declared is not universal to ML dialects and it leads to some subtlety in their semantics.<sup>1</sup> For example, when an exception declaration is nested immediately within a recursive function, each successive invocation of the function results in the creation of a new exception constructor. This is referred to as the *generative* nature of Standard ML exceptions.

Exceptions are the only construct leading to non-constant time operations for managing regions in the ML-Kit[49]. When an exception is raised, the (control) stack is scanned for a matching handler causing a statically unknown number of regions to be popped. Exceptions may even propagate to the top-level, in which case the name of the exception is printed. This aspect of region systems has never been formalized.

The constructs provided by Standard ML, and their treatment in relation to region-

---

<sup>1</sup>The CAML dialect allows only top-level exception declarations.

---

Source code:

```
fun f(x) =  
  let exception foo of string  
      val ex = (foo "hello")  
  in  
    (if (x=0) then (raise ex)  
      else (f(x-1)))  
    handle foo(y) => 5  
  end
```

ML-Kit Output (Simplified):

```
fun f (x)=  
  letregion r8  
  in  
    let exception foo : ((string,r8)-->(exn,r1),r1)  
        val ex : (exn,r1) = foo at r1 "hello"at r8  
    in  
      if (x=0) ...  
    end  
  end (*r8*)
```

---

FIGURE 14. ML-Kit Example

inference can be clarified by a simple example. In FIGURE 14, we give the source code for a simple recursive function along with the region-annotated output of the ML-Kit.<sup>2</sup>

The function named `f` locally declares an exception constructor named `foo`. Then an exception value is built by applying this constructor to an argument string. The function itself accepts an integer argument and simply counts down to zero via a sequence of recursive calls. At this point, the exception is raised and handled and the function returns the constant 5. Although contrived, this example demonstrates the basic features of declaring, raising, and handling constructed exceptions in ML.

The region annotated version of `f` is shown in a skeletal and heavily simplified form. First, the function declares a local region, named `r8`, into which the string `hello` will be allocated. The types for both the exception constructor and the exception value are explicitly given. The region named `r1` is *global* to the program and thus never reclaimed. Each invocation of the recursive function will result in the allocation of a new exception constructor in the region `r1`, followed by a string in the local region `r8`, and then a constructed exception value again in the region `r1`. This situation therefore leads to a space leak. The ML-Kit can be configured to emit warnings about space leaks which in this case would indicate that the function `f` has escaping effects on the region `r1`. It is important to note that although the ML-Kit has allocated both the exception constructor and exception value in a global region, the string being carried by the exception is stored in a local region which is reclaimed before the function returns. Thus, the space leak associated with exceptions is related to the small and constant amount of space required for constructors[49]. Obviously, a more serious situation would arise if any value carried by an exception were globally allocated. The resulting space leak would be proportional

---

<sup>2</sup>The ML-Kit can be configured to output various intermediate representations used by the compiler.



not only to the depth of the recursion, but also the unbounded size of the value being carried.

### Type and Effect Rules for Exceptions

Our treatment of region systems based on explicit effects will allow us to ignore the global effects associated with exceptions and focus on the issue of how ordinary data carried by exception constructors fits into the region-based execution model.

To build our intuition about the relationship between exceptions and regions, we will informally discuss an extension to  $\mathcal{RL}$  using simplified judgments of the form  $\Gamma \triangleright e : \tau, \varphi$  which are sufficient for surface terms. Our study will be based on idealized constructs for ML-like exceptions which are similar, although not identical, to those defined by Wright and Felleison [59]. Consider adding the following new phrases to our language:

$$e ::= \dots \mid \text{exception } x \text{ in } e \mid \text{raise } e_1 \text{ with } e_2 \mid \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3$$

Exception declarations introduce (unary) exception constructors. Exceptions are both constructed and raised via the construct (raise  $e_1$  with  $e_2$ ) in which  $e_1$  should evaluate to a constructor to be used with the argument  $e_2$ . The construct (try  $e_1$  handle  $e_2$  with  $x \Rightarrow e_3$ ) provides a means of handling a particular exception, given by  $e_2$ , which may be raised during the evaluation of  $e_1$ . Operationally, we assume  $e_2$  is evaluated first yielding a constructor, and then the body  $e_1$  is evaluated. If  $e_1$  produces an ordinary value then that value is delivered by the whole expression. If  $e_1$  produces an exception matching the constructor given by  $e_2$ , then the value carried by that exception is bound to  $x$  and evaluation proceeds with  $e_3$ . Thus, unless additional exceptions are raised, the expression

---


$$\begin{array}{c}
\frac{\Gamma \langle x : \tau_1 \text{ exn} \rangle \triangleright e : \tau_2, \varphi}{\Gamma \triangleright \text{exception } x \text{ in } e : \tau_2, \varphi} \\
\\
\frac{\Gamma \triangleright e_1 : \tau_1 \text{ exn}, \varphi_1 \quad \Gamma \triangleright e_2 : \tau_1, \varphi_2}{\Gamma \triangleright \text{raise } e_1 \text{ with } e_2 : \tau_2, \varphi_1 \cup \varphi_2} \\
\\
\frac{\Gamma \triangleright e_1 : \tau_1, \varphi_1 \quad \Gamma \triangleright e_2 : \tau_2 \text{ exn}, \varphi_2}{\Gamma \triangleright \text{fn } x \Rightarrow e_3 : \tau_2 \rightarrow \langle \varphi_3, \tau_1 \rangle, \emptyset} \\
\hline
\Gamma \triangleright \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3 : \tau_1, \varphi_1 \cup \varphi_2 \cup \varphi_3
\end{array}$$


---

FIGURE 15. Type and Effect Rules for Exceptions

will produce either the value of  $e_1$ , or the value of  $e_3$  whose evaluation depends on the exception argument dynamically bound to  $x$ .

The type and effect rules are shown in FIGURE 15. These rules simply propagate effect information from each sub-expression in the obvious way. Exception constructors are assigned types of the form  $(\tau \text{ exn})$  which serve as a convenient foil for the functional types of unary constructors in ML. Note that the type for expressions of the form  $(\text{raise } e_1 \text{ with } e_2)$  is completely unconstrained. This flexibility is required so that the same exception can be raised at arbitrary program points whose types may differ.

The question is now whether these simple rules are semantically sound in relation to the region-based execution model. In other words, if an exception carries a value with effects in some particular regions, a handler for that exception may manipulate that value thereby incurring those effects. Therefore the regions involved should not be collected as long as the possibility remains that the exception can be handled. Recall that regions are introduced according to the following (simplified) rule:

$$\frac{\Gamma \triangleright e : \tau, \varphi \quad \rho \notin \text{frv}(\Gamma) \quad \rho \notin \text{frv}(\tau)}{\Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}}$$

The following example, which is not typable, shows an unsound use of exceptions:

exception  $x$  in (try (letreg  $\rho$  in (raise  $x$  with (new  $\rho$   $\underline{\xi}$ ))) handle  $x$  with  $y \Rightarrow$  (get  $y$ ))

In this example, the region named  $\rho$  will be deallocated before the handler is applied, resulting in the dereferencing of a dead location. Note that because the type of the raise expression is not directly related to the type (ref  $\rho$  int) being carried, the criterion  $\rho \notin \text{frv}(\tau)$  cannot detect the fact that this reference escapes. However, because the handler must be in the scope of the exception declaration itself, the criterion  $\rho \notin \text{frv}(\Gamma)$  prevents this example from being typable. Therefore the soundness of this solution rests on the fact that exceptions can only be handled within the scope of their declarations.

### The Language $\mathcal{RE}$

Although the intuition behind extending our region systems with exceptions is simple, verifying the soundness of such an extension is rather involved. Our use of explicit continuations will pay dividends in the preservation argument for  $\mathcal{RE}$  by allowing us to reason about the unwinding of the control stack. In order to cleanly accommodate exceptions, we will require some reorganization of the syntactic categories defined in CHAPTER III for  $\mathcal{RL}$ . As a technical convenience, we will assume a new set of exception variables, ranged over by  $h$ . Exception variables will serve as names for exception constructors in the manner just described. We continue to use the notation  $\vec{h}$  to signify a sequence of distinct exception variables.

Region identifiers are the only syntactic category to remain completely unchanged from its definition in  $\mathcal{RL}$  :

$$\gamma ::= \rho \mid \xi$$

Values will now include exception variables which may be referenced, passed as function arguments, or otherwise manipulated like other first-class values:

$$v ::= x \mid h \mid \underline{n} \mid \text{fn } x \Rightarrow e \mid \text{rec } f\langle\vec{\rho}\rangle(x) \Rightarrow e \mid (\xi, l)$$

Intuitively, because programs may produce uncaught exceptions at the top-level, another form of value should be needed to represent them. Exception packets, ranged over by  $p$ , are defined by:

$$p ::= \text{fail } h \text{ with } v$$

Packets represent raised exceptions which have not yet been caught. They carry both the name of the exception and value to be passed to an appropriate handler. Terminating programs will now produce answers, ranged over by  $a$ , which are either values or exception packets:

$$a ::= v \mid p$$

Expressions are extended to include answers and the three new phrases for exceptions:

$$e ::= a \mid e\langle\vec{\gamma}\rangle \mid e_1 e_2 \mid \text{new } \gamma e \mid \text{get } e \mid \text{letreg } \rho \text{ in } e \mid \text{exception } \vec{h} \text{ in } e \mid \\ \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3 \mid \text{raise } e_1 \text{ with } e_2$$

Exception declarations are slightly different from the construct introduced earlier. Besides the switch to exception names, the declaration allows a sequence of distinct exceptions to be declared simultaneously. The significance of these choices will become clear later.

Rather than extend our definition of continuations directly, it will be useful to separate our continuation frames into groups. Applicative frames, ranged over by  $\delta$ , are

defined as:

$$\delta ::= ([\cdot] \langle \bar{\gamma} \rangle) \mid ([\cdot] e) \mid (v [\cdot]) \mid (\text{new } \gamma [\cdot]) \mid (\text{get } [\cdot]) \mid (\text{raise } [\cdot] \text{ with } e) \mid \\ (\text{raise } v \text{ with } [\cdot]) \mid (\text{try } e \text{ handle } [\cdot] \text{ with } x \Rightarrow e)$$

These include most of the frames seen previously, along with new ones to evaluate the sub-expressions of our new phrases. Note that for try-expressions, evaluation will begin with the sub-expression representing the exception being handled. Next, we define non-binding frames, ranged over by  $\kappa$ , to include the applicative ones:

$$\kappa ::= \delta \mid (\text{pop}; [\cdot]) \mid (\text{try } [\cdot] \text{ handle } v \text{ with } x \Rightarrow e)$$

These new frames play a more complex rôle in the semantics because of their relation to region management and exception handling. Finally, our revised definition of continuations is as follows:

$$K ::= [\cdot] \mid K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \mid K \diamond \kappa$$

Empty continuations are again represented by  $[\cdot]$  and are extended to the right. The main departure from our earlier definition is that the frame  $(\text{exception } \vec{h} \text{ in } [\cdot])$  is a binding construct which provides new exception names to be used in frames to the right.

Types are extended with a new type for our unary exception constructors:

$$\tau ::= \text{int} \mid \alpha \mid \text{ref } \gamma \tau \mid \tau \text{ exp} \mid \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle \mid \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau$$

All other definitions for effects, substitution, typing contexts, etc., carry over directly.

### Dynamic Semantics

The main complication involved in handling generative exceptions is related to the scope of exception declarations which must be maintained throughout evaluation. Our approach is very similar to that used Wright and Felleison [59]. Consider, for example, an application of the form

$$(\text{exception } h \text{ in } e) (\text{exception } h' \text{ in } e')$$

Because the evaluation of both  $e$  and  $e'$  could produce values with embedded occurrences of  $h$  and  $h'$ , one cannot simply remove the exception declarations leaving these names unbound. Furthermore, even if two exception names declared above were identical, that is  $h \equiv h'$ , we should not confuse them. Instead, we rely on the identification of expressions up to consistent renaming of bound variables, and the further convention that bound variables are distinct in distinct expressions. Our dynamic semantics must maintain a distinction between the exceptions declared above and allow exceptions embedded in values to propagate beyond the scope of their original declarations. This is accomplished by merging exception declarations and then lifting them, whenever necessary, across intervening continuation frames. The values stored in memory may contain occurrences of any exception declared within the frames of the current continuation.

The transition rules for  $\mathcal{RE}$  are defined in FIGURES 16 and 17. Again the rules are grouped according to their function. The computation rules are identical to those defined in CHAPTER III. The decomposition include those previously seen but have been extended to handle the sub-expressions occurring in our exception constructs. The region management rules differ only because an answer may appear as the result of

## [Decomposition]

- (T1)  $(\bar{s}, K, e \langle \bar{\gamma} \rangle) \mapsto (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), e)$
- (T2)  $(\bar{s}, K, e_1 e_2) \mapsto (\bar{s}, K \diamond ([\cdot] e_2), e_1)$
- (T3)  $(\bar{s}, K \diamond ([\cdot] e), v) \mapsto (\bar{s}, K \diamond (v [\cdot]), e)$
- (T4)  $(\bar{s}, K, \text{new } \gamma e) \mapsto (\bar{s}, K \diamond (\text{new } \gamma [\cdot]), e)$
- (T5)  $(\bar{s}, K, \text{get } e) \mapsto (\bar{s}, K \diamond (\text{get } [\cdot]), e)$
- (T6)  $(\bar{s}, K, \text{exception } \vec{h} \text{ in } e) \mapsto (\bar{s}, K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]), e)$
- (T7)  $(\bar{s}, K, \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3) \mapsto (\bar{s}, K \diamond (\text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_3), e_2)$
- (T8)  $(\bar{s}, K \diamond (\text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_2), v) \mapsto (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } v \text{ with } x \Rightarrow e_2), e_1)$
- (T9)  $(\bar{s}, K, \text{raise } e_1 \text{ with } e_2) \mapsto (\bar{s}, K \diamond (\text{raise } [\cdot] \text{ with } e_2), e_1)$
- (T10)  $(\bar{s}, K \diamond (\text{raise } [\cdot] \text{ with } e), v) \mapsto (\bar{s}, K \diamond (\text{raise } v \text{ with } [\cdot]), e)$

## [Lifting and Merging]

- (T11)  $(\bar{s}, K \diamond \kappa \diamond (\text{exception } \vec{h} \text{ in } [\cdot]), a) \mapsto (\bar{s}, K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond \kappa, a)$
- (T12)  $(\bar{s}, K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond (\text{exception } \vec{h}' \text{ in } [\cdot]), a) \mapsto (\bar{s}, K \diamond (\text{exception } \vec{h}, \vec{h}' \text{ in } [\cdot]), a)$

FIGURE 16. Transition Rules for  $\mathcal{RE}$  (Part 1)

## [Computation]

$$(T13) \quad (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e) \xrightarrow{\quad} (\bar{s}, K, (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e)][\bar{\rho} := \bar{\gamma}])$$

$$(T14) \quad (\bar{s}, K \diamond ((\text{fn } x \Rightarrow e) [\cdot]), v) \xrightarrow{\quad} (\bar{s}, K, e[x := v])$$

$$(T15) \quad (\bar{s}, K \diamond (\text{new } \xi [\cdot]), v) \xrightarrow{\quad} (\bar{s} \langle \xi \mapsto R(l \mapsto v) \rangle, K, (\xi, l)) \text{ if } \bar{s}(\xi) = R \text{ and } l \notin \text{dom}(R)$$

$$(T16) \quad (\bar{s}, K \diamond (\text{get } [\cdot]), (\xi, l)) \xrightarrow{\quad} (\bar{s}, K, v) \text{ if } \bar{s}(\xi)(l) = v$$

## [Region Management]

$$(T17) \quad (\bar{s}, K, \text{letreg } \rho \text{ in } e) \xrightarrow{\quad} (\bar{s} @ \xi \mapsto \emptyset, K \diamond (\text{pop}; [\cdot]), e[\rho := \xi]) \text{ if } \xi \notin \text{frn}(\bar{s}) \cup \text{frn}(e)$$

$$(T18) \quad (\bar{s} @ \xi \mapsto R, K \diamond (\text{pop}; [\cdot]), a) \xrightarrow{\quad} (\bar{s}, K, a)$$

## [Raising, Unwinding, Handling, and Unhandling]

$$(T19) \quad (\bar{s}, K \diamond (\text{raise } h \text{ with } [\cdot]), v) \xrightarrow{\quad} (\bar{s}, K, \text{fail } h \text{ with } v)$$

$$(T20) \quad (\bar{s}, K \diamond \delta, p) \xrightarrow{\quad} (\bar{s}, K, p)$$

$$(T21) \quad (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e), p) \xrightarrow{\quad} (\bar{s}, K, (\text{fn } x \Rightarrow e) v) \text{ if } p \equiv (\text{fail } h \text{ with } v)$$

$$(T22) \quad (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e), p) \xrightarrow{\quad} (\bar{s}, K, p) \text{ if } p \equiv (\text{fail } h' \text{ with } v) \text{ and } h \not\equiv h'$$

$$(T23) \quad (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e_2), v) \xrightarrow{\quad} (\bar{s}, K, v)$$

FIGURE 17. Transition Rules for  $\mathcal{RE}$  (Part 2)



evaluating a let-region expression, rather than an ordinary value. Whenever progress is blocked by an intervening exception declaration, that declaration is exchanged with the preceding frame by rule (T11). If the preceding frame is another exception declaration, then the two are merged according to rule (T12).

The rules related to raising exceptions are more interesting. When an exception is raised, a packet is formed by rule (T19). The distinction between values and packets is important because packet formation leads to the unwinding of the continuation in search of an appropriate handler as expressed by rule (T20). Any intervening (pop; [·]) frames will also result in the collection of regions according to rule (T18). When a handler is found, it either matches the given packet or is ignored according to rules (21) and (22).

### Static Semantics

The static semantics of  $\mathcal{RE}$  is defined in FIGURES 18, 19, 20, and 21. These judgments rely on a new form of environment for maintaining assumptions about exception names. Exception environments, ranged over by  $\Phi$ , are finite maps from exception variables to types. These maps are subject to our usual notational conventions for typing environments such as functional updates, written  $\Phi\langle h : \tau \rangle$ . Informally, if  $\Phi(h) = \tau$  then the exception name  $h$  is assumed to have type  $(\tau \text{ exn})$ . Each of our previous judgment forms, other than  $\triangleright M$ , will be modified to carry exception environments.

The typing rules for answers are defined in FIGURE 18 with judgments of the form  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau$ . The values defined for  $\mathcal{RL}$  are included among  $\mathcal{RE}$  answers and each of our previous rules has been simply adapted to the new judgment form. In analogy to (*var*), the rule (*constr*) types exception names according to the assumption given in the exception environment. The rule (*packet*) types uncaught exceptions in

---

(var)	$\frac{\Gamma(x) = \tau}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright x : \tau}$
(constr)	$\frac{\Phi(h) = \tau}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright h : \tau \text{ exn}}$
(num)	$\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \underline{n} : \text{int}$
(abs)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \langle x : \tau_1 \rangle \triangleright e : \tau_2, \varphi_1 \quad \varphi_1 \subseteq \varphi_2 \quad fcv(\tau_1) \cup fcv(\varphi_2) \subseteq \Delta \quad x \notin \text{dom}(\Gamma)}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{fn } x \Rightarrow e : \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle}$
(rec)	$\frac{\bar{\zeta}; \Delta \boxplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Phi; \Gamma \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau \rangle \triangleright \text{fn } x \Rightarrow e : \tau \quad fcv(\forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau) \subseteq \Delta \quad f \notin \text{dom}(\Gamma)}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau}$
(loc-live)	$\frac{\bar{\zeta}(\xi)(l) = \tau \quad fcv(\tau) = \emptyset}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright (\xi, l) : \tau}$
(loc-dead)	$\frac{\xi \notin \text{dom}(\bar{\zeta}) \quad fcv(\tau) = \emptyset}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright (\xi, l) : \tau}$
(packet)	$\frac{\Phi(h) = \tau_1 \quad \bar{\zeta}; \Delta; \Phi; \Gamma \triangleright v : \tau_1 \quad fcv(\tau_2) \subseteq \Delta}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{fail } h \text{ with } v : \tau_2}$

---

FIGURE 18. Static Semantics for  $\mathcal{RE}$  (Part 1)

roughly the same manner as we saw for raise-expressions. In particular, the type of the overall expression is not related to the type of the exception involved. It is constrained only to be a closed type.

The typing rules for expressions are defined in FIGURE 19 with judgments of the form  $\bar{\tau}; \Delta; \Phi; \Gamma \triangleright a : \tau, \varphi$ . Again, each of our previous rules has been modified to carry an exception environment and the rule (*ans*) replaces (*var*). The rules (*exndec*), (*try*), and (*raise*) are appropriately modified versions of the rules given in FIGURE 15. The main difference is our allowance for multiple exceptions to be declared in (*exndec*) to accommodate the merging of exception declarations.

The typing rules for continuations are defined in FIGURES 20 and 21 with judgments of the form  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau$ . Intuitively, continuations now provide a context for values of a particular type,  $\tau$ , which may also mention any exceptions declared in  $\Phi$ . The exception context is extended by the rule (*k-exndec*) each time a declaration frame is encountered. These new declarations also apply to any expression embedded in any continuation frame to the right.

The modified rules for typing memories, regions, and machine states are defined in FIGURE 22. Because exception variables are first-class values, they may themselves be allocated, or appear embedded in other values allocated in memory. The exception environment appearing in the type of the current continuation in the rule (*prog*) contains bindings for every exception declaration encountered during evaluation. This environment is used to type memory, via the judgment form  $\Phi \triangleright \bar{\tau} : \bar{\tau}$ , and each memory region, via the judgment form  $\bar{\tau}; \Phi \triangleright R : \Sigma$ .

---

(ans)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau, \emptyset}$
(papp)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau, \varphi}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e \langle \vec{\gamma} \rangle : \tau[\vec{\rho} := \vec{\gamma}, \vec{\epsilon} := \vec{\varphi}, \vec{\alpha} := \vec{\tau}], \varphi}$
(app)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e_1 : \tau_1 \rightarrow \langle \varphi_1, \tau_2 \rangle, \varphi_2 \quad \bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e_2 : \tau_1, \varphi_3}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e_1 e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi_3}$
(new)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi \quad frv(\gamma) \subseteq \Delta}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{new } \gamma e : \text{ref } \gamma \tau, \varphi \cup \{\gamma\}}$
(get)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \text{ref } \gamma \tau, \varphi}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi \cup \{\gamma\}}$
(letreg)	$\frac{\bar{\zeta}; \Delta \uplus \{\rho\}; \Phi; \Gamma \triangleright e : \tau, \varphi \quad \rho \notin frv(\tau)}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}}$
(exndec)	$\frac{\vec{h} \equiv h_1, \dots, h_k \quad \{\vec{h}\} \cap \text{dom}(\Phi) = \emptyset}{\bar{\zeta}; \Delta; \Phi \langle h_1 : \tau_1 \rangle \dots \langle h_k : \tau_k \rangle; \Gamma \triangleright e : \tau, \varphi}$ $\frac{fcv(\tau_1) \cup \dots \cup fcv(\tau_k) \subseteq \Delta}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{exception } \vec{h} \text{ in } e : \tau, \varphi}$
(try)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e_1 : \tau_1, \varphi_1 \quad \bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e_2 : \tau_2 \text{ expn}, \varphi_2}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{fn } x \Rightarrow e_2 : \tau_2 \rightarrow \langle \varphi_3, \tau_1 \rangle}$ $\frac{}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3 : \tau_1, \varphi_1 \cup \varphi_2 \cup \varphi_3}$
(raise)	$\frac{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e_1 : \tau_1 \text{ expn}, \varphi_1}{\bar{\zeta}; \Delta; \Phi; \Gamma; e_2 \triangleright \tau_1 : \varphi_2, \quad fcv(\tau_2) \subseteq \Delta}$ $\frac{}{\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright \text{raise } e_1 \text{ with } e_2 : \tau_2, \varphi_1 \cup \varphi_2}$

---

FIGURE 19. Static Semantics for  $\mathcal{RE}$  (Part 2)

---

(k-empty)	$\frac{\text{frn}(\tau) \subseteq \text{dom}(\bar{\tau}) \quad \text{fcv}(\tau) = \emptyset}{\bar{\tau} \triangleright [\cdot] \rightsquigarrow \langle \rangle, \tau}$
(k-papp)	$\frac{\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]}{\bar{\tau} \triangleright K \circ ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \Phi, \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau}$
(k-app-l)	$\frac{\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2 \quad \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \tau_1, \varphi_1 \quad \varphi_1 \cup \varphi_2 \subseteq \text{dom}(\bar{\tau})}{\bar{\tau} \triangleright K \circ ([\cdot] e) \rightsquigarrow \Phi, \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle}$
(k-app-r)	$\frac{\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2 \quad \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle \quad \varphi \subseteq \text{dom}(\bar{\tau})}{\bar{\tau} \triangleright K \circ (v [\cdot]) \rightsquigarrow \Phi, \tau_1}$
(k-new)	$\frac{\bar{\tau} \triangleright K \rightsquigarrow \Phi, \text{ref } \gamma \tau \quad \gamma \in \text{dom}(\bar{\tau})}{\bar{\tau} \triangleright K \circ (\text{new } \gamma [\cdot]) \rightsquigarrow \Phi, \tau}$
(k-get)	$\frac{\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau \quad \gamma \in \text{dom}(\bar{\tau})}{\bar{\tau} \triangleright K \circ (\text{get } [\cdot]) \rightsquigarrow \Phi, \text{ref } \gamma \tau}$
(k-pop)	$\frac{\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau}{\bar{\tau} @ \xi \mapsto \Sigma \triangleright K \circ (\text{pop}; [\cdot]) \rightsquigarrow \Phi, \tau}$

---

FIGURE 20. Static Semantics for  $\mathcal{RE}$  (Part 3)

---


$$\begin{array}{l}
(k\text{-exndec}) \quad \frac{\zeta \triangleright K \rightsquigarrow \Phi, \tau \quad \vec{h} \equiv h_1, \dots, h_k \\
\{\vec{h}\} \cap \text{dom}(\Phi) = \emptyset \quad \Phi' = \Phi \langle h_1 : \tau_1 \rangle \cdots \langle h_k : \tau_k \rangle \\
\text{fcv}(\tau_1) \cup \cdots \cup \text{fcv}(\tau_k) = \emptyset}{\zeta \triangleright K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \rightsquigarrow \Phi', \tau} \\
\\
(k\text{-try-h}) \quad \frac{\zeta \triangleright K \rightsquigarrow \Phi, \tau_1 \quad \zeta; \emptyset; \Phi; \langle \rangle \triangleright e_1 : \tau_1, \varphi_1 \\
\zeta; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e_2 : \tau_2 \rightarrow \langle \varphi_2, \tau_1 \rangle \quad \varphi_1 \cup \varphi_2 \subseteq \text{dom}(\zeta)}{\zeta \triangleright K \diamond (\text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_2) \rightsquigarrow \Phi, \tau_2 \text{ expn}} \\
\\
(k\text{-try-b}) \quad \frac{\zeta \triangleright K \rightsquigarrow \Phi, \tau_1 \quad \zeta; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_2 \text{ expn} \\
\zeta; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e : \tau_2 \rightarrow \langle \varphi, \tau_1 \rangle \quad \varphi \subseteq \text{dom}(\zeta)}{\zeta \triangleright K \diamond (\text{try } [\cdot] \text{ handle } v \text{ with } x \Rightarrow e) \rightsquigarrow \Phi, \tau_1} \\
\\
(k\text{-raise-l}) \quad \frac{\zeta \triangleright K \rightsquigarrow \Phi, \tau_2 \quad \zeta; \emptyset; \Phi; \langle \rangle \triangleright e : \tau_1, \varphi \quad \varphi \subseteq \text{dom}(\zeta)}{\zeta \triangleright K \diamond (\text{raise } [\cdot] \text{ with } e) \rightsquigarrow \Phi, \tau_1 \text{ expn}} \\
\\
(k\text{-raise-r}) \quad \frac{\zeta \triangleright K \rightsquigarrow \Phi, \tau_2 \quad \zeta; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1 \text{ expn}}{\zeta \triangleright K \diamond (\text{raise } v \text{ with } [\cdot]) \rightsquigarrow \Phi, \tau_1}
\end{array}$$


---

FIGURE 21. Static Semantics for  $\mathcal{RE}$  (Part 4)

---


$$\begin{array}{l}
(\text{region}) \quad \frac{\text{dom}(R) = \text{dom}(\Sigma) \\
\zeta; \emptyset; \Phi; \langle \rangle \triangleright R(l) : \Sigma(l) \quad (\forall l \in \text{dom}(R))}{\zeta; \Phi \triangleright R : \Sigma} \\
\\
(\text{stack}) \quad \frac{\vec{s} = \xi_1 \mapsto R_1, \dots, \xi_k \mapsto R_k \\
\vec{\zeta} = \xi_1 \mapsto \Sigma_1, \dots, \xi_k \mapsto \Sigma_k \\
\zeta; \Phi \triangleright \vec{s}(\xi) : \vec{\zeta}(\xi) \quad (\forall \xi \in \text{dom}(\vec{s}))}{\Phi \triangleright \vec{s} : \vec{\zeta}} \\
\\
(\text{prog}) \quad \frac{\Phi \triangleright \vec{s} : \vec{\zeta} \quad \zeta \triangleright K \rightsquigarrow \Phi, \tau \\
\zeta; \emptyset; \Phi; \langle \rangle \triangleright e : \tau, \varphi \quad \varphi \subseteq \text{dom}(\zeta)}{\triangleright (\vec{s}, K, e)}
\end{array}$$


---

FIGURE 22. Static Semantics for  $\mathcal{RE}$  (Part 5)

## CHAPTER V

SOUNDNESS OF  $\mathcal{RE}$ 

Before proving preservation and progress, Propositions 5.10 and 5.11, we will require a number of auxiliary properties. These properties have simple inductive proofs and, for the most part, have appeared in some form in previous work involving syntactic soundness proofs and region systems[11, 59].

Basic Properties

The following Lemma expresses a simple but important invariant maintained by the static semantics, namely, that types and effects are well-formed with respect to the constructor environment:

Lemma 5.1 ( $\mathcal{RE}$  Proper Typing)

Suppose  $fcv(\Phi) \cup fcv(\Gamma) \subseteq \Delta$ .

1. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau$  then  $fcv(\tau) \subseteq \Delta$ .
2. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$  then  $fcv(\tau) \cup fcv(\varphi) \subseteq \Delta$ .
3. If  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau$  then  $fcv(\Phi) \cup fcv(\tau) = \emptyset$ .

PROOF: Parts 1 and 2 are established by mutual induction on the derivations with all cases following directly from the induction hypothesis. Part 3 is by induction on the derivation using parts 1 and 2.

□

As a technical convenience, we state the following Lemmas related exception declarations. These Lemmas will be used in the proof of preservation for the rules (T6) and (T11):

**Lemma 5.2** ( $\mathcal{RE}$  Exception Context Extension)

Suppose  $\{h_1, \dots, h_k\} \cap \text{dom}(\Phi) = \emptyset$  and let  $\Phi' = \Phi \langle h_1 : \tau_1 \rangle \cdots \langle h_k : \tau_k \rangle$ .

1. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau$  then  $\bar{\zeta}; \Delta; \Phi'; \Gamma \triangleright a : \tau$ .
2. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$  then  $\bar{\zeta}; \Delta; \Phi'; \Gamma \triangleright e : \tau, \varphi$ .
3. If  $\Phi \triangleright \bar{s} : \bar{\zeta}$  then  $\Phi' \triangleright \bar{s} : \bar{\zeta}$ .

PROOF: Parts 1 and 2 are straightforward to establish by mutual induction.

Part 3 follows from part 1 by inspection of the rules (*stack*) and (*region*).

□

**Lemma 5.3** ( $\mathcal{RE}$  Exception Declaration Lifting)

If  $\bar{\zeta} \triangleright K \diamond \kappa \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \rightsquigarrow \Phi, \tau$  then

$\bar{\zeta} \triangleright K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond \kappa \rightsquigarrow \Phi, \tau$ .

PROOF: By case analysis on  $\kappa$  using Lemma 5.2.

□

Value and constructor substitution properties, in some form, are standard parts of syntactic soundness proofs. We require constructor substitution not only in the preservation case for polymorphic applications, rule (T13), but also in the case for region allocation, rule (T17).



**Lemma 5.4** (*RE Value Substitution*)

Suppose  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau$  and let  $a' \equiv a[x := v]$  and  $e' \equiv e[x := v]$ .

1. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \langle x : \tau \rangle \triangleright a : \tau'$  then  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a' : \tau'$ .
2. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \langle x : \tau \rangle \triangleright e : \tau', \varphi$  then  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e' : \tau', \varphi$ .

PROOF: By straightforward mutual induction on the derivations. Note that in cases involving axioms such as for rule (*num*), we use the fact that any variable not occurring free in an expression may be removed from the typing context  $\Gamma$ . Similarly,  $v$  above cannot be a variable since its judgment refers to an empty typing context. We do not formally state these properties here.

□

**Lemma 5.5** (*RE Constructor Substitution*)

Suppose  $fcv(\bar{\gamma}) \cup fcv(\bar{\varphi}) \cup fcv(\bar{\tau}) \subseteq \Delta$  and let

- (a1)  $\Phi' = \Phi[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,
- (a2)  $\Gamma' = \Gamma[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,
- (a3)  $e' = e[\bar{\rho} := \bar{\gamma}]$ ,
- (a4)  $a' = a[\bar{\rho} := \bar{\gamma}]$ ,
- (a5)  $\varphi' = \varphi[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}]$ , and
- (a6)  $\tau' = \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ .

1. If  $\bar{\zeta}; \Delta \uplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Phi; \Gamma \triangleright a : \tau$  then  $\bar{\zeta}; \Delta; \Phi'; \Gamma' \triangleright a' : \tau'$ .
2. If  $\bar{\zeta}; \Delta \uplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Phi; \Gamma \triangleright e : \tau, \varphi$  then  $\bar{\zeta}; \Delta; \Phi'; \Gamma' \triangleright e' : \tau', \varphi'$ .

PROOF: By mutual induction on the derivations. Note that the constructor substitution does not apply to the memory type because the premises of the rules (*loc-live*) and (*loc-dead*) allow only closed types to be used.

□

The following Lemmas are used in the preservation proof for the cases involving allocation and collection of entire regions given by rules (T17) and (T18) respectively. Note that these Lemmas do not refer to the continuation judgment form because these operations do not effect the typing of expressions already embedded in the continuation stack.

**Lemma 5.6 ( $\mathcal{RE}$  Region Allocation)**

Suppose  $\xi \notin \text{frn}(a)$  and  $\xi \notin \text{frn}(e)$  and  $\xi \notin \text{frn}(\bar{s})$  and let  $\bar{\zeta}' = \bar{\zeta} @ \xi \mapsto \emptyset$ .

1. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau$  then  $\bar{\zeta}'; \Delta; \Phi; \Gamma \triangleright a : \tau$ .
2. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$  then  $\bar{\zeta}'; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$ .
3. If  $\Phi \triangleright \bar{s} : \bar{\zeta}$  then  $\Phi \triangleright \bar{s} @ \xi \mapsto \langle \rangle : \bar{\zeta}'$ .

PROOF: Parts 1 and 2 are established by mutual induction on the derivations. The conditions restricting  $\xi$ 's occurrence are required in the case for dead locations. Part 3 follows by inspection of the rules (*stack*) and (*region*) using part 1.

□

**Lemma 5.7 ( $\mathcal{RE}$  Region Collection)**

Let  $\bar{\zeta} = \bar{\zeta}' @ \xi \mapsto \Sigma$ .

1. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau$  then  $\bar{\zeta}'; \Delta; \Phi; \Gamma \triangleright a : \tau$ .
2. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$  then  $\bar{\zeta}'; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$ .
3. If  $\Phi \triangleright \bar{s} @ \xi \mapsto R : \bar{\zeta}$  then  $\Phi \triangleright \bar{s} : \bar{\zeta}'$ .

PROOF: Parts 1 and 2 are established by mutual induction on the derivations. Note that the rule (*loc-dead*) is required when  $a \equiv (\xi', l)$  was derived by (*loc-live*), that is, when  $\xi' \equiv \xi$ . Part 3 follows by inspection of the rules (*stack*) and (*region*) using part 1.

□

The proof of preservation for reference allocation, rule (*T15*), requires the following Lemma stating that memory types can be extended with new locations:

**Lemma 5.8** ( $\mathcal{RE}$  Reference Allocation)

Suppose  $\bar{\zeta}(\xi) = \Sigma$  and  $l \notin \text{dom}(\Sigma)$  and let  $\bar{\zeta}' = \bar{\zeta} \langle \xi \mapsto \Sigma \langle l \mapsto \tau \rangle \rangle$  and  $\bar{s}' = \bar{s} \langle \xi \mapsto \bar{s}(\xi) \langle l \mapsto v \rangle \rangle$ .

1. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright a : \tau$  then  $\bar{\zeta}'; \Delta; \Phi; \Gamma \triangleright a : \tau$ .
2. If  $\bar{\zeta}; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$  then  $\bar{\zeta}'; \Delta; \Phi; \Gamma \triangleright e : \tau, \varphi$ .
3. If  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau$  then  $\bar{\zeta}' \triangleright K \rightsquigarrow \Phi, \tau$ .
4. If  $\Phi \triangleright \bar{s} : \bar{\zeta}$  and  $\bar{\zeta}'; \emptyset; \Phi; \langle \rangle \triangleright v : \tau$  then  $\Phi \triangleright \bar{s}' : \bar{\zeta}'$ .

PROOF: Parts 1 and 2 are established by mutual induction on the derivations. Part 3 is by induction on the derivation using parts 1 and 2. Part 4 follows by inspection of the rules (*stack*) and (*region*) using part 1.

□

Finally, the following standard property summarizes the syntactic forms that values can have based on their types and is useful in the proof of Proposition 5.11:

**Lemma 5.9** ( $\mathcal{RE}$  Canonical Forms)

Suppose  $\Phi \triangleright \bar{s} : \bar{\tau}$  and  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau$ .

1. If  $\tau \equiv \tau'$  *exn* then  $v \equiv h$ .
2. If  $\tau \equiv \text{ref } \xi \tau'$  then  $v \equiv (\xi, l)$  and either  $\xi \notin \text{dom}(\bar{s})$  or  $\bar{s}(\xi)(l) = v$ .
3. If  $\tau \equiv \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle$  then  $v \equiv (\text{fn } x \Rightarrow e)$ .
4. If  $\tau \equiv \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau'$  then  $v \equiv (\text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e)$ .

PROOF: The proof is by inspection of the typing rules for values using the fact that  $v \neq x$  since the judgment for  $v$  has an empty typing context.

□

**Preservation and Progress**

**Proposition 5.10** ( $\mathcal{RE}$  Typability Preservation)

If  $\triangleright M$  and  $M \mapsto M'$  then  $\triangleright M'$

PROOF:

By case analysis on the transition rules.

Case (T1)

Assume  $\triangleright (\bar{s}, K, e \langle \bar{\gamma} \rangle)$ . From the premises of (*prog*) and (*p-app*)

(a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,

(b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,

(c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e \langle \bar{\tau} \rangle : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}], \varphi,$

(d)  $\varphi \subseteq \text{dom}(\bar{\tau}),$  and

(e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau, \varphi.$

By (*k-papp*) with (b)

(f)  $\bar{\tau} \triangleright K \diamond([\cdot] \langle \bar{\tau} \rangle) \rightsquigarrow \Phi, \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau.$

We conclude  $\triangleright (\bar{s}, K \diamond([\cdot] \langle \bar{\tau} \rangle), e)$  by (*prog*) with (a), (d), (e), and (f).

Case (T2)

Assume  $\triangleright (\bar{s}, K, e_1 e_2).$  From the premises of (*prog*) and (*app*)

(a)  $\Phi \triangleright \bar{s} : \bar{\tau},$

(b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2,$

(c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e_1 e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi_3,$

(d)  $\varphi_1 \cup \varphi_2 \cup \varphi_3 \subseteq \text{dom}(\bar{\tau}),$

(e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e_1 : \tau_1 \rightarrow \langle \varphi_1, \tau_2 \rangle, \varphi_2,$  and

(f)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e_2 : \tau_1, \varphi_3.$

From (d) we know  $\varphi_1 \cup \varphi_3 \subseteq \text{dom}(\bar{\tau})$  so by (*k-app-l*) with (b) and (f)

(g)  $\bar{\tau} \triangleright K \diamond([\cdot] e_2) \rightsquigarrow \Phi, \tau_1 \rightarrow \langle \varphi_1, \tau_2 \rangle.$

Since (d) also implies  $\varphi_2 \subseteq \text{dom}(\bar{\tau})$  we conclude  $\triangleright (\bar{s}, K \diamond([\cdot] e_2), e_1)$  by (*prog*) with (a), (e), and (g).

Case (T3)

Assume  $\triangleright (\bar{s}, K \diamond([\cdot] e), v).$  From the premises of (*prog*), (*k-app-l*), and (*ans*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \diamond ([\cdot] e) \rightsquigarrow \Phi, \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle, \emptyset$ ,
- (d)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2$ ,
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \tau_1, \varphi_1$ ,
- (f)  $\varphi_1 \cup \varphi_2 \subseteq \text{dom}(\bar{\tau})$ , and
- (g)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1 \rightarrow \langle \varphi_2, \tau_2 \rangle$ .

From (f) we know  $\varphi_2 \subseteq \text{dom}(\bar{\tau})$  so by (*k-app-r*) with (d) and (e)

- (h)  $\bar{\tau} \triangleright K \diamond (v [\cdot]) \rightsquigarrow \Phi, \tau_1$ .

Since (f) also implies  $\varphi_1 \subseteq \text{dom}(\bar{\tau})$  we conclude  $\triangleright (\bar{s}, K \diamond (v [\cdot]), e)$  by (*prog*) with (a), (e), and (h).

Case (*T4*)

Assume  $\triangleright (\bar{s}, K, \text{new } \gamma e)$ . From the premises of (*prog*) and (*new*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \text{ref } \gamma \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{new } \gamma e : \text{ref } \gamma \tau, \varphi \cup \{\gamma\}$ ,
- (d)  $\varphi \cup \{\gamma\} \subseteq \text{dom}(\bar{\tau})$ , and
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \tau, \varphi$ .

From (d) we know  $\gamma \in \text{dom}(\bar{\tau})$  so by (*k-new*) with (b)

- (f)  $\bar{\tau} \triangleright K \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \Phi, \tau$ .

Since (d) also implies  $\varphi \subseteq \text{dom}(\bar{\tau})$  we conclude  $\triangleright (\bar{s}, K \diamond (\text{new } \gamma [\cdot]), e)$  by (*prog*) with (a), (e), and (f).

Case (T5)

Assume  $\triangleright (\bar{s}, K, \text{get } e)$ . From the premises of (*prog*) and (*get*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{get } e : \tau, \varphi \cup \{\gamma\}$ ,
- (d)  $\varphi \cup \{\gamma\} \subseteq \text{dom}(\bar{\tau})$ , and
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \text{ref } \gamma \tau, \varphi$ .

From (d) we know  $\gamma \in \text{dom}(\bar{\tau})$  so by (*k-get*) with (b)

- (f)  $\bar{\tau} \triangleright K \diamond (\text{get } [\cdot]) \rightsquigarrow \Phi, \text{ref } \gamma \tau$ .

Since (d) also implies  $\varphi \subseteq \text{dom}(\bar{\tau})$  we conclude  $\triangleright (\bar{s}, K \diamond (\text{get } [\cdot]), e)$  by (*prog*) with (a), (e), and (f).

Case (T6)

Assume  $\triangleright (\bar{s}, K, \text{exception } \vec{h} \text{ in } e)$  and  $\vec{h} \equiv h_1, \dots, h_k$ . From the premises of (*prog*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{exception } \vec{h} \text{ in } e : \tau, \varphi$ ,
- (d)  $\varphi \subseteq \text{dom}(\bar{\tau})$ , and
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \text{ref } \gamma \tau, \varphi$ .

Let  $\Phi' = \Phi \langle h_1 : \tau_1 \rangle \cdots \langle h_k \mapsto \tau_k \rangle$ . From the premises of (*exndec*) with (e)

- (f)  $\{\vec{h}\} \cap \text{dom}(\Phi) = \emptyset$ ,
- (g)  $\bar{\zeta}; \emptyset; \Phi'; \langle \rangle \triangleright e : \tau, \varphi$ , and
- (h)  $\text{fcv}(\tau_1) \cup \cdots \cup \text{fcv}(\tau_k) = \emptyset$ .

By (*k-exndec*) with (b), (f), and (h)

- (i)  $\bar{\zeta} \triangleright K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \rightsquigarrow \Phi', \tau$ .

By Lemma 5.2 with (a) and (f)

- (j)  $\Phi' \triangleright \bar{s} : \bar{\zeta}$ .

We conclude  $\triangleright (\bar{s}, K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]), e)$  by (*prog*) with (d), (g), (i), and (j).

Case (*T7*)

Assume  $\triangleright (\bar{s}, K, \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3)$ . From the premises of (*prog*) and (*try*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau_1$ ,
- (c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{try } e_1 \text{ handle } e_2 \text{ with } x \Rightarrow e_3 : \tau_1, \varphi_1 \cup \varphi_2 \cup \varphi_3$ ,
- (d)  $\varphi_1 \cup \varphi_2 \cup \varphi_3 \subseteq \text{dom}(\bar{\zeta})$ ,
- (e)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright e_1 : \tau_1, \varphi_1$ ,
- (f)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright e_2 : \tau_2 \text{ exn}, \varphi_2$ , and
- (g)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e_3 : \tau_2 \rightarrow \langle \varphi_3, \tau_1 \rangle$ .



From (d) we know  $\varphi_1 \cup \varphi_3 \subseteq \text{dom}(\bar{\zeta})$  so by (*k-try-h*) with (b), (e), and (g)

$$(h) \bar{\zeta} \triangleright K \diamond (\text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_3) \rightsquigarrow \Phi, \tau_2 \text{ exn.}$$

We conclude  $\triangleright (\bar{s}, K \diamond (\text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_3), e_2)$  by (*prog*) with (a), (f), and (h) and the fact that (d) implies  $\varphi_2 \subseteq \text{dom}(\bar{\zeta})$ .

Case (T8)

Assume  $\triangleright (\bar{s}, K \diamond \text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_2, v)$ . From the premises of (*prog*), (*k-try-h*), and (*ans*)

$$(a) \Phi \triangleright \bar{s} : \bar{\zeta},$$

$$(b) \bar{\zeta} \triangleright K \diamond \text{try } e_1 \text{ handle } [\cdot] \text{ with } x \Rightarrow e_2 \rightsquigarrow \Phi, \tau_2 \text{ exn,}$$

$$(c) \bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_2 \text{ exn, } \emptyset,$$

$$(d) \bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright e_1 : \tau_1, \varphi_1,$$

$$(e) \bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau_1,$$

$$(f) \bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e_2 : \tau_2 \rightarrow \langle \varphi_2, \tau_1 \rangle,$$

$$(g) \varphi_1 \cup \varphi_2 \subseteq \text{dom}(\bar{\zeta}), \text{ and}$$

$$(h) \bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_2 \text{ exn.}$$

From (f) we know  $\varphi_2 \subseteq \text{dom}(\bar{\zeta})$  so by (*k-try-b*) with (e), (f), and (h)

$$(i) \bar{\zeta} \triangleright K \diamond (\text{try } [\cdot] \text{ handle } v \text{ with } x \Rightarrow e_2) \rightsquigarrow \Phi, \tau_1.$$

We conclude  $\triangleright (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } v \text{ with } x \Rightarrow e_2), e_1)$  (*prog*) with (a), (d), and (i) and the fact that (g) implies  $\varphi_1 \subseteq \text{dom}(\bar{\zeta})$ .

Case (T9)

Assume  $\triangleright (\bar{s}, K, \text{raise } e_1 \text{ with } e_2)$ . From the premises of (*prog*) and (*raise*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{raise } e_1 \text{ with } e_2 : \tau_2, \varphi_1 \cup \varphi_2$ ,
- (d)  $\varphi_1 \cup \varphi_2 \subseteq \text{dom}(\bar{\tau})$ ,
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e_1 : \tau_1 \text{ exn}, \varphi_1$ , and
- (f)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e_2 : \tau_1, \varphi_2$ .

From (d) we know  $\varphi_2 \subseteq \text{dom}(\bar{\tau})$  so by (*k-raise-l*) with (b) and (e)

- (g)  $\bar{\tau} \triangleright K \circ (\text{raise } [\cdot] \text{ with } e_2) \rightsquigarrow \Phi, \tau_1 \text{ exn}$ .

We conclude  $\triangleright (\bar{s}, K \circ (\text{raise } [\cdot] \text{ with } e_2), e_1)$  by (*prog*) with (a), (e), and (g) and the fact that (d) implies  $\varphi_1 \subseteq \text{dom}(\bar{\tau})$ .

Case (*T10*)

Assume  $\triangleright (\bar{s}, K \circ (\text{raise } [\cdot] \text{ with } e), v)$  From the premises of (*prog*), (*k-raise-l*), and (*ans*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \circ \text{raise } [\cdot] \text{ with } e \rightsquigarrow \Phi, \tau_1 \text{ exn}$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1 \text{ exn}, \emptyset$ ,
- (d)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2$ ,
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e : \tau_1, \varphi$ ,
- (f)  $\varphi \subseteq \text{dom}(\bar{\tau})$ , and
- (g)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1 \text{ exn}$ .

By (*k-raise-r*) with (d) and (g)

(h)  $\bar{\tau} \triangleright K \diamond (\text{raise } v \text{ with } [\cdot]) \rightsquigarrow \Phi, \tau_1.$

We conclude  $\triangleright (\bar{\tau}, K \diamond (\text{raise } v \text{ with } [\cdot]), e)$  by (*prog*) with (a), (e), (f), and (h).

Case (T11)

Assume  $\triangleright (\bar{\tau}, K \diamond \kappa \diamond (\text{exception } \vec{h} \text{ in } [\cdot]), a)$  From the premises of (*prog*)

(a)  $\Phi \triangleright \bar{\tau} : \bar{\tau},$

(b)  $\bar{\tau} \triangleright K \diamond \kappa \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \rightsquigarrow \Phi, \tau,$  and

(c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright a : \tau, \emptyset.$

By Lemma 5.3 with (b)

(d)  $\bar{\tau} \triangleright K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond \kappa \rightsquigarrow \Phi, \tau.$

We conclude  $\triangleright (\bar{\tau}, K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond \kappa, a)$  by (*prog*) with (a), (c), and (d).

Case (T12)

Assume  $\triangleright (\bar{\tau}, K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond (\text{exception } \vec{h}' \text{ in } [\cdot]), a)$  where  $\vec{h} \equiv h_1, \dots, h_k$  and  $\vec{h}' \equiv h'_1, \dots, h'_k$ . From the premises of (*prog*) and (*k-exndec*)

(a)  $\Phi'' \triangleright \bar{\tau} : \bar{\tau},$

(b)  $\bar{\tau} \triangleright K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \diamond (\text{exception } \vec{h}' \text{ in } [\cdot]) \rightsquigarrow \Phi'', \tau,$

(c)  $\bar{\tau}; \emptyset; \Phi''; \langle \rangle \triangleright a : \tau, \emptyset,$

(d)  $\bar{\tau} \triangleright K \diamond (\text{exception } \vec{h} \text{ in } [\cdot]) \rightsquigarrow \Phi', \tau,$

(e)  $\{\vec{h}'\} \cap \text{dom}(\Phi') = \emptyset,$

$$(f) \text{ fcv}(\tau'_1) \cup \dots \cup \text{ fcv}(\tau'_k) = \emptyset,$$

$$(g) \bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau,$$

$$(h) \{\vec{h}\} \cap \text{ dom}(\Phi) = \emptyset, \text{ and}$$

$$(i) \text{ fcv}(\tau_1) \cup \dots \cup \text{ fcv}(\tau_k) = \emptyset$$

where  $\Phi' = \Phi \langle h_1 : \tau_1 \rangle \dots \langle h_k : \tau_k \rangle$  and  $\Phi'' = \Phi' \langle h'_1 : \tau'_1 \rangle \dots \langle h'_k : \tau'_k \rangle$ .

From (e) and (h) we know the elements of  $\vec{h}$  and  $\vec{h}'$  are all distinct and we have  $\{\vec{h}, \vec{h}'\} \cap \text{ dom}(\Phi) = \emptyset$ . Using this fact with (f), (g), and (i) we have by (*k-exndec*) that

$$(j) \bar{\tau} \triangleright K \diamond (\text{exception } \vec{h}, \vec{h}' \text{ in } [\cdot]) \rightsquigarrow \Phi'', \tau.$$

We conclude  $\triangleright (\bar{s}, K \diamond \text{exception } \vec{h}, \vec{h}' \text{ in } [\cdot], a)$  by (*prog*) with (a), (c), and (j).

Case (*T13*)

Assume  $\triangleright (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e)$  From the premises of (*prog*), (*k-papp*), (*ans*), and (*rec*)

$$(a) \Phi \triangleright \bar{s} : \bar{\tau},$$

$$(b) \bar{\tau} \triangleright K \diamond [\cdot] \langle \bar{\gamma} \rangle \rightsquigarrow \Phi, \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau,$$

$$(c) \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau, \emptyset,$$

$$(d) \bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}],$$

$$(e) \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau, \text{ and}$$

$$(f) \bar{\tau}; \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Phi; \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau \rangle \triangleright \text{fn } x \Rightarrow e : \tau.$$

By Lemma 5.4 with (e) and (f)

(g)  $\bar{\tau}; \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Phi; \langle \rangle \triangleright (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle x \rangle (\bar{\rho}) \Rightarrow e)] : \tau.$

By Lemma 7.1 with (d) we have  $fcv(\tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]) = \emptyset$  which implies

(h)  $fcv(\bar{\gamma}) \cup fcv(\bar{\varphi}) \cup fcv(\bar{\tau}) = \emptyset.$

Also by Lemma 7.1 with (d) we know  $fcv(\Phi) = \emptyset$  and hence  $\Phi[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}] = \Phi.$  Let  $v \equiv (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e)][\bar{\rho} := \bar{\gamma}].$

By Lemma 5.5 with (g) and (h)

(i)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}].$

By (ans) with (i)

(j)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}], \emptyset.$

We conclude  $\triangleright (\bar{s}, K, (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e)][\bar{\rho} := \bar{\gamma}])$  by (prog) with (a), (d), and (j).

Case (T14)

Assume  $\triangleright (\bar{s}, K \diamond ((\text{fn } x \Rightarrow e) [\cdot]), v).$  From the premises of (prog), (k-app-r), (ans), and (abs)

(a)  $\Phi \triangleright \bar{s} : \bar{\tau},$

(b)  $\bar{\tau} \triangleright K \diamond ((\text{fn } x \Rightarrow e) [\cdot]) \rightsquigarrow \Phi, \tau_1,$

(c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1, \emptyset,$

(d)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau_2,$

(e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle,$

- (f)  $\varphi \subseteq \text{dom}(\bar{\tau})$ ,
- (g)  $\bar{\tau}; \emptyset; \Phi; \langle x : \tau_1 \rangle \triangleright e : \tau_2, \varphi'$ ,
- (h)  $\varphi' \subseteq \varphi$ , and
- (i)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1$ .

By Lemma 5.4 with (g) and (i)

- (j)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e[x := v] : \tau, \varphi'$ .

We conclude  $\triangleright (\bar{\tau}, K, e[x := v])$  by (*prog*) with (a), (d), and (j) and using the fact that  $\varphi' \subseteq \text{dom}(\bar{\tau})$  which follows from (f) and (h).

Case (T15)

Assume  $\triangleright (\bar{\tau}, K \diamond (\text{new } \xi [\cdot]), v)$  where  $\bar{\tau}(\xi) = R$  for some  $R$  and choose  $l \notin \text{dom}(R)$ . From the premises of (*prog*), (*k-new*), and (*ans*)

- (a)  $\Phi \triangleright \bar{\tau} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \Phi, \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau, \emptyset$ ,
- (d)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \text{ref } \xi \tau$ ,
- (e)  $\xi \in \text{dom}(\bar{\tau})$ , and
- (f)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau$ .

From the premises of (*stack*) and (*region*) with (a) and our assumption about  $\xi$  we know  $\bar{\tau}(\xi) = \Sigma$  for some  $\Sigma$  and  $l \notin \text{dom}(\Sigma)$ . Let  $\bar{\tau}' = \bar{\tau} \langle \xi \mapsto \Sigma \langle l \mapsto \tau \rangle \rangle$ . By Lemma 5.8 with (d) and (f)

- (g)  $\bar{\tau}' \triangleright K \rightsquigarrow \Phi, \text{ref } \gamma \tau$  and

(h)  $\bar{\zeta}' ; \emptyset ; \Phi ; \langle \rangle \triangleright v : \tau$ .

By Lemma 5.8 with (a) and (h)

(i)  $\Phi \triangleright \bar{s} \langle \xi \mapsto R \langle l \mapsto v \rangle \rangle : \bar{\zeta}'$ .

By Lemma 5.1 with (b) we know  $fcv(\tau) = \emptyset$  so by (*loc-live*)

(j)  $\bar{\zeta}' ; \emptyset ; \Phi ; \langle \rangle \triangleright (\xi, l) : \text{ref } \xi \tau$ .

By (*ans*) with (j)

(k)  $\bar{\zeta}' ; \emptyset ; \Phi ; \langle \rangle \triangleright (\xi, l) : \text{ref } \xi \tau, \emptyset$ .

We conclude  $\triangleright (\bar{s}, K, (\xi, l))$  by (*prog*) with (g), (i), and (k).

Case (T16)

Assume  $\triangleright (\bar{s}, K \diamond (\text{get } [\cdot]), (\xi, l))$  where  $\bar{s}(\xi) = R$  and  $R(l) = v$  for some  $R$  and  $v$ . From the premises of (*prog*), (*k-get*), and (*ans*) and from inspection of the typing rules for locations

(a)  $\Phi \triangleright \bar{s} : \bar{\zeta}$ ,

(b)  $\bar{\zeta} \triangleright K \diamond (\text{get } [\cdot]) \rightsquigarrow \Phi, \text{ref } \xi \tau$ ,

(c)  $\bar{\zeta} ; \emptyset ; \Phi ; \langle \rangle \triangleright (\xi, l) : \text{ref } \xi \tau, \emptyset$ ,

(d)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau$ ,

(e)  $\xi \in \text{dom}(\bar{\zeta})$ , and

(f)  $\bar{\zeta} ; \emptyset ; \Phi ; \langle \rangle \triangleright (\xi, l) : \tau$ .

From (e) we know (f) was derived by (*loc-live*) with premise  $\bar{\zeta}(\xi)(l) = \tau$ .

Using this fact with (a) we have from the premises of (*stack*) and (*region*) that

(g)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau$ .

By *(ans)* with (g)

(h)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau, \emptyset$ .

We conclude  $\triangleright (\bar{\tau}, K, v)$  by *(prog)* with (a), (d), and (h).

Case (T17)

Assume  $\triangleright (\bar{\tau}, K, \text{letreg } \rho \text{ in } e)$  and choose  $\xi$  such that  $\xi \notin \text{frn}(\bar{\tau}) \cup \text{frn}(e)$  and  $\bar{\tau} @ \xi \mapsto \langle \rangle$  exists, in other words,  $\xi \notin \text{dom}(\bar{\tau})$ . From the premises of *(prog)* and *(letreg)*

(a)  $\Phi \triangleright \bar{\tau} : \bar{\tau}$ ,

(b)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau$ ,

(c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}$ ,

(d)  $\varphi \setminus \{\rho\} \subseteq \text{dom}(\bar{\tau})$ ,

(e)  $\bar{\tau}; \{\rho\}; \Phi; \langle \rangle \triangleright e : \tau, \varphi$ , and

(f)  $\rho \notin \text{frv}(\tau)$ .

Let  $\bar{\tau}' = \bar{\tau} @ \xi \mapsto \emptyset$ . By Lemma 5.6 with (a) and (e) and our assumptions about  $\xi$

(g)  $\Phi \triangleright \bar{\tau} @ \xi \mapsto \langle \rangle : \bar{\tau}'$  and

(h)  $\bar{\tau}'; \{\rho\}; \Phi; \langle \rangle \triangleright e : \tau, \varphi$ .

By *(k-pop)* with (b)

(i)  $\bar{\tau}' \triangleright K \diamond (\text{pop}; [\cdot]) \rightsquigarrow \Phi, \tau$ .



By Lemma 5.1 with (b) we know  $\Phi[\rho := \xi] = \Phi$  and from (f) we know  $\tau[\rho := \xi] \equiv \tau$  so by Lemma 5.5 with (e)

$$(j) \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright e[\rho := \xi] : \tau, \varphi[\rho := \xi].$$

From (d) we have

$$(k) \varphi[\rho := \xi] \subseteq \text{dom}(\bar{\tau}').$$

We conclude  $\triangleright (\bar{\tau} @ \xi \mapsto \emptyset, K \diamond (\text{pop}; [\cdot]), e[\rho := \xi])$  by (*prog*) with (g), (i), (j), and (k).

Case (T18)

Assume  $\triangleright (\bar{\tau}, K \diamond (\text{pop}; [\cdot]), a)$ . From the premises of (*prog*) and (*k-pop*) and from inspection of the rule (*stack*)

$$(a) \Phi \triangleright \bar{\tau} @ \xi \mapsto R : \bar{\tau} @ \xi \mapsto \Sigma,$$

$$(b) \bar{\tau} @ \xi \mapsto \Sigma \triangleright K \diamond (\text{pop}; [\cdot]) \rightsquigarrow \Phi, \tau,$$

$$(c) \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright a : \tau, \emptyset, \text{ and}$$

$$(d) \bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau.$$

By Lemma 5.7 with (a) and (c)

$$(e) \Phi \triangleright \bar{\tau} : \bar{\tau} \text{ and}$$

$$(f) \bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright a : \tau, \emptyset.$$

We conclude  $\triangleright (\bar{\tau}, K, a)$  by (*prog*) with (c), (e), and (f).

Case (T19)

Assume  $\triangleright (\bar{\tau}, K \diamond (\text{raise } h \text{ with } [\cdot]), v)$  From the premises of (*prog*), (*k-raise-r*), (*ans*), and (*constr*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K \diamond (\text{raise } h \text{ with } [\cdot]) \rightsquigarrow \Phi, \tau_1$ ,
- (c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1, \emptyset$ ,
- (d)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau_2$ ,
- (e)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright h : \tau_1 \text{ exn}$ ,
- (f)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1$ , and
- (g)  $\Phi(h) = \tau_1$ .

By Lemma 7.1 with (d) we have

- (h)  $fcv(\tau_2) = \emptyset$ .

By (*packet*) with (f), (g), and (h)

- (i)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau_2$ .

By (*ans*) with (i)

- (j)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau_2, \emptyset$ .

We conclude  $\triangleright (\bar{s}, K, \text{fail } h \text{ with } v)$  by (*prog*) with (a), (d), and (j).

Case (T20)

Assume  $\triangleright (\bar{s}, K \diamond \delta, \text{fail } h \text{ with } v)$  From the premises of (*prog*), (*ans*), and (*packet*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K \diamond \delta \rightsquigarrow \Phi, \tau_2$ ,
- (c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau_2, \emptyset$ ,

(d)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau_2,$

(e)  $\Phi(h) = \tau_1,$  and

(f)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_1.$

From (b) by inspection of the typing rules for continuations frames ranged over by  $\delta$  it is easy to see that there exists  $\tau$  such that

(g)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau.$

By Lemma 5.1 with (g)

(h)  $fcv(\tau) = \emptyset.$

By (*packet*) with (e), (f), and (h)

(i)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau.$

By (*ans*) with (i)

(j)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau, \emptyset.$

We conclude  $\triangleright (\bar{s}, K, \text{fail } h \text{ with } v)$  by (*prog*) with (a), (g), and (j).

Case (T21)

Assume  $\triangleright (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e), \text{fail } h \text{ with } v).$  From the premises of (*prog*), (*k-try-b*), (*constr*), (*ans*), and (*packet*)

(a)  $\Phi \triangleright \bar{s} : \bar{\zeta},$

(b)  $\bar{\zeta} \triangleright K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e) \rightsquigarrow \Phi, \tau_1,$

(c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau_1, \emptyset,$

(d)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau_1,$

- (e)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright h : \tau_2 \text{ exn},$
- (f)  $\Phi(h) = \tau_2,$
- (g)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e : \tau_2 \rightarrow \langle \varphi, \tau_1 \rangle,$
- (h)  $\varphi \subseteq \text{dom}(\bar{\zeta}),$
- (i)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h \text{ with } v : \tau_1,$
- (j)  $\Phi(h) = \tau,$  and
- (k)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau.$

From (f) and (j) we know  $\tau \equiv \tau_2$ . By (*ans*) with (g) and (k)

- (l)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fn } x \Rightarrow e : \tau_2 \rightarrow \langle \varphi, \tau_1 \rangle, \emptyset$  and
- (m)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_2, \emptyset.$

By (*app*) with (l) and (m)

- (n)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright (\text{fn } x \Rightarrow e) v : \tau_1, \varphi.$

We conclude  $\triangleright (\bar{s}, K, (\text{fn } x \Rightarrow e) v)$  by (*prog*) with (a), (d), (h), and (n).

Case (T22)

Assume  $\triangleright (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e), \text{fail } h' \text{ with } v)$ . From the premises of (*prog*), (*k-try-b*),

- (a)  $\Phi \triangleright \bar{s} : \bar{\zeta},$
- (b)  $\bar{\zeta} \triangleright K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e) \rightsquigarrow \Phi, \tau,$
- (c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright \text{fail } h' \text{ with } v : \tau, \emptyset,$  and
- (d)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau.$

We conclude  $\triangleright (\bar{s}, K, \text{fail } h' \text{ with } v)$  by *(prog)* with (a), (c), and (d).

Case (T23)

Assume  $\triangleright (\bar{s}, K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e), v)$ . From the premises of *(prog)*, *(k-try-b)*,

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \diamond (\text{try } [\cdot] \text{ handle } h \text{ with } x \Rightarrow e) \rightsquigarrow \Phi, \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau, \emptyset$ , and
- (d)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \tau$ .

We conclude  $\triangleright (\bar{s}, K, v)$  by *(prog)* with (a), (c), and (d).

□

We now turn to the proof of progress. Although tedious, proving progress for  $\mathcal{RE}$  in detail is important as it verifies the mutual consistency of our fairly complex set of syntactic definitions and transition rules.

Proposition 5.11 ( $\mathcal{RE}$  Progress)

If  $\triangleright M$  then either  $M \mapsto M'$  or  $M \equiv (\bar{s}, K, a)$  and either  $K \equiv [\cdot]$  or  $K \equiv [\cdot] \diamond (\text{exception } \bar{h} \text{ in } [\cdot])$ .

PROOF: Assume  $M \equiv (\bar{s}, K, e)$  and  $\triangleright M$ . The proof is by induction on the structure of  $e$ . If  $e$  is not an answer, then it reduces by either (T1), (T2), (T4), (T5), (T6), (T7), (T9), or (T17).

If  $e \equiv a$  then we proceed by induction on the structure of  $K$ :

Case  $([\cdot])$

Then  $M \equiv (\bar{s}, [\cdot], a)$ .

Case ( $K' \diamond (\text{exception } \vec{h} \text{ in } [\cdot])$ )

Now by induction on the structure of  $K'$ :

If  $K' \equiv [\cdot]$  then  $M \equiv (\bar{s}, [\cdot] \diamond (\text{exception } \vec{h} \text{ in } [\cdot]), a)$ . If  $K' \equiv K'' \diamond (\text{exception } \vec{h}' \text{ in } [\cdot])$  then  $M$  reduces by (T12). If  $K' \equiv K'' \diamond \kappa$  then  $M$  reduces by (T11).

Case ( $K' \diamond ([\cdot] \langle \bar{\gamma} \rangle)$ )

If  $e \equiv p$  then  $M$  reduces by (T20). Assume  $e \equiv v$ . From the premises of (*prog*), (*k-papp*), and (*ans*) and from inspection of the typing rule for answers

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K' \diamond ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \Phi, \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau, \emptyset$ ,
- (d)  $\bar{\tau} \triangleright K' \rightsquigarrow \Gamma, \tau[\vec{\rho} := \bar{\gamma}, \vec{\epsilon} := \vec{\varphi}, \vec{\alpha} := \vec{\tau}]$ , and
- (e)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \forall \vec{\rho}, \vec{\epsilon}, \vec{\alpha}. \tau$ .

By Lemma 5.9 with (a) and (e) we know  $v \equiv (\text{rec } f \langle \vec{\rho} \rangle)(x) \Rightarrow e$ . Furthermore, from (d) we know the substitution  $[\vec{\rho} := \bar{\gamma}]$  exists, in other words  $|\vec{\rho}| = |\bar{\gamma}|$ . Therefore  $M$  reduces by (T13).

Case ( $K' \diamond [\cdot] e$ )

If  $e \equiv p$  then  $M$  reduces by (T20). If  $e \equiv v$  then  $M$  reduces by (T3).

Case ( $K' \diamond (v' [\cdot])$ )

If  $e \equiv p$  then  $M$  reduces by (T20). Assume  $e \equiv v$ . From the premises of (*prog*) and (*ans*) and from inspection of the typing rule for answers

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K' \diamond (v' [\cdot]) \rightsquigarrow \Phi, \tau_2$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v' : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle, \emptyset$ , and
- (d)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v' : \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle$ .

By Lemma 5.9 with (a) and (d) we know  $v' \equiv (\text{fn } x \Rightarrow e')$ . Therefore  $M$  reduces by (T14).

Case ( $K' \diamond (\text{new } \gamma [\cdot])$ )

If  $e \equiv p$  then  $M$  reduces by (T20). Assume  $e \equiv v$ . From the premises of (*prog*) and (*k-new*)

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K' \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \Phi, \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau, \emptyset$ ,
- (d)  $\bar{\tau} \triangleright K \rightsquigarrow \Phi, \text{ref } \gamma \tau$ , and
- (e)  $\gamma \in \text{dom}(\bar{\tau})$ .

From (a) and (e) by inspection of the rule (*stack*) we know  $\gamma \equiv \xi$  and  $\xi \in \text{dom}(\bar{s})$  and so  $\bar{s}(\xi) = R$  for some  $R$ . Therefore  $M$  reduces by (T15) by choosing any  $l \notin \text{dom}(R)$ .

Case ( $K' \diamond (\text{get } [\cdot])$ )

If  $e \equiv p$  then  $M$  reduces by (T20). Assume  $e \equiv v$ . From the premises of (*prog*), (*k-get*), and (*ans*) and from inspection of the typing rule for answers

- (a)  $\Phi \triangleright \bar{s} : \bar{\tau}$ ,

(b)  $\bar{\zeta} \triangleright K' \diamond (\text{get } [\cdot]) \rightsquigarrow \Phi, \text{ref } \gamma \tau,$

(c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \text{ref } \gamma \tau, \emptyset,$

(d)  $\bar{\zeta} \triangleright K \rightsquigarrow \Phi, \tau,$

(e)  $\gamma \in \text{dom}(\bar{\zeta}),$  and

(f)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \text{ref } \gamma \tau.$

From (a) and (e) and the premises of (*stack*) we know  $\gamma \equiv \xi$  and  $\xi \in \text{dom}(\bar{\zeta})$ . Using this fact with (a) and (f) we know by Lemma 5.9 that  $v \equiv (\xi, l)$  and  $\bar{\zeta}(\xi)(l) = v'$ . Therefore  $M$  reduces by (T16).

Case ( $K' \diamond (\text{raise } [\cdot] \text{ with } e)$ )

If  $e \equiv p$  then  $M$  reduces by (T20). If  $e \equiv v$  then  $M$  reduces by (T10)

Case ( $K' \diamond (\text{raise } v \text{ with } [\cdot])$ )

If  $e \equiv p$  then  $M$  reduces by (T20). Assume  $e \equiv v$ . From the premises of (*prog*) and (*k-raise-r*)

(a)  $\Phi \triangleright \bar{\zeta} : \bar{\zeta},$

(b)  $\bar{\zeta} \triangleright K' \diamond (\text{raise } v \text{ with } [\cdot]) \rightsquigarrow \Phi, \tau,$  and

(c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau \text{ exn.}$

By Lemma 7.7 with (a) and (c) we know  $v \equiv h$ . Therefore  $M$  reduces by (T19).

Case ( $K' \diamond (\text{pop}; [\cdot])$ )

From the premises of (*prog*)

(a)  $\Phi \triangleright \bar{\zeta} : \bar{\zeta},$  and



(b)  $\bar{\zeta} \triangleright K \diamond (\text{pop}; [\cdot]) \rightsquigarrow \Phi, \tau.$

By inspection of the rule (*k-pop*) with (b) we know  $\bar{\zeta}$  has the form  $\bar{\zeta}' @ \xi \mapsto \Sigma$  and so from the premises of (*stack*) we know  $\bar{s}$  has the form  $\bar{s}' @ \xi \mapsto R.$  Therefore  $M$  reduces by (*T18*).

Case ( $K' \diamond (\text{try} [\cdot] \text{ handle } v \text{ with } x \Rightarrow e')$ )

If  $e \equiv v'$  then  $M$  reduces by (*T23*). Assume  $e \equiv p.$  From the premises of (*prog*) and (*k-try-b*)

(a)  $\Phi \triangleright \bar{s} : \bar{\zeta},$

(b)  $\bar{\zeta} \triangleright K' \diamond (\text{try} [\cdot] \text{ handle } v \text{ with } x \Rightarrow e') \rightsquigarrow \Phi, \tau_1, \text{ and}$

(c)  $\bar{\zeta}; \emptyset; \Phi; \langle \rangle \triangleright v : \tau_2 \text{ expn.}$

By Lemma 5.9 with (a) and (c) we know  $v \equiv h.$  Now if  $p \equiv \text{fail } h \text{ with } v''$  then  $M$  reduces by (*T21*). Otherwise  $p \equiv \text{fail } h' \text{ with } v''$  and  $h \neq h'$  and  $M$  reduces by (*T22*).

□

Finally, having established preservation and progress properties, soundness for  $\mathcal{RE}$  follows easily. Recall that a program,  $M,$  *diverges* if there exists an infinite reduction starting from  $M.$

Proposition 5.12 ( $\mathcal{RE}$  Soundness)

If  $\triangleright M$  then either  $M$  diverges or  $M \mapsto^* (\bar{s}, K, a)$  and  $\triangleright (\bar{s}, K, a)$  and either  $K \equiv [\cdot]$  or  $K \equiv [\cdot] \diamond (\text{exception } \bar{h} \text{ in } [\cdot]).$

PROOF: Suppose  $\triangleright M$  and  $M \mapsto^* M'$  and that there exists no  $M''$  such that  $M' \mapsto M''.$  By induction on the length of the reduction sequence

using Proposition 5.10 we have  $\triangleright M'$ . By Proposition 5.11 we have  $M' \equiv (\bar{s}, K, a)$  and either  $K \equiv [\cdot]$  or  $K \equiv [\cdot] \diamond$  (exception  $\bar{h}$  in  $[\cdot]$ ).

□

## CHAPTER VI

### CONTINUATIONS

Continuations are a heavily studied subject having roots in both semantic foundations and implementation technologies[2, 41]. The ability to capture and later invoke continuations provides a powerful programming paradigm which can be used to implement features such as coroutines and backtracking[15, 20]. This power comes at a certain expense. Clinger *et al.* have compared a number of proposed implementation strategies for first-class continuations[10]. Folklore also suggests that continuations and references together subsume exception handling constructs such as those introduced in CHAPTER IV. For example, Reynolds provides an (untyped) encoding[39].

#### Programming with Continuations

Although not part of Standard ML itself, the Standard ML of New Jersey compiler provides constructs for capturing and invoking continuations[3, 36]. We have already seen that incorporating imperative features into a system based on Milner polymorphism is delicate. Just as mutable references can lead to potential unsoundness, continuations must also be handled with care. The typing discipline for continuations employed by SML/NJ was proposed by Harper *et al.* [19].

A typical use of first-class continuations is to provide an efficient exit from a (non-tail) recursive function. The example shown in FIGURE 23 is adapted from [19]. The function `product` takes a list of integers, captures a continuation representing the return point, and then recursively traverses the list using the internally declared function

---

```

let fun product l =
  callcc(fn exit =>
    let fun loop [] = 1
        | loop (0::t1) = throw exit 0
        | loop (hd::t1) = hd * loop(t1)
    in loop l
    end)
in product [4,5,6,0,7]
end

```

---

FIGURE 23. SMI/NJ Example

loop. If a zero is encountered, the captured continuation is used to return without the overhead of returning from each of the intervening recursive invocations.

### The Problem

From a typing perspective, continuations are slightly different from ordinary functions. Ignoring effects for a moment, functions themselves are normally assigned types of the form  $(\tau_1 \rightarrow \tau_2)$  where  $\tau_1$  represents the type of the expected argument, and  $\tau_2$  is the type returned by the function. In contrast, continuations do not “return” in the sense of ordinary functions. Thus, the typing discipline of SML/NJ assigns continuation types of the form  $(\tau \text{ cont})$ . This type can be understood as an abbreviation for the function type  $(\tau \rightarrow \text{ans})$  where  $\text{ans}$  is the top-level type of the program.

In each of the type and effect systems we have studied, functions are assigned types of the form  $(\tau_1 \rightarrow \langle \varphi, \tau_2 \rangle)$  where  $\varphi$  represents the latent effect incurred by a function application. Intuitively, in order to accommodate continuations in to this framework, we will need to assign them types of the form  $(\text{cont } \varphi \tau)$  where again,  $\varphi$  represents

the effects which may occur after invoking the continuation. The question is now what effect to use.

### Effects vs. Capabilities

We will motivate our solution to typing continuations by contrasting two approaches to effect systems. The systems studied so far are “bottom-up” in nature in the sense that effect information is propagated additively from each sub-expression to enclosing expressions. Crary *et al.* introduced a dual view, based on the notion of capabilities, in which the effects generated by an expression are constrained, in top-down fashion, to occur only in the regions allowed by the current capability.

Although the purpose of casting region systems in terms of capabilities was originally related to typing programs in continuation-passing style, one can easily see the equivalence of these dual views for direct-style programs. We will formalize this connection for a minimal effect system based on the following syntax:

$$e ::= x \mid \text{fn } x \Rightarrow e \mid e_1 e_2 \mid \text{new } \rho e \mid \text{get } e \mid \text{letreg } \rho \text{ in } e$$

This language is a simple subset of  $\mathcal{RL}$  for which we omit type, region, and effect polymorphism

An effect system for this language would typically be defined using judgments of the form  $\Gamma \triangleright e : \tau, \varphi$  where  $\varphi$  is a set of region variables and  $\tau$  is defined by:

$$\tau ::= \alpha \mid \text{ref } \rho \tau \mid \tau_1 \rightarrow \langle \varphi, \tau_2 \rangle$$

In the absence of polymorphism, type variables play the rôle of base types.

An equivalent system, based on the idea of capabilities, can be constructed with judgments of the form  $\Gamma ; \varphi \triangleright e : \tau$ . In this judgment form, the effect  $\varphi$  is instead regarded as a capability which describes all of the allowable regions which  $e$  may use. Types are modified as follows:

$$\tau := \alpha \mid \text{ref } \rho \tau \mid \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$$

In contrast to the latent effects used previously, functions will be assigned types of the form  $\langle \varphi, \tau_1 \rangle \rightarrow \tau_2$  where  $\varphi$  is the capability which must be held in order to call the function.

The typing rules for these systems are defined in FIGURES 24 and 25. For ease of comparison, we will henceforth merge the syntax of function types so that both systems are defined with the neutral notation  $(\tau_1 \xrightarrow{\varphi} \tau_2)$ . The rules for the effect system should be unsurprising. They represent a simplification of the systems studied so far. The capability system can be understood as taking the opposite approach to propagating effect information. For example, the rules (*e-var*) and (*e-abs*) indicate that these values incur no effect. In contrast, the rules (*c-var*) and (*c-abs*) indicate that no evaluation, and hence no capability is required. The rule (*e-app*) adds the effects of evaluating a function, its argument, and the latent effect of applying the function. Instead, the rule (*c-app*) enforces that the current capability held is sufficient to evaluate the function and argument, and that it subsumes the capability required to call the function. For constructs that normally generate new effects according to the rules (*e-new*) and (*e-get*), these effects are checked for legality according the current capability by rules (*c-new*) and (*c-get*). Finally, just as a region variable can be masked from the effect by rule (*e-reg*), the rule (*c-reg*) extends the current capability to include the new region.

---


$$\begin{array}{l}
(e\text{-var}) \quad \frac{\Gamma(x) = \tau}{\Gamma \triangleright x : \tau, \emptyset} \\
(e\text{-abs}) \quad \frac{\Gamma(x : \tau_1) \triangleright e : \tau_2, \varphi \quad x \notin \text{dom}(\Gamma)}{\Gamma \triangleright \text{fn } x \Rightarrow e : \tau_1 \xrightarrow{\varphi} \tau_2, \emptyset} \\
(e\text{-app}) \quad \frac{\Gamma \triangleright e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2, \varphi_2 \quad \Gamma \triangleright e_2 : \tau_1, \varphi_3}{\Gamma \triangleright e_1 e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi_3} \\
(e\text{-new}) \quad \frac{\Gamma \triangleright e : \tau, \varphi}{\Gamma \triangleright \text{new } \rho e : \text{ref } \rho \tau, \varphi \cup \{\rho\}} \\
(e\text{-get}) \quad \frac{\Gamma \triangleright e : \text{ref } \rho \tau, \varphi}{\Gamma \triangleright \text{get } e : \tau, \varphi \cup \{\rho\}} \\
(e\text{-reg}) \quad \frac{\Gamma \triangleright e : \tau, \varphi \quad \rho \notin \text{fru}(\Gamma) \cup \text{fru}(\tau)}{\Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}} \\
(e\text{-sub}) \quad \frac{\Gamma \triangleright e : \tau, \varphi_1 \quad \varphi_1 \subseteq \varphi_2}{\Gamma \triangleright e : \tau, \varphi_2}
\end{array}$$


---

FIGURE 24. Simple Effect-Based Region System

---


$$\begin{array}{l}
(c\text{-var}) \quad \frac{\Gamma(x) = \tau}{\Gamma; \emptyset \triangleright x : \tau} \\
(c\text{-abs}) \quad \frac{\Gamma(x : \tau_1); \varphi \triangleright e : \tau_2 \quad x \notin \text{dom}(\Gamma)}{\Gamma; \emptyset \triangleright \text{fn } x \Rightarrow e : \tau_1 \xrightarrow{\varphi} \tau_2} \\
(c\text{-app}) \quad \frac{\Gamma; \varphi_1 \triangleright e_1 : \tau_1 \xrightarrow{\varphi_2} \tau_2 \quad \Gamma; \varphi_1 \triangleright e_2 : \tau_1 \quad \varphi_2 \subseteq \varphi_1}{\Gamma; \varphi_1 \triangleright e_1 e_2 : \tau_2} \\
(c\text{-new}) \quad \frac{\Gamma; \varphi \triangleright e : \tau \quad \rho \in \varphi}{\Gamma; \varphi \triangleright \text{new } \rho e : \text{ref } \rho \tau} \\
(c\text{-get}) \quad \frac{\Gamma; \varphi \triangleright e : \text{ref } \rho \tau \quad \rho \in \varphi}{\Gamma; \varphi \triangleright \text{get } e : \tau} \\
(c\text{-reg}) \quad \frac{\Gamma; \varphi \uplus \{\rho\} \triangleright e : \tau \quad \rho \notin \text{fru}(\Gamma) \cup \text{fru}(\tau)}{\Gamma; \varphi \triangleright \text{letreg } \rho \text{ in } e : \tau} \\
(c\text{-sub}) \quad \frac{\Gamma; \varphi_1 \triangleright e : \tau \quad \varphi_1 \subseteq \varphi_2}{\Gamma; \varphi_2 \triangleright e : \tau}
\end{array}$$


---

FIGURE 25. Simple Capability-Based Region System



The soundness of the capability system defined in FIGURE 25 is suggested simply by relating it to the well-known effect system of FIGURE 24:

Proposition 6.1 (Equivalence)

$\Gamma \triangleright e : \tau, \varphi$  if and only if  $\Gamma; \varphi \triangleright e : \tau$ .

PROOF: Each direction is established by induction on the height of the given derivation with case analysis on the last rule used. Cases labeled by effect rules apply to the “only if” direction and those labeled by capability rules apply to the “if” direction. The cases for (*e-var*) and (*c-var*) are immediate. The cases for (*e-abs*), (*e-sub*), (*c-abs*), and (*c-sub*) follow directly from the IH. The remaining cases are given below with the exceptions (*e-get*) and (*c-get*) which are similar to (*e-new*) and (*c-new*) respectively:

Case (*e-app*)

Assume  $\Gamma \triangleright e_1 e_2 : \tau_2, \varphi_1 \cup \varphi_2 \cup \varphi_3$ . From the premises of (*e-app*)

- (a)  $\Gamma \triangleright e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2, \varphi_2$ , and
- (b)  $\Gamma \triangleright e_2 : \tau_1, \varphi_3$ .

By the IH with (a) and (b)

- (c)  $\Gamma; \varphi_2 \triangleright e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2$ , and
- (d)  $\Gamma; \varphi_3 \triangleright e_2 : \tau_1$ .

By (*c-sub*) with (c) and (d)

- (e)  $\Gamma; \varphi_1 \cup \varphi_2 \cup \varphi_3 \triangleright e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2$ , and
- (f)  $\Gamma; \varphi_1 \cup \varphi_2 \cup \varphi_3 \triangleright e_2 : \tau_1$ .

Now since  $\varphi_1 \subseteq \varphi_1 \cup \varphi_2 \cup \varphi_3$  we conclude  $\Gamma; \varphi_1 \cup \varphi_2 \cup \varphi_3 \triangleright e_1 e_2 : \tau$  by *(c-app)* with (e) and (f).

Case *(e-new)*

Assume  $\Gamma \triangleright \text{new } \rho e : \text{ref } \rho \tau, \varphi \cup \{\rho\}$ . From the premise of *(e-new)*

(a)  $\Gamma \triangleright e : \tau, \varphi$ .

By the IH with (a)

(b)  $\Gamma; \varphi \triangleright e : \tau$ .

By *(c-sub)* with (b)

(c)  $\Gamma; \varphi \cup \{\rho\} \triangleright e : \tau$ .

Now since  $\rho \in \varphi \cup \{\rho\}$  we conclude  $\Gamma; \varphi \cup \{\rho\} \triangleright \text{new } \rho e : \text{ref } \rho \tau$  by *(c-new)* with (c).

Case *(e-reg)*

Assume  $\Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi \setminus \{\rho\}$ . From the premises of *(e-reg)*

(a)  $\Gamma \triangleright e : \tau, \varphi$  and

(b)  $\rho \notin \text{frv}(\Gamma) \cup \text{frv}(\tau)$ .

By the IH with (a)

(c)  $\Gamma; \varphi \triangleright e : \tau$ .

Now if  $\rho \in \varphi$  then  $\varphi = \varphi' \uplus \{\rho\}$  and  $\varphi' = \varphi \setminus \{\rho\}$  so we can conclude  $\Gamma; \varphi' \triangleright \text{letreg } \rho \text{ in } e : \tau$  by *(c-reg)* with (b) and (c). Otherwise, by *(c-sub)* with (c) we have

(d)  $\Gamma; \varphi \uplus \{\rho\} \triangleright e : \tau$

and the conclusion follows from (b) and (d).

Case (*c-app*)

Assume  $\Gamma; \varphi_1 \triangleright e_1 e_2 : \tau_2$ . From the premises of (*c-app*)

(a)  $\Gamma; \varphi_1 \triangleright e_1 : \tau_1 \xrightarrow{\varphi_2} \tau_2$ ,

(b)  $\Gamma; \varphi_1 \triangleright e_2 : \tau_1$ , and

(c)  $\varphi_2 \subseteq \varphi_1$ .

By the IH with (a) and (b)

(e)  $\Gamma \triangleright e_1 : \tau_1 \xrightarrow{\varphi_2} \tau_2, \varphi_1$  and

(f)  $\Gamma \triangleright e_2 : \tau_1, \varphi_1$ .

Now from (c) we know  $\varphi_2 \cup \varphi_1 \cup \varphi_1 = \varphi_1$  so we conclude  $\Gamma \triangleright e_1 e_2 : \tau_2, \varphi_1$  by (*e-app*) with (e) and (f).

Case (*c-new*)

Assume  $\Gamma; \varphi \triangleright \text{new } \rho e : \text{ref } \rho \tau$ . From the premises of (*c-new*)

(a)  $\Gamma; \varphi \triangleright e : \tau$  and

(b)  $\rho \in \varphi$ .

By the IH with (a)

(c)  $\Gamma \triangleright e : \tau, \varphi$ .

Now from (b) we know  $\varphi \cup \{\rho\} = \varphi$  so we conclude  $\Gamma \triangleright \text{new } \rho e : \text{ref } \rho \tau, \varphi$  by (*e-new*) with (c).

Case (*c-reg*)

Assume  $\Gamma; \varphi \triangleright \text{letreg } \rho \text{ in } e : \tau$ . From the premises of (*c-reg*)

(a)  $\Gamma; \varphi \uplus \{\rho\} \triangleright e : \tau$  and

(b)  $\rho \notin \text{frv}(\Gamma) \cup \text{frv}(\tau)$ .

By the IH with (a)

(c)  $\Gamma \triangleright e : \tau, \varphi \uplus \{\rho\}$ .

Now since  $(\varphi \uplus \{\rho\}) \setminus \{\rho\} = \varphi$  we conclude  $\Gamma \triangleright \text{letreg } \rho \text{ in } e : \tau, \varphi$  by (*e-reg*) with (b) and (c).

□

### Typing Continuations

As we saw in the example in FIGURE 23, invoking a continuation amounts to “jumping” to some program point which was dynamically captured. This behavior poses problems for the region-based execution model. When a continuation is captured, some number of regions will be present in memory. When execution is restarted at that point, these region may not only be involved in allocations and dereferences, but they will subsequently be freed by an appropriate number of stack popping operations. Therefore, just as we saw with exceptions, invoking a continuation should operationally involve popping the stack down so that it contains only the regions present at the time of capture. Of course, this is only possible if these regions are still present!

Our operational intuition about continuations suggests that the effect description assigned in types of the form  $(\text{cont } \varphi \tau)$  should reflect a static approximation of the set of regions present on the stack. Our solution is motivated simply by the fact that

with minor modifications, capabilities can provide exactly this approximation. In order to make this work, consider a modified system obtained by removing the rule (*c-sub*) from FIGURE 25. Thus, the current capability carried through a derivation can only be modified by adding a region via the rule (*c-reg*). This modification is not quite sufficient, however, because function applications still involve subsumption. Therefore we replace the rule (*c-app*) with

$$\frac{\Gamma; \varphi \triangleright e_1 : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2 \quad \Gamma; \varphi \triangleright e_2 : \tau_1}{\Gamma; \varphi \triangleright e_1 e_2 : \tau_2}$$

Now, the capability held will always accurately reflect the regions currently in memory.

Consider extending the capability language with the following constructs for capturing and invoking continuations:

$$e ::= \dots \mid \text{callcc } e \mid \text{throw } e_1 e_2$$

These constructs will be typed by the following (simplified) rules:

$$\frac{\Gamma; \varphi \triangleright e : \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau}{\Gamma; \varphi \triangleright \text{callcc } e : \tau}$$

$$\frac{\Gamma; \varphi_1 \triangleright e_1 : \text{cont } \varphi_2 \tau_1 \quad \Gamma; \varphi_1 \triangleright e_2 : \tau_1 \quad \varphi_1 \subseteq \varphi_2}{\Gamma; \varphi_1 \triangleright \text{throw } e_1 e_2 : \tau_2}$$

These rules reflect the basic character of ML-style continuations. For example, the type for throw-expressions is unconstrained so that the same continuation may be invoked from different contexts. More importantly, the proviso  $\varphi_1 \subseteq \varphi_2$  insures that the regions present at the time of capture are a subset of those still present. Alarming, this is exactly the flexibility we just removed from our ordinary function applications. However,

the operational behavior of a continuation is different; recall that we expect the stack to be popped down to match  $\varphi_2$  as an implicit part of invoking the continuation.

At first sight, our modification of the rule (*c-app*) above may appear overly restrictive; each function type allows for function applications only in a specific region environment. However, our analysis has actually revealed an implicit form of polymorphism which was inherent in the effect and capability systems studied so far. In other words, functions were always assumed to be polymorphic with respect to the region environments they required. This assumption was harmless in the absence of first-class continuations. In order for functions to recover this ability, they must be typed in a principled way which accounts for multiple region environments. Polymorphism in capabilities provides exactly this mechanism.

### The Language $\mathcal{RC}$

The language  $\mathcal{RC}$  will extend  $\mathcal{RL}$  with ML-like continuation primitives. We have already seen utility of explicit continuations in reasoning about exceptions in CHAPTER IV. Not surprisingly, this advantage will be especially evident in reasoning about the reification of continuations. Region identifiers are defined as usual:

$$\gamma ::= \rho \mid \xi$$

Values are extended to include a dynamic representation of captured continuations:

$$v ::= x \mid \underline{n} \mid \text{fn } x \Rightarrow e \mid \text{rec } f \langle \vec{\rho} \rangle (x) \Rightarrow e \mid (\xi, l) \mid (\varphi, \underline{n}, K)$$

The value  $(\varphi, \underline{n}, K)$  contains information relevant to both the dynamic and static se-

antics, as we shall see. Expressions are extended with the constructs discussed in the previous section:

$$e ::= v \mid e \langle \bar{\gamma} \rangle \mid e_1 e_2 \mid \text{new } \gamma e \mid \text{get } e \mid \text{letreg } \rho \text{ in } e \mid \\ \text{callcc } e \mid \text{throw } e_1 e_2$$

Our previous definition of continuations requires only addition frames for the evaluation of sub-expressions occurring in our new constructs:

$$K ::= [\cdot] \mid K \diamond (\{\cdot\} \langle \bar{\gamma} \rangle) \mid K \diamond ([\cdot] e) \mid K \diamond (v [\cdot]) \mid \\ K \diamond (\text{new } \gamma [\cdot]) \mid K \diamond (\text{get } [\cdot]) \mid K \diamond (\text{pop}; [\cdot]) \mid \\ K \diamond (\text{callcc } [\cdot]) \mid K \diamond (\text{throw } [\cdot] e) \mid K \diamond (\text{throw } v [\cdot])$$

Finally, types are extended as follows:

$$\tau ::= \text{int} \mid \alpha \mid \text{ref } \rho \tau \mid \text{cont } \varphi \tau \mid \langle \varphi, \tau_1 \rangle \rightarrow \tau_2 \mid \forall \bar{\rho}, \bar{c}, \bar{\alpha}. \tau$$

We will continue to use  $\varphi$  to represent capabilities and we now consider  $\epsilon$  as ranging over capability variables. This choice of vocabulary is irrelevant, however, to our technical development and all our usual conventions related to types and substitutions carry over directly.

### Dynamic Semantics

The transition rules for  $\mathcal{RC}$  are straightforward. When a continuation is invoked, the region stack must be popped down to provide only the regions present at the time of its capture. We define the operation  $(\bar{s} \downarrow n)$  to produce the prefix for the sequence  $\bar{s}$  of

---

[Decomposition]

- (T1)  $(\bar{s}, K, e \langle \bar{\gamma} \rangle) \mapsto (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), e)$
- (T2)  $(\bar{s}, K, e_1 e_2) \mapsto (\bar{s}, K \diamond ([\cdot] e_2), e_1)$
- (T3)  $(\bar{s}, K \diamond ([\cdot] e), v) \mapsto (\bar{s}, K \diamond (v [\cdot]), e)$
- (T4)  $(\bar{s}, K, \text{new } \gamma e) \mapsto (\bar{s}, K \diamond (\text{new } \gamma [\cdot]), e)$
- (T5)  $(\bar{s}, K, \text{get } e) \mapsto (\bar{s}, K \diamond (\text{get } [\cdot]), e)$
- (T6)  $(\bar{s}, K, \text{callcc } e) \mapsto (\bar{s}, K \diamond (\text{callcc } [\cdot]), e)$
- (T7)  $(\bar{s}, K, \text{throw } e_1 e_2) \mapsto (\bar{s}, K \diamond (\text{throw } [\cdot] e_2), e_1)$
- (T8)  $(\bar{s}, K \diamond (\text{throw } [\cdot] e), v) \mapsto (\bar{s}, K \diamond (\text{throw } v [\cdot]), e)$
- 

FIGURE 26. Transition Rules for  $\mathcal{RC}$  (Part 1)

length  $n$  if it exists. In other words, we define

$$(\bar{s}_1 @ \bar{s}_2 \downarrow n) = \bar{s}_1 \text{ if and only if } |\bar{s}_1| = n$$

The rules defined in FIGURES 26 and 27 are grouped as usual. The computation and region management rules are identical to those for  $\mathcal{RL}$ . The decomposition rules have been extended appropriately for our new phrases. The rules for continuations are more interesting. For technical reasons, captured continuations are tagged with an effect describing the current regions in memory. This type information has no run-time significance, but is needed in order to constrain name generation for reasons similar to those discussed in CHAPTER III. Continuations are also tagged with a numeral reflecting the



---

**[Computation]**

- (T9) 
$$(\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e) \mapsto (\bar{s}, K, (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e)][\bar{\rho} := \bar{\gamma}])$$
- (T10) 
$$(\bar{s}, K \diamond ((\text{fn } x \Rightarrow e) [\cdot]), v) \mapsto (\bar{s}, K, e[x := v])$$
- (T11) 
$$(\bar{s}, K \diamond (\text{new } \xi [\cdot]), v) \mapsto (\bar{s}(\xi \mapsto R \langle l \mapsto v \rangle), K, (\xi, l))$$
  
if  $\bar{s}(\xi) = R$  and  $l \notin \text{dom}(R)$
- (T12) 
$$(\bar{s}, K \diamond (\text{get } [\cdot]), (\xi, l)) \mapsto (\bar{s}, K, v)$$
  
if  $\bar{s}(\xi)(l) = v$

**[Region Management]**

- (T13) 
$$(\bar{s}, K, \text{letreg } \rho \text{ in } e) \mapsto (\bar{s} @ \xi \mapsto \langle \rangle, K \diamond (\text{pop}; [\cdot]), e[\rho := \xi])$$
  
if  $\xi \notin \text{frn}(\bar{s}) \cup \text{frn}(e)$
- (T14) 
$$(\bar{s} @ \xi \mapsto R, K \diamond (\text{pop}; [\cdot]), v) \mapsto (\bar{s}, K, v)$$

**[Continuations]**

- (T15) 
$$(\bar{s}, K \diamond (\text{calcc } [\cdot]), v) \mapsto (\bar{s}, K, v(\varphi, \underline{n}, K'))$$
  
where  $\varphi = \text{dom}(\bar{s})$  and  $n = |\bar{s}|$
- (T16) 
$$(\bar{s}, K \diamond (\text{throw } (\varphi, \underline{n}, K') [\cdot]), v) \mapsto (\bar{s} \downarrow n, K', v)$$
- 

**FIGURE 27. Transition Rules for  $\mathcal{RC}$  (Part 2)**

current size of stack as shown in (T15). The size of the stack is the only addition run-time information required in (T16) which supplies the appropriate prefix of the current stack to the reinstated continuation. Note that if an insufficient number of regions were present, then rule (T16) could not be applied.

### Static Semantics

The static semantics for  $\mathcal{RC}$ , defined in FIGURES 28, 29, 30, and 31, use judgment forms based on capabilities, but are otherwise similar to their counterparts from CHAPTER III. The most important modification to our original rules is that function applications, given by (*app*) in FIGURE 29 are constrained in the manner previously discussed. Furthermore, functional abstractions, typed by rule (*abs*) in FIGURE 28, no longer involve subsumption of latent effects.

Continuations referring to dead regions may appear in dead code in the same manner as we discussed for dead locations in CHAPTER III. Therefore, the typing rules for values given in FIGURE 28 include rules for both live and dead continuations.

---

(var)	$\frac{\Gamma(x) = \tau}{\bar{\zeta}; \Delta; \Gamma \triangleright x : \tau}$
(num)	$\bar{\zeta}; \Delta; \Gamma \triangleright \underline{n} : \text{int}$
(abs)	$\frac{\bar{\zeta}; \Delta; \Gamma \langle x : \tau_1 \rangle; \varphi \triangleright e : \tau_2 \quad fcv(\varphi) \cup fcv(\tau_1) \subseteq \Delta \quad x \notin \text{dom}(\Gamma)}{\bar{\zeta}; \Delta; \Gamma \triangleright \text{fn } x \Rightarrow e : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2}$
(rec)	$\frac{\bar{\zeta}; \Delta \uplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Gamma \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau \rangle \triangleright \text{fn } x \Rightarrow e : \tau \quad fcv(\forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau) \subseteq \Delta \quad f \notin \text{dom}(\Gamma)}{\bar{\zeta}; \Delta; \Gamma \triangleright \text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau}$
(loc-live)	$\frac{\bar{\zeta}(\xi)(l) = \tau \quad fcv(\tau) = \emptyset}{\bar{\zeta}; \Delta; \Gamma \triangleright (\xi, l) : \text{ref } \xi \tau}$
(loc-dead)	$\frac{\xi \notin \text{dom}(\bar{\zeta}) \quad fcv(\tau) = \emptyset}{\bar{\zeta}; \Delta; \Gamma \triangleright (\xi, l) : \text{ref } \xi \tau}$
(cont-live)	$\frac{\bar{\zeta} \downarrow n \triangleright K \rightsquigarrow \tau \quad \varphi = \text{dom}(\bar{\zeta} \downarrow n)}{\bar{\zeta}; \Delta; \Gamma \triangleright (\varphi, \underline{n}, K) : \text{cont } \varphi \tau}$
(cont-dead)	$\frac{\bar{\zeta}' \triangleright K \rightsquigarrow \tau \quad \varphi = \text{dom}(\bar{\zeta}') \quad \varphi \not\subseteq \text{dom}(\bar{\zeta})}{\bar{\zeta}; \Delta; \Gamma \triangleright (\varphi, \underline{n}, K) : \text{cont } \varphi \tau}$

---

FIGURE 28. Static Semantics for  $\mathcal{RC}$  (Part 1)

---

(val)	$\frac{\bar{\varsigma}; \Delta; \Gamma \triangleright v : \tau}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright v : \tau}$
(papp)	$\frac{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau \quad fcv(\bar{\gamma}) \cup fcv(\bar{\varphi}) \cup fcv(\bar{\tau}) \subseteq \Delta}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e \langle \bar{\gamma} \rangle : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]}$
(app)	$\frac{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e_1 : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2 \quad \bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e_2 : \tau_1}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e_1 e_2 : \tau_2}$
(new)	$\frac{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e : \tau \quad \gamma \in \varphi}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright \text{new } \gamma e : \text{ref } \gamma \tau}$
(get)	$\frac{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e : \text{ref } \gamma \tau \quad \gamma \in \varphi}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e : \tau}$
(letreg)	$\frac{\bar{\varsigma}; \Delta \uplus \{\rho\}; \Gamma; \varphi \uplus \{\rho\} \triangleright e : \tau \quad \rho \notin \text{frv}(\tau)}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright \text{letreg } \rho \text{ in } e : \tau}$
(callcc)	$\frac{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright e : \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau}{\bar{\varsigma}; \Delta; \Gamma; \varphi \triangleright \text{callcc } e : \tau}$
(throw)	$\frac{\bar{\varsigma}; \Delta; \Gamma; \varphi_1 \triangleright e_1 : \text{cont } \varphi_2 \tau_1 \quad \bar{\varsigma}; \Delta; \Gamma; \varphi_1 \triangleright e_2 : \tau_1 \quad \varphi_2 \subseteq \varphi_1 \quad fcv(\tau_2) \subseteq \Delta}{\bar{\varsigma}; \Delta; \Gamma; \varphi_1 \triangleright \text{throw } e_1 e_2 : \tau_2}$

---

FIGURE 29. Static Semantics for  $\mathcal{RC}$  (Part 2)

---

(k-empty)	$\frac{\text{frn}(\tau) \subseteq \text{dom}(\bar{\zeta}) \quad \text{fcv}(\tau) = \emptyset}{\bar{\zeta} \triangleright [\cdot] \rightsquigarrow \tau}$
(k-papp)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau [\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]}{\bar{\zeta} \triangleright K \diamond ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau}$
(k-app-l)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau_2 \quad \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright e : \tau_1 \quad \varphi = \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \diamond ([\cdot] e) \rightsquigarrow \langle \varphi, \tau_1 \rangle \rightarrow \tau_2}$
(k-app-r)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau_2 \quad \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2 \quad \varphi = \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \diamond (v [\cdot]) \rightsquigarrow \tau_1}$
(k-new)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \text{ref } \gamma \tau \quad \gamma \in \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \tau}$
(k-get)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau \quad \gamma \in \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \diamond (\text{get } [\cdot]) \rightsquigarrow \text{ref } \gamma \tau}$
(k-pop)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau}{\bar{\zeta} @ \xi \mapsto \Sigma \triangleright K \diamond (\text{pop}; [\cdot]) \rightsquigarrow \tau}$
(k-callcc)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau \quad \varphi = \text{dom}(\bar{\zeta})}{\bar{\zeta} \triangleright K \diamond (\text{callcc } [\cdot]) \rightsquigarrow \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau}$
(k-throw-l)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau_2 \quad \bar{\zeta}; \emptyset; \langle \rangle; \varphi_1 \triangleright e : \tau_1 \quad \varphi_1 = \text{dom}(\bar{\zeta}) \quad \varphi_2 \subseteq \varphi_1}{\bar{\zeta} \triangleright K \diamond (\text{throw } [\cdot] e) \rightsquigarrow \text{cont } \varphi_2 \tau_1}$
(k-throw-r)	$\frac{\bar{\zeta} \triangleright K \rightsquigarrow \tau_2 \quad \bar{\zeta}; \emptyset; \langle \rangle; \varphi_1 \triangleright v : \text{cont } \varphi_2 \tau_1 \quad \varphi_1 = \text{dom}(\bar{\zeta}) \quad \varphi_2 \subseteq \varphi_1}{\bar{\zeta} \triangleright K \diamond (\text{throw } v [\cdot]) \rightsquigarrow \tau_1}$

---

FIGURE 30. Static Semantics for  $\mathcal{RC}$  (Part 3)

---


$$\begin{array}{l}
 \text{(region)} \quad \frac{\overline{\tau}; \emptyset; \langle \rangle \triangleright R(l) : \Sigma(l) \quad \text{dom}(R) = \text{dom}(\Sigma) \quad (\forall l \in \text{dom}(R))}{\overline{\tau} \triangleright R : \Sigma} \\
 \\
 \text{(stack)} \quad \frac{\begin{array}{l} \overline{\tau} = \xi_1 \mapsto R_1, \dots, \xi_k \mapsto R_k \\ \overline{\tau} = \xi_1 \mapsto \Sigma_1, \dots, \xi_k \mapsto \Sigma_k \\ \overline{\tau} \triangleright \overline{\tau}(\xi) : \tau(\xi) \quad (\forall \xi \in \text{dom}(\overline{\tau})) \end{array}}{\triangleright \overline{\tau} : \overline{\tau}} \\
 \\
 \text{(prog)} \quad \frac{\begin{array}{l} \triangleright \overline{\tau} : \overline{\tau} \quad \overline{\tau} \triangleright K \rightsquigarrow \tau \\ \overline{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e : \tau \quad \varphi = \text{dom}(\overline{\tau}) \end{array}}{\triangleright (\overline{\tau}, K, e)}
 \end{array}$$


---

FIGURE 31. Static Semantics for  $\mathcal{RC}$  (Part 4)

## CHAPTER VII

SOUNDNESS OF  $\mathcal{RC}$ 

The soundness proof for  $\mathcal{RC}$  follows along similar lines as that of  $\mathcal{RE}$  given in CHAPTER V, although the relative semantic cleanliness of continuations leads to fewer cases. After stating the required properties, we provide detailed proofs of preservation and progress.

Basic Properties

Each of the Lemmas in this section has a counterpart in CHAPTER V which has been adapted to the new judgment forms. Here, we merely state the revised properties and refer the reader to CHAPTER V for sketches of their proofs and comments on their rôle in the proof of preservation.

Lemma 7.1 ( $\mathcal{RC}$  Proper Typing)

Suppose  $fcv(\Gamma) \cup fcv(\varphi) \subseteq \Delta$ .

1. If  $\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau$  then  $fcv(\tau) \subseteq \Delta$ .
2. If  $\bar{\zeta}; \Delta; \Gamma; \varphi \triangleright e : \tau$  then  $fcv(\tau) \subseteq \Delta$ .
3. If  $\bar{\zeta} \triangleright K \rightsquigarrow \tau$  then  $fcv(\tau) = \emptyset$ .

Lemma 7.2 ( $\mathcal{RC}$  Value Substitution)

Suppose  $\bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \tau$ .

1. If  $\bar{\zeta}; \Delta; \Gamma \langle x \mapsto \tau \rangle \triangleright v' : \tau'$  then  $\bar{\zeta}; \Delta; \Gamma \triangleright v'[x := v] : \tau'$ .

2. If  $\bar{\zeta}; \Delta; \Gamma \langle x \mapsto \tau \rangle; \varphi \triangleright e : \tau'$  then  $\bar{\zeta}; \Delta; \Gamma; \varphi \triangleright e[x := v] : \tau'$ .

**Lemma 7.3** (*RC Constructor Substitution*)

Suppose  $fcv(\bar{\gamma}) \cup fcv(\bar{\varphi}) \cup fcv(\bar{\tau}) \subseteq \Delta$  and let

(a1)  $\Gamma' = \Gamma[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,

(a2)  $e' = e[\bar{\rho} := \bar{\gamma}]$ ,

(a3)  $v' = v[\bar{\rho} := \bar{\gamma}]$ ,

(a4)  $\varphi' = \varphi[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}]$ , and

(a5)  $\tau' = \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ .

1. If  $\bar{\zeta}; \Delta \uplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Gamma \triangleright v : \tau$  then  $\bar{\zeta}; \Delta; \Gamma' \triangleright v' : \tau'$ .

2. If  $\bar{\zeta}; \Delta \uplus \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \Gamma; \varphi \triangleright e : \tau$  then  $\bar{\zeta}; \Delta; \Gamma'; \varphi' \triangleright e' : \tau'$ .

**Lemma 7.4** (*RC Region Allocation*)

Let  $\bar{\zeta}' = \bar{\zeta} @ \xi \mapsto \langle \rangle$

1. If  $\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau$  and  $\xi \notin frn(v)$  then  $\bar{\zeta}'; \Delta; \Gamma; \varphi \triangleright v : \tau$ .

2. If  $\bar{\zeta}; \Delta; \Gamma; \varphi \triangleright e : \tau$  and  $\xi \notin frn(e)$  then  $\bar{\zeta}'; \Delta; \Gamma; \varphi \triangleright e : \tau$ .

3. If  $\triangleright \bar{s} : \bar{\zeta}$  and  $\xi \notin frn(\bar{s})$  then  $\triangleright \bar{s} @ \xi \mapsto \langle \rangle : \bar{\zeta}'$ .

**Lemma 7.5** (*RC Region Collection*)

Let  $\bar{\zeta} = \bar{\zeta}' @ \xi \mapsto \Sigma$ .

1. If  $\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau$  then  $\bar{\zeta}'; \Delta; \Gamma \triangleright v : \tau$ .



2. If  $\bar{\zeta}; \Delta; \Gamma; \varphi \triangleright e : \tau$  then  $\bar{\zeta}'; \Delta; \Gamma; \varphi \triangleright e : \tau$ .
3. If  $\triangleright \bar{s} @ \xi \mapsto R : \bar{\zeta}$  then  $\triangleright \bar{s} : \bar{\zeta}'$ .

**Lemma 7.6** (*RC Reference Allocation*)

Suppose  $\bar{\zeta}(\xi) = \Sigma$  and  $l \notin \text{dom}(\Sigma)$  and let  $\bar{\zeta}' = \bar{\zeta}(\xi \mapsto \Sigma(l \mapsto \tau))$ .

1. If  $\bar{\zeta}; \Delta; \Gamma \triangleright v : \tau$  then  $\bar{\zeta}'; \Delta; \Gamma \triangleright v : \tau$ .
2. If  $\bar{\zeta}; \Delta; \Gamma; \varphi \triangleright e : \tau$  then  $\bar{\zeta}'; \Delta; \Gamma; \varphi \triangleright e : \tau$ .
3. If  $\bar{\zeta} \triangleright K \rightsquigarrow \tau$  then  $\bar{\zeta}' \triangleright K \rightsquigarrow \tau$ .
4. If  $\triangleright \bar{s} : \bar{\zeta}$  and  $\bar{\zeta}'; \emptyset; \langle \rangle \triangleright v : \tau$  then  $\triangleright \bar{s}(\xi \mapsto \bar{s}(\xi)(l \mapsto v)) : \bar{\zeta}'$ .

**Lemma 7.7** (*RC Canonical Forms*)

Suppose  $\triangleright \bar{s} : \bar{\zeta}$  and  $\bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \tau$ .

1. If  $\tau \equiv \text{ref } \xi \tau'$  then  $v \equiv (\xi, l)$  and either  $\xi \notin \text{dom}(\bar{s})$  or  $\bar{s}(\xi)(l) = v$ .
2. If  $\tau \equiv \text{cont } \varphi \tau'$  then  $v \equiv (\varphi, \underline{n}, K)$  and either  $\varphi \not\subseteq \text{dom}(\bar{s})$  or  $n \leq |\bar{s}|$ .
3. If  $\tau \equiv \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$  then  $v \equiv \text{fn } x \Rightarrow e$ .
4. If  $\tau \equiv \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau'$  then  $v \equiv \text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e$ .

**Preservation and Progress**

**Proposition 7.8** (*RC Typability Preservation*)

If  $\triangleright M$  and  $M \mapsto M'$  then  $\triangleright M'$

PROOF: By case analysis on the transition rules.

Case (T1)

Assume  $\triangleright (\bar{s}, K, e \langle \bar{\gamma} \rangle)$ . From the premises of (*prog*) and (*papp*)

- (a)  $\triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \rightsquigarrow \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,
- (c)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e \langle \bar{\gamma} \rangle : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,
- (d)  $\varphi = \text{dom}(\bar{\tau})$ ,
- (e)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ , and

By (*k-papp*) with (b)

- (f)  $\bar{\tau} \triangleright K \diamond ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ .

We conclude  $\triangleright (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), e)$  by (*prog*) with (a), (d), (e), and (f).

Case (T2)

Assume  $\triangleright (\bar{s}, K, e_1 e_2)$ . From the premises of (*prog*) and (*app*)

- (a)  $\triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \rightsquigarrow \tau_2$ ,
- (c)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e_1 e_2 : \tau_2$ ,
- (d)  $\varphi = \text{dom}(\bar{s})$ ,
- (e)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e_1 : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$ , and
- (f)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e_2 : \tau_1$ .

By (*k-app-l*) with (b), (d), and (f)

- (g)  $\bar{\tau} \triangleright K \diamond ([\cdot] e_2) \rightsquigarrow \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$ .

We conclude  $\triangleright (\bar{s}, K \diamond ([\cdot] e_2), e_1)$  by (*prog*) with (a), (d), (e), and (g).

Case (T3)

Assume  $\triangleright (\bar{s}, K \diamond ([\cdot] e), v)$ . From the premises of (*prog*) and (*k-app-l*)

- (a)  $\triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K \diamond ([\cdot] e) \rightsquigarrow \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$ ,
- (c)  $\bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$ ,
- (d)  $\varphi = \text{dom}(\bar{s})$ ,
- (e)  $\bar{\zeta} \triangleright K \rightsquigarrow \tau_2$ , and
- (f)  $\bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright e : \tau_1$ .

By (*k-app-r*) with (c), (d), and (e)

- (g)  $\bar{\zeta} \triangleright K \diamond (v [\cdot]) \rightsquigarrow \tau_1$ .

We conclude  $\triangleright (\bar{s}, K \diamond (v [\cdot]), e)$  by (*prog*) with (a), (d), (f), and (g).

Case (T4)

Assume  $\triangleright (\bar{s}, K, \text{new } \gamma e)$ . From the premises of (*prog*) and (*new*)

- (a)  $\triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K \rightsquigarrow \text{ref } \gamma \tau$ ,
- (c)  $\bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright \text{new } \gamma e : \text{ref } \gamma \tau$ ,
- (d)  $\varphi = \text{dom}(\bar{\zeta})$ ,
- (e)  $\bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright e : \tau$ , and
- (f)  $\gamma \in \varphi$ .

By (*k-new*) with (b) and (f)

$$(g) \bar{\tau} \triangleright K \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \tau.$$

We conclude  $\triangleright (\bar{s}, K \diamond (\text{new } \gamma [\cdot]), e)$  by (*prog*) with (a), (d), (e), and (g).

Case (T5)

Assume  $\triangleright (\bar{s}, K, \text{get } e)$ . From the premises of (*prog*) and (*get*)

$$(a) \triangleright \bar{s} : \bar{\tau},$$

$$(b) \bar{\tau} \triangleright K \rightsquigarrow \tau,$$

$$(c) \bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright \text{get } e : \tau,$$

$$(d) \varphi = \text{dom}(\bar{\tau}),$$

$$(e) \bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e : \text{ref } \gamma \tau, \text{ and}$$

$$(f) \gamma \in \varphi.$$

By (*k-get*) with (b) and (f)

$$(g) \bar{\tau} \triangleright K \diamond (\text{get } [\cdot]) \rightsquigarrow \text{ref } \gamma \tau.$$

We conclude  $\triangleright (\bar{s}, K \diamond (\text{get } [\cdot]), e)$  by (*prog*) with (a), (d), (e), and (g).

Case (T6)

Assume  $\triangleright (\bar{s}, K, \text{callcc } e)$ . From the premises of (*prog*) and (*callcc*)

$$(a) \triangleright \bar{s} : \bar{\tau},$$

$$(b) \bar{\tau} \triangleright K \rightsquigarrow \tau,$$

$$(c) \bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright \text{callcc } e : \tau,$$

$$(d) \varphi = \text{dom}(\bar{\tau}), \text{ and}$$

(e)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e : \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau$ .

By (*k-callcc*) with (b) and (d)

(f)  $\bar{\tau} \triangleright K \diamond (\text{callcc } [\cdot]) \rightsquigarrow \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau$ .

We conclude  $\triangleright (\bar{s}, K \diamond (\text{callcc } [\cdot]), e)$  by (*prog*) with (a), (d), (e), and (f).

Case (T7)

Assume  $\triangleright (\bar{s}, K, \text{throw } e_1 e_2)$ . From the premises of (*prog*) and (*throw*)

(a)  $\triangleright \bar{s} : \bar{\tau}$ ,

(b)  $\bar{\tau} \triangleright K \rightsquigarrow \tau_2$ ,

(c)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi_1 \triangleright \text{throw } e_1 e_2 : \tau_2$ ,

(d)  $\varphi_1 = \text{dom}(\bar{\tau})$ ,

(e)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi_1 \triangleright e_1 : \text{cont } \varphi_2 \tau_1$ ,

(f)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi_1 \triangleright e_2 : \tau_1$ , and

(g)  $\varphi_2 \subseteq \varphi_1$ .

By (*k-throw-l*) with (b), (c), (d), and (g)

(h)  $\bar{\tau} \triangleright K \diamond (\text{throw } [\cdot] e_2) \rightsquigarrow \text{cont } \varphi_2 \tau_1$ .

We conclude  $\triangleright (\bar{s}, K \diamond (\text{throw } [\cdot] e_2), e_1)$  by (*prog*) with (a), (d), (f), and

(h).

Case (T8)

Assume  $\triangleright (\bar{s}, K \diamond (\text{throw } [\cdot] e), v)$ . From the premises of (*prog*) and (*k-throw-l*)

- (a)  $\triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \diamond (\text{throw } [\cdot] e) \rightsquigarrow \text{cont } \varphi_2 \tau_1$ ,
- (c)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi_1 \triangleright v : \text{cont } \varphi_2 \tau_1$ ,
- (d)  $\varphi_1 = \text{dom}(\bar{\tau})$ , and
- (e)  $\bar{\tau} \triangleright K \rightsquigarrow \tau_2$ ,
- (f)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi_1 \triangleright e : \tau_1$ , and
- (g)  $\varphi_2 \subseteq \varphi_1$ .

By (*k-throw-r*) with (c), (d), (e), and (g)

- (h)  $\bar{\tau} \triangleright K \diamond (\text{throw } v [\cdot]) \rightsquigarrow \tau_1$ .

We conclude  $\triangleright (\bar{s}, K \diamond (\text{throw } v [\cdot]), e)$  by (*prog*) with (a), (d), (f), and (h).

Case (T9)

Assume  $\triangleright (\bar{s}, K \diamond ([\cdot] \langle \bar{\gamma} \rangle), \text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e)$ . From the premises of (*prog*), (*k-papp*), (*val*), and (*rec*)

- (a)  $\triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K \diamond ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright \text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ ,
- (d)  $\varphi = \text{dom}(\bar{\tau})$ ,
- (e)  $\bar{\tau} \triangleright K \rightsquigarrow \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ ,
- (f)  $\bar{\tau}; \emptyset; \langle \rangle \triangleright \text{rec } f \langle \bar{\rho} \rangle (x) \Rightarrow e : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ , and
- (g)  $\bar{\tau}; \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \langle f : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau \rangle \triangleright (\text{fn } x \Rightarrow e) : \tau$ .

By Lemma 7.2 with (f) and (g)

$$(h) \ \bar{\zeta}; \{\bar{\rho}, \bar{\epsilon}, \bar{\alpha}\}; \langle \rangle \triangleright (\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e)] : \tau$$

By Lemma 7.1 with (e) we have  $fcv(\tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]) = \emptyset$  which implies

$$(i) \ fcv(\bar{\gamma}) \cup fcv(\bar{\varphi}) \cup fcv(\bar{\tau}) = \emptyset.$$

Let  $v \equiv ((\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle \bar{\rho} \rangle(x) \Rightarrow e)])[\bar{\rho} := \bar{\gamma}]$ . By Lemma 7.3 with (h) and (i)

$$(j) \ \bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}].$$

By (val) with (j)

$$(k) \ \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}].$$

We conclude  $\triangleright (\bar{s}, K, ((\text{fn } x \Rightarrow e)[f := (\text{rec } f \langle x \rangle(\bar{\rho}) \Rightarrow e)])[\bar{\rho} := \bar{\gamma}])$  by (prog) with (a), (d), (e), and (k).

Case (T10)

Assume  $\triangleright (\bar{s}, K \circ ((\text{fn } x \Rightarrow e) [\cdot]), v)$ . From the premises of (prog), (k-app-r), (val), and (abs)

$$(a) \ \triangleright \bar{s} : \bar{\zeta},$$

$$(b) \ \bar{\zeta} \triangleright K \circ ((\text{fn } x \Rightarrow e) [\cdot]) \rightsquigarrow \tau_1,$$

$$(c) \ \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \tau_1,$$

$$(d) \ \varphi = \text{dom}(\bar{\zeta}),$$

$$(e) \ \bar{\zeta} \triangleright K \rightsquigarrow \tau_2,$$

$$(f) \bar{\tau}; \emptyset; \langle \rangle \triangleright \text{fn } x \Rightarrow e : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2,$$

$$(g) \bar{\tau}; \emptyset; \langle \rangle \triangleright v : \tau_1, \text{ and}$$

$$(h) \bar{\tau}; \emptyset; \langle x : \tau_1 \rangle; \varphi \triangleright e : \tau_2.$$

By Lemma 7.2 with (g) and (h)

$$(i) \bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright e[x := v] : \tau_2.$$

We conclude  $\triangleright (\bar{\tau}, K, e[x := v])$  by (*prog*) with (a), (d), (f), and (i).

Case (*T11*)

Assume  $\triangleright (\bar{s}, K \diamond (\text{new } \xi [\cdot]), v)$  where  $\bar{s}(\xi) = R$  for some  $R$  and choose  $l \notin \text{dom}(R)$ . From the premises (*prog*) and (*k-new*)

$$(a) \triangleright \bar{s} : \bar{\tau},$$

$$(b) \bar{\tau} \triangleright K \diamond (\text{new } \xi [\cdot]) \rightsquigarrow \tau,$$

$$(c) \bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright v : \tau,$$

$$(d) \varphi = \text{dom}(\bar{\tau}),$$

$$(e) \bar{\tau} \triangleright K \rightsquigarrow \text{ref } \xi \tau, \text{ and}$$

$$(f) \xi \in \text{dom}(\bar{\tau}),$$

From the premises of the rules (*stack*) and (*region*) with (a) and our assumptions about  $\xi$  and  $l$  we know  $\bar{\tau}(\xi) = \Sigma$  for some  $\Sigma$  and  $l \notin \text{dom}(\Sigma)$ . Let  $\bar{\tau}' = \bar{\tau}(\xi \mapsto \Sigma \langle l \mapsto \tau \rangle)$ . By Lemma 7.6 with (c) and (e)

$$(g) \bar{\tau}'; \emptyset; \langle \rangle; \varphi \triangleright v : \tau \text{ and}$$

$$(h) \bar{\tau}' \triangleright K \rightsquigarrow \text{ref } \xi \tau.$$



From the premise of *(val)* with (g)

$$(i) \bar{\tau}' ; \emptyset ; \langle \rangle \triangleright v : \tau.$$

By Lemma 7.6 with (a) and (i)

$$(j) \triangleright \bar{\tau} \langle \xi \mapsto R \langle l \mapsto v \rangle \rangle : \bar{\tau}'.$$

By Lemma 7.1 with (i) we know  $fcv(\tau) = \emptyset$  so by *(loc-live)*

$$(k) \bar{\tau}' ; \emptyset ; \langle \rangle \triangleright (\xi, l) : \text{ref } \xi \tau.$$

By *(val)* with (k)

$$(l) \bar{\tau}' ; \emptyset ; \langle \rangle ; \varphi \triangleright (\xi, l) : \text{ref } \xi \tau.$$

We conclude  $\triangleright (\bar{\tau} \langle \xi \mapsto R \langle l \mapsto v \rangle \rangle, K, (\xi, l))$  by *(prog)* with (d), (h), (j), and (l).

Case *(T12)*

Assume  $\triangleright (\bar{\tau}, K \diamond (\text{get } [\cdot]), (\xi, l))$  where  $\bar{\tau}(\xi) = R$  and  $R(l) = v$  for some  $R$  and  $v$ . From the premises of *(prog)*, *(k-get)*, and *(val)* and from inspection of the typing rules for locations

$$(a) \triangleright \bar{\tau} : \bar{\tau},$$

$$(b) \bar{\tau} \triangleright K \diamond (\text{get } [\cdot]) \rightsquigarrow \text{ref } \xi \tau,$$

$$(c) \bar{\tau} ; \emptyset ; \langle \rangle ; \varphi \triangleright (\xi, l) : \text{ref } \xi \tau,$$

$$(d) \varphi = \text{dom}(\bar{\tau}),$$

$$(e) \bar{\tau} \triangleright K \rightsquigarrow \tau,$$

$$(f) \xi \in \text{dom}(\bar{\tau}), \text{ and}$$

(g)  $\bar{\zeta}; \emptyset; \langle \rangle \triangleright (\xi, l) : \text{ref } \xi \tau$ .

From (f) we know (g) was derived by (loc-live) with premise  $\bar{\zeta}(\xi)(l) = \tau$ .

Using this fact with (a) we have from the premises of (*stack*) and (*region*) that

(h)  $\bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \tau$ .

By (*val*) with (h)

(i)  $\bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \tau$ .

We conclude  $\triangleright (\bar{s}, K, v)$  by (*prog*) with (a), (d), (e), and (i).

Case (T13)

Assume  $\triangleright (\bar{s}, K, \text{letreg } \rho \text{ in } e)$  and choose  $\xi$  such that  $\xi \notin \text{frn}(\bar{s}) \cup \text{frn}(e)$  and  $\bar{s} @ \xi \mapsto \langle \rangle$  exists, in other words,  $\xi \notin \text{dom}(\bar{s})$ .

From the premises of (*prog*) and (*letreg*)

(a)  $\triangleright \bar{s} : \bar{\zeta}$ ,

(b)  $\bar{\zeta} \triangleright K \rightsquigarrow \tau$ ,

(c)  $\bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright \text{letreg } \rho \text{ in } e : \tau$ ,

(d)  $\varphi = \text{dom}(\bar{\zeta})$ ,

(e)  $\bar{\zeta}; \{\rho\}; \langle \rangle; \varphi \uplus \{\rho\} \triangleright e : \tau$ , and

(f)  $\rho \notin \text{frv}(\tau)$ .

Let  $\bar{\zeta}' = \bar{\zeta} @ \xi \mapsto \langle \rangle$ . By Lemma 7.4 with (a) and (e) and our assumptions about  $\xi$

(g)  $\triangleright \bar{s} @ \xi \mapsto \langle \rangle : \bar{\tau}'$  and

(h)  $\bar{\tau}' ; \{\rho\} ; \langle \rangle ; \varphi \uplus \{\rho\} \triangleright e : \tau$ .

By (*k-pop*) with (b)

(i)  $\bar{\tau}' \triangleright K \diamond (\text{pop}; [\cdot]) \rightsquigarrow \tau$ .

From (f) we know  $\tau[\rho := \xi] \equiv \tau$  so by Lemma 7.3 with (h)

(j)  $\bar{\tau}' ; \emptyset ; \langle \rangle ; \varphi \uplus \{\xi\} \triangleright e[\rho := \xi] : \tau$ .

From (d) we have

(k)  $\varphi \uplus \{\xi\} = \text{dom}(\bar{\tau}')$ .

We conclude  $\triangleright (\bar{s} @ \xi \mapsto \langle \rangle, K \diamond (\text{pop}; [\cdot]), e[\rho := \xi])$  by (*prog*) with (g), (i), (j), and (k).

Case (*T14*)

Assume  $\triangleright (\bar{s} @ \xi \mapsto R, K \diamond (\text{pop}; [\cdot]), v)$ . From the premises of (*prog*) and (*k-pop*) and from inspection of the rule (*stack*)

(a)  $\triangleright \bar{s} @ \xi \mapsto R : \bar{\tau} @ \xi \mapsto \Sigma,$

(b)  $\bar{\tau} @ \xi \mapsto \Sigma \triangleright K \diamond (\text{pop}; [\cdot]) \rightsquigarrow \tau,$

(c)  $\bar{\tau} @ \xi \mapsto \Sigma ; \emptyset ; \langle \rangle ; \varphi \triangleright v : \tau,$

(d)  $\varphi = \text{dom}(\bar{\tau} @ \xi \mapsto \Sigma),$  and

(e)  $\bar{\tau} \triangleright K \rightsquigarrow \tau.$

By Lemma 7.5 with (a) and (c)

(f)  $\triangleright \bar{s} : \bar{\tau}$  and

$$(g) \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \tau.$$

From the premise of *(val)* with (g)

$$(h) \bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \tau.$$

Let  $\varphi' = \text{dom}(\bar{\zeta})$ . By *(val)* with (h)

$$(i) \bar{\zeta}; \emptyset; \langle \rangle; \varphi' \triangleright v : \tau.$$

We conclude  $\triangleright (\bar{\zeta}, K, v)$  by *(prog)* with (e), (f), and (i).

Case *(T15)*

Assume  $\triangleright (\bar{\zeta}, K \diamond (\text{callcc } [\cdot]), v)$ . From the premises of *(prog)*, *(k-callcc)*, and *(val)*

$$(a) \triangleright \bar{\zeta} : \bar{\zeta},$$

$$(b) \bar{\zeta} \triangleright K \diamond (\text{callcc } [\cdot]) \rightsquigarrow \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau,$$

$$(c) \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright v : \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau,$$

$$(d) \varphi = \text{dom}(\bar{\zeta}),$$

$$(e) \bar{\zeta} \triangleright K \rightsquigarrow \tau, \text{ and}$$

$$(f) \bar{\zeta}; \emptyset; \langle \rangle \triangleright v : \langle \varphi, (\text{cont } \varphi \tau) \rangle \rightarrow \tau.$$

Let  $n = |\bar{\zeta}| = |\bar{\zeta}|$ . Note that  $\bar{\zeta} \downarrow n = \bar{\zeta}$ . By *(cont-live)* with (d) and (e)

$$(g) \bar{\zeta}; \emptyset; \langle \rangle \triangleright (\varphi, \underline{n}, K) : \text{cont } \varphi \tau.$$

By *(val)* with (g)

$$(h) \bar{\zeta}; \emptyset; \langle \rangle; \varphi \triangleright (\varphi, \underline{n}, K) : \text{cont } \varphi \tau.$$

By (*app*) with (c) and (h)

$$(i) \bar{\tau}; \emptyset; \langle \rangle; \varphi \triangleright v(\varphi, \underline{n}, K) : \tau.$$

We conclude  $\triangleright (\bar{\tau}, K, v(\varphi, \underline{n}, K))$  by (*prog*) with (a), (d), (e), and (i).

Case (T16)

Assume  $\triangleright (\bar{\tau}, K \diamond (\text{throw}(\varphi, \underline{n}, K') [\cdot]), v)$ . From the premises of (*prog*) and (*val*)

$$(a) \triangleright \bar{\tau} : \bar{\tau},$$

$$(b) \bar{\tau} \triangleright K \diamond (\text{throw}(\varphi, \underline{n}, K') [\cdot]) \rightsquigarrow \tau, \text{ and}$$

$$(c) \bar{\tau}; \emptyset; \langle \rangle; \varphi' \triangleright v : \tau,$$

$$(d) \varphi' = \text{dom}(\bar{\tau}), \text{ and}$$

$$(e) \bar{\tau}; \emptyset; \langle \rangle \triangleright v : \tau.$$

From the premises of (*k-app-r*) with (b) and by inspection of the typing rules for values

$$(f) \bar{\tau}; \emptyset; \langle \rangle \triangleright (\varphi, \underline{n}, K') : \text{cont } \varphi \tau \text{ and}$$

$$(g) \varphi \subseteq \text{dom}(\bar{\tau}).$$

From (g) we know (f) was derived by (*cont-live*) with premises

$$(h) \bar{\tau} \downarrow n \triangleright K' \rightsquigarrow \tau \text{ and}$$

$$(i) \varphi = \text{dom}(\bar{\tau} \downarrow n).$$

Therefore  $n \leq |\bar{\tau}| = |\bar{\tau}|$  so using  $(|\bar{\tau}| - n)$  applications of Lemma 7.5 with (a) and (e)

(j)  $\triangleright \bar{s} \downarrow n : \bar{\tau} \downarrow n$ , and

(k)  $\bar{\tau} \downarrow n; \emptyset; \langle \rangle \triangleright v : \tau$ .

By (*val*) with (k)

(l)  $\bar{\tau} \downarrow n; \emptyset; \langle \rangle; \varphi \triangleright v : \tau$ .

We conclude  $\triangleright (\bar{s} \downarrow n, K', v)$  by (*prog*) with (h), (i), (j), and (l).

□

**Proposition 7.9 (*RC Progress*)**

If  $\triangleright M$  then either  $M \mapsto M'$  or  $M \equiv (\bar{s}, [\cdot], v)$ .

PROOF: Assume  $M \equiv (\bar{s}, K, e)$  and  $\triangleright M$ . The proof is by induction on the structure of  $e$ . If  $e$  is not a value then it reduces by either (*T1*), (*T2*), (*T4*), (*T5*), (*T6*), (*T7*), or (*T13*). Now if  $e \equiv v$  then we proceed by induction on the structure of  $K$ .

Case ( $[\cdot]$ )

Then  $M \equiv (\bar{s}, [\cdot], v)$ .

Case ( $K' \diamond ([\cdot] \langle \bar{\gamma} \rangle)$ )

From the premises of (*prog*), (*val*), and (*k-papp*)

(a)  $\triangleright \bar{s} : \bar{\tau}$ ,

(b)  $\bar{\tau} \triangleright K' \diamond ([\cdot] \langle \bar{\gamma} \rangle) \rightsquigarrow \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ ,

(c)  $\bar{\tau}; \emptyset; \langle \rangle \triangleright v : \forall \bar{\rho}, \bar{\epsilon}, \bar{\alpha}. \tau$ , and

(d)  $\bar{\tau} \triangleright K' \rightsquigarrow \tau[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$ .

From (d) we know the substitution  $[\bar{\rho} := \bar{\gamma}, \bar{\epsilon} := \bar{\varphi}, \bar{\alpha} := \bar{\tau}]$  exists. In other words we have  $|\bar{\rho}| = |\bar{\gamma}|$  and  $|\bar{\epsilon}| = |\bar{\varphi}|$  and  $|\bar{\alpha}| = |\bar{\tau}|$ . By Lemma 7.7 with (a) and (c) we know  $v \equiv (\text{rec } f \langle x \rangle)(\bar{\rho}) \Rightarrow e'$ . Therefore  $M$  reduces by (T9).

Case  $(K' \diamond ([\cdot] e'))$

Then  $M$  reduces by (T3).

Case  $(K' \diamond v' [\cdot])$

From the premises of (*prog*) and (*k-app-r*)

- (a)  $\triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K' \diamond (v' [\cdot]) \rightsquigarrow \tau_1$ , and
- (c)  $\bar{\zeta}; \emptyset; \langle \rangle \triangleright v' : \langle \varphi, \tau_1 \rangle \rightarrow \tau_2$ .

By Lemma 7.7 with (a) and (c) we know  $v' \equiv (\text{fn } x \Rightarrow e)$  and so  $M$  reduces by (T10).

Case  $(K' \diamond (\text{new } \gamma [\cdot]))$

From the premises of (*prog*) and (*k-new*)

- (a)  $\triangleright \bar{s} : \bar{\zeta}$ ,
- (b)  $\bar{\zeta} \triangleright K' \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \tau$ , and
- (c)  $\gamma \in \text{dom}(\bar{\zeta})$ .

From (a) and (c) and the premises of (*stack*) we know  $\gamma \equiv \xi$  such that  $\bar{s}(\xi) = R$ . Therefore  $M$  reduces by (T11) by choosing  $l \notin \text{dom}(R)$ .

Case  $(K' \diamond (\text{get } [\cdot]))$

From the premises of (*prog*), (*val*), and (*k-get*)

- (a)  $\triangleright \bar{s} : \bar{\tau}$ ,
- (b)  $\bar{\tau} \triangleright K' \diamond (\text{new } \gamma [\cdot]) \rightsquigarrow \text{ref } \gamma \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \langle \rangle \triangleright v : \text{ref } \gamma \tau$ , and
- (d)  $\gamma \in \text{dom}(\bar{\tau})$ .

From (a) and (d) and the premises of *(stack)* we know  $\gamma \equiv \xi$  and  $\xi \in \text{dom}(\bar{s})$ . Using this fact with (a) and (c) we know by Lemma 7.7 that  $v \equiv (\xi, l)$  and  $\bar{s}(\xi)(l) = v'$ . Therefore  $M$  reduces by (T12).

Case  $(K' \diamond (\text{pop}; [\cdot]))$

From the premises of *(prog)* and *(k-pop)*

- (a)  $\triangleright \bar{s} : \bar{\tau}$  and
- (b)  $\bar{\tau} \triangleright K' \diamond (\text{pop}; [\cdot]) \rightsquigarrow \tau$ .

From the premise of *(k-pop)* we know  $\bar{\tau}$  must have the form  $\bar{\tau}' @ \xi \mapsto \Sigma$  and so from the premises of *(stack)* we know  $\bar{s}$  has the form  $\bar{s}' @ \xi \mapsto R$ . Therefore  $M$  reduces by (T14).

Case  $(K' \diamond (\text{callcc } [\cdot]))$

Then  $M$  reduces by (T15).

Case  $(K' \diamond (\text{throw } [\cdot] e'))$

Then  $M$  reduces by (T8).

Case  $(K' \diamond (\text{throw } v' [\cdot]))$

From the premises of *(prog)* and *(k-throw-r)*

- (a)  $\triangleright \bar{s} : \bar{\tau}$ ,



- (b)  $\bar{\tau} \triangleright K' \diamond (\text{throw } v' [\cdot]) \rightsquigarrow \tau$ ,
- (c)  $\bar{\tau}; \emptyset; \langle \rangle \triangleright v : \tau$ ,
- (d)  $\bar{\tau}; \emptyset; \langle \rangle \triangleright v' : \text{cont } \varphi \tau$ , and
- (e)  $\varphi \subseteq \text{dom}(\bar{\tau})$ .

By Lemma 7.7 with (a), (d), and (e) we know  $v' \equiv (\varphi, \underline{n}, K'')$  and  $n \leq |\bar{\tau}|$ .

Therefore  $M$  reduces by (T16).

□

**Proposition 7.10 (RC Soundness)**

If  $\triangleright M$  then either  $M$  diverges or  $M \mapsto^* (\bar{s}, [\cdot], v)$  and  $\triangleright (\bar{s}, [\cdot], v)$ .

PROOF: Suppose  $\triangleright M$  and  $M \mapsto^* M'$  and that there exists no  $M''$  such that  $M' \mapsto M''$ . By induction on the length of the reduction sequence using Proposition 7.8 we have  $\triangleright M'$ . By Proposition 7.9 we have  $M' \equiv (\bar{s}, [\cdot], v)$ .

□

## BIBLIOGRAPHY

- [1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. *ACM SIGPLAN Notices*, 30(6):174–185, June 1995.
- [2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [3] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, August 1991.
- [4] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 88–97, 1999.
- [5] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992.
- [6] Nick Benton and Andrew Kennedy. Monads, effects and transformations. In *(HOOTS 99) Higher Order Operational Techniques in Semantics*. Electronic Notes in Theoretical Computer Science, volume 26, 1999.
- [7] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, New York, NY, USA, 1996. ACM Press.
- [8] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *Proceedings of the symposium on Principles of programming languages*, 2001.
- [9] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, March 1996.
- [10] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, April 1999.

- [11] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 262–275, New York, NY, USA, 1999. ACM Press.
- [12] Luis Damas. Principal type schemes for functional programs. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, January 1982.
- [13] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.
- [14] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.*, 102:235–271, 1992. Tech. Rep. 89-100, Rice University.
- [15] Daniel P. Friedman, Christopher T. Haynes, and Eugene E. Kohlbecker. *Programming with Continuations*. Springer-Verlag, 1984.
- [16] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [17] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [18] Robert Harper. *Programming Language : Theory and Practice*. Draft, 2002.
- [19] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ml. *Journal of Functional Programming*, 3 part 4:465–484, 1993. Preliminary version in POPL 91.
- [20] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298. ACM, ACM, August 1984.
- [21] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In *In The Fourth International Workshop on Higher Order Operational Techniques in Semantics, HOOTS*, volume 41 of *Electronic Notes in Theoretical Computer Science*, September 2000.
- [22] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

- [23] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.
- [24] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25, 1993.
- [25] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, John Fairbairn, Joseph Fasel, Maria Guzman, Keven Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partian, and John Peterson. Report on the programming language haskell, version 1.2. *Sigplan*, 27(5), May 1992.
- [26] Pierre Jouvelot and David Gifford. Reasoning about continuations with control effects. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, Portland, OR, June 1989. ACM, ACM Press.
- [27] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 303–310. ACM Press, January 1991.
- [28] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised<sup>5</sup> report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
- [29] Richard Kieburtz. Taming effects with monadic typing. *ACM SIGPLAN Notices*, 34(1):51–62, January 1999.
- [30] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.
- [31] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp Symbol. Comput.*, 8:193–341, 1995.
- [32] Xavier Leroy. Polymorphism by name for references and continuations. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231, January 1993.
- [33] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 291–302, January 1991.
- [34] Jon M. Lucassen and David Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, ACM Press, January 1988.

- [35] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.
- [36] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.
- [37] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [38] Eugenio Moggi and Fabrizio Palumbo. Monadic encapsulation of effects : a revised approach. In *(HOOTS 99) Higher Order Operational Techniques in Semantics*. Electronic Notes in Theoretical Computer Science, volume 26, 1999.
- [39] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
- [40] Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, volume 34.9 of *ACM Sigplan Notices*, pages 8–17, N.Y., September 27–29 1999. ACM Press.
- [41] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical monograph prg-11, Oxford University Computing Laboratory, Programming Research Group, 1974.
- [42] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
- [43] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In the *IEEE Symposium on Logic in Computer Science*, pages 162–173, June 1992.
- [44] David Tarditi. "Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML". PhD thesis, "Carnegie Mellon University", 1996.
- [45] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices*, 31(5):181–192, May 1996.
- [46] Mads Tofte. *Operational semantics and polymorphic type inference*. PhD thesis, University of Edinburgh, 1987.
- [47] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, November 1990.

- [48] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 20, 1998.
- [49] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ml kit (for version 3). Technical Report DIKU-TR-98/25, University of Copenhagen, 1998.
- [50] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report 93/15, Department of Computer Science, Copenhagen University, July 1993.
- [51] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value calculus using a stack of regions. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, 1994.
- [52] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.
- [53] Andrew Tolmach. Optimizing ML using a hierarchy of monadic types. *Lecture Notes in Computer Science*, 1473, 1998.
- [54] Magnus Vejlstrup. Multiplicity inference. Master's thesis, University of Copenhagen, 1994.
- [55] Philip Wadler. The marriage of effects and monads. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 63–74. ACM, June 1999.
- [56] David Walker and Kevin Watkins. On regions and linear types. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, 2001.
- [57] Andrew K. Wright. Typing references by effect inference. In *European Symposium on Programming*, pages 473–491. Springer-Verlag, 1992.
- [58] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995. Preliminary Version is Polymorphism for Imperative Languages without Imperative Types, Rice Technical Report TR93-200.
- [59] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.
- [60] Silvano Dal Zilio and Andrew D. Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, 2000.