Region-based Program Specialization

An Operational Approach to Polymorphic Offline Partial Evaluation for ML-like Languages

Simon Helsen

Dissertation zur Erlangung des Doktorgrades der
Fakultät für Angewandte Wissenschaften der
Albert-Ludwigs-Universität Freiburg im Breisgau

2002

Dekan:	Prof. Dr. Thomas Ottmann, Universität Freiburg
Erstreferent:	Prof. Dr. Peter Thiemann, Universität Freiburg
Zweitreferent:	Prof. Dr. Fritz Henglein, Københavns Universitet (Denmark)

Datum der Promotion: 27 September 2002

Zusammenfassung

Programmspezialisierung oder partielle Auswertung ist eine bewährte Methode zur Programmoptimierung. Sie ist besonders dann sehr effektiv, wenn ein Teil der Eingabe des Quellprogramms statisch, also wenig veränderlich ist; denn partielle Auswerter spezialisieren Programme, indem sie möglichst viele Operationen durch das aggressive Weiterleiten von Konstanten und das Entfalten von Funktionen bezüglich der statischen Eingabe reduzieren. Eine besonders viel benutzte Variante ist die *offline* partielle Auswertung, die durch Bindungszeitanalyse ermittelt, welche Operationen spezialisiert werden können. In der Vergangenheit haben offline-Systeme sich jedoch überwiegend auf naive Techniken zur Annotation von Bindungszeiten oder einfach getypte Programmiersprachen beschränkt. Das volle Potential der Methode wurde dadurch nicht annähernd ausgeschöpft; eine Integration von fortgeschrittenen Annotationstechniken und eine Erweiterung auf komplexere Typsysteme blieben, sowohl theoretisch als auch praktisch, offene und interessante Fragen.

In der vorliegenden Arbeit präsentieren wir im Kontext ML-ähnlicher funktionaler Sprachen eine zufriedenstellende Antwort auf diese Fragen. Unsere grundlegende Idee ist es, Techniken aus dem Bereich der Speicherverwaltung zu erweitern und in den Bereich der Programmspezialisierung zu übertragen, um eine neue, rein operationelle Theorie der *offline* polymorphen Spezialisierung zu entwickeln. Wir erweitern dazu den Regionenkalkül von Tofte und Talpin, einem polymorph getypten Lambda-Kalkül mit Annotationen zur expliziten Beschreibung von Speicherzuweisung und -freigabe für unsere Zwecke. Mit dem Regionenkalkül lassen sich funktionale Sprachen mit Hindley-Milner-Typsystem ohne die herkömmlichen dynamischen Speicherverwaltungstechniken implementieren. Speicherverwaltungsoperationen werden stattdessen durch eine *statische* Typ-und-Effektanalyse in wohlgetypte ML-Programme eingefügt. Der formale Korrektheitsbeweis unserer Spezialisierungsmethode benutzt zwei theoretische Bausteine: einen neuen Beweis der Typkorrektheit des Regionenkalküls sowie eine neue, korrekt bewiesene Gleichungslogik für regionsannotierte Ausdrücke.

Für den ersten Baustein formulieren wir eine neue Typkorrektheitseigenschaft des Region-und-Effektsystems des Regionenkalküls. Diese ist notwendig, um formal garantieren zu können, dass Regionen nur dann freigegeben werden, wenn die in ihr gespeicherten Objekte nicht mehr benutzt werden können. Während der ursprüngliche Typkorrektheitsbeweis von Tofte und Talpin auf einer operationellen Großschritt-Semantik und einer komplexen koinduktiv definierten Relation basiert, verwenden wir zwei neue operationelle Kleinschritt-Semantiken. So können wir einfache Induktionsbeweise mit den syntaktischen Methoden von Wright, Felleisen und Harper führen. Die erste der beiden Semantiken beschreibt keinen expliziten Speicher. Sie ist einfach, elegant und erlaubt unkomplizierte Beweise. Die zweite imperative Semantik ist zwar komplizierter als die erste, sie lässt aber die Modellierung von Operationen auf Speicherzellen zu. Für die seiteneffektfreien und monomorphen Fragmente der beiden Semantiken zeigen wir die Gleichwertigkeit zur ursprünglichen Semantik von Tofte und Talpin. Dadurch erhalten wir auch als Spezialfall einen alternativen und einfacheren Typkorrektheitsbeweis der Großschritt-Formulierung, welcher auch über die Programmspezialisierung hinaus von Interesse ist.

Der zweite theoretische Baustein unseres Programmspezialisierers, eine getypte Gleichunglogik für regionsannotierte Ausdrücke, ermöglicht formale Beweise relativ zur dynamischen Semantik des Regionenkalküls. Wir benutzen zu diesem Zweck eine kleine Variante des Tofte-Talpin-Kalküls, die rekursive Funktionen erster Klasse und polymorphe Rekursion über Regionen beinhaltet. Mit Hilfe von Bisimulation — einer Beweistechnik, die ursprünglich für Prozesskalküle entwickelt wurde — sind wir in der Lage, die Korrektheit unserer Gleichungslogik bezüglich kontextueller Äquivalenz im Sinne von Morris zu zeigen. Dazu führen wir mit Methoden von Gordon, Abramsky, et al., einen Begriff der applikativen Bisimilarität für unseren Kalkül ein, welcher sowohl Typen als auch Effekte berücksichtigt. Unter Verwendung einer Beweismethode von Howe, zeigen wir, dass die Bisimilarität eine Kongruenzrelation ist. Daraus folgt einerseits die Gleichheit von kontextueller Äquivalenz und Bisimilarität, sowie andererseits, dass die durch unsere Gleichungslogik induzierte Theorie in der Bisimilaritätsrelation enthalten ist.

Das Zusammenspiel beider Bausteine ergibt die gewünschte neue Methode zur offline partiellen Auswertung von funktionalen Programmiersprachen mit ML-ähnlichem Typsystem. Unsere Programmspezialisierungsmethode beinhaltet eine polymorphe Bindungszeitanalyse mit polymorpher Rekursion, welche die Regioninferenz um eine *Constraint*-Analyse erweitert. Die Verbindung zwischen Regionen und Programmspezialisierung folgt der Erkenntnis, dass Bindungszeiten als Eigenschaften von Regionen betrachtet werden können. Wir formulieren Spezialisierung als Erweiterung der operationellen Kleinschritt-Semantik des Regionenkalküls ohne explizitem Speicher. Wir beweisen die Korrektheit der Constraint-Analyse bezüglich der Spezialisierungssemantik mit rein syntaktischen Techniken. Zusätzlich zeigen wir mit Hilfe der getypten Gleichungslogik unseres Regionenkalküls, dass jeder Schritt des Spezialisierers die kontextuelle Äquivalenz erhält.

Durch die Integration zweier scheinbar unverwandter Forschungsgebiete — Programmspezialisierung und Speicherverwaltung — haben wir somit an Hand eleganter und operationeller Methoden eine neue, korrekte und mächtige offline partielle Auswertungstechnik entwickelt.

Summary

Program specialization or partial evaluation is a proven program optimization technique dating back to the early seventies. It is based on aggressive constant propagation and function unfolding, and is usually employed as an extra tool in the program compilation process. Partial evaluation works remarkably well when the program input can be factored in static, less variable, and dynamic, often changing, parts. A partial evaluator transforms the source program by reducing as much operations as possible with respect to the static input. The resulting program is usually faster than the original program whenever the latter has to be executed with its static input. One common flavor of program specialization is *offline* partial evaluation. It employs a program analysis to determine what operations can be reduced at specialization time. Historically, offline systems used simple binding-time annotation techniques, or considered simply-typed programming languages. These approaches have hampered offline methods in reaching their full potential. In this thesis, we aim to address these limitations.

We describe the development of a new and entirely operational theory for offline polymorphic specialization of ML-like languages by combining techniques from dynamic memory management with program specialization. Our approach is based on the region calculus of Tofte and Talpin, a polymorphically typed lambda calculus with annotations that make memory allocation and deallocation explicit. It is intended as an intermediate language for implementing Hindley-Milner typed functional languages without traditional trace-based garbage collection. Static region and effect inference can be used to annotate a statically-typed ML program with memory management primitives. The formal correctness proof of our novel specialization technique based on a region type system requires two theoretical building blocks: a type soundness proof for the region calculus and a correctly proven equational theory between region-annotated terms.

To establish the first building block, we present a new formulation of type soundness for the region and effect system of the region calculus. Such a property is crucial to guarantee safe deallocation of regions, *i.e.*, deallocation should only take place for objects which are provably dead. The original soundness proof by Tofte and Talpin is based on a big-step evaluation-style operational semantics and requires a complex co-inductive safety relation. We reformulate type soundness with respect to two small-step operational semantics. Following the standard syntactic approach of Wright, Felleisen and Harper, we obtain simple inductive proofs. The first semantics is storeless. It is simple and gives rise to perspicuous proofs. The second imperative semantics is slightly more complicated than the storeless formulation. However, its additional expressiveness allows us to model operations on reference cells with destructive update. A pure monomorphic fragment of both small-step semantics is then proven equivalent to the original big-step operational approach of Tofte and Talpin. As a byproduct, we obtain an alternative and simpler type soundness proof for their evaluation-style formulation.

The second theoretical building block for our program specializer provides a mechanism to reason about the dynamic semantics of region-annotated programs. To this end, we introduce a typed equational theory, which we formalize for a small variation of the Tofte-Talpin region calculus, including first-class recursive function objects with polymorphic recursion for regions. Using bisimulation, a proof method originating from

work on process calculi, we prove that the equational theory is sound with respect to Morris-style contextual equivalence. Following methods by Gordon, Abramsky and others, we specify a notion of applicative bisimilarity based on types and effects. With a proof technique due to Howe, we are able to show that our form of bisimilarity forms a congruence relation. This result allows us to prove that the bisimilarity relation equals contextual equivalence on the one hand, and that it includes the equational theory on the other.

Putting together these two theoretical building blocks yields a new method for offline partial evaluation of functional programming languages with an ML-style typing discipline. Our method comprises a polymorphic binding-time analysis with polymorphic recursion, which is conceived as a constraint analysis on top of region inference. The relation between program specialization and regions is a result of regarding binding times as properties of regions. Specialization is specified as a small-step semantics, extending the earlier storeless formulation with two-level terms. Again using syntactic proof techniques, we prove soundness of the constraint analysis with respect to the specializer. In addition, we prove that specialization is sound, *i.e.* that it preserves the call-by-value semantics of the region calculus. To achieve this, it is sufficient to show that the reductions of the specializer preserves contextual equivalence of region terms. This result is proven with our equational theory for the region calculus.

Bringing together two seemingly unrelated research areas, program specialization and memory management, we obtain a new, powerful and correctly proven offline partial evaluation technique by employing elegant operational methods in semantics.

Acknowledgments

Although it is absolutely impossible to thank everybody who helped me to complete this work, an attempt is permitted at this point.

I am most grateful to Peter Thiemann, my supervisor, boss and mentor alike. His never ending support and belief in my capabilities have been important factors in the successful end of this thesis. In difficult times, he did *the right thing*™. It has also been an honor to be his first doctoral student.

Over the last 5 years, I had the joy to work together with many interesting and competent people: in Tübingen, it was Mike Sperber's online help and his fantastic knowledge about programming languages and their implementation that made me become a better computer scientist. Professor Klaeren, our boss, was extremely flexible and supporting. Christoph Schmitz I wish to thank for his unconditional help with worldly stuff, not the least the move from Tübingen to Freiburg. I enjoyed tutoring a course given by Martin Plümicke.

Via e-mail, I learned quite a bit from Steven Weeks about *Standard ML* and his *MLton* compiler. It was a pleasure to be a co-author with Cristiano Calcagno in London. At a rather important time, Olivier Danvy and his group in Aarhus, Denmark, gave me a rustic stay in BRICS.

In Freiburg, a most interesting and interested colleague and friend was Georg Struth. I have good memories of sharing the office with him. In the last year and a half, I also enjoyed many fruitful discussions with Matthias Neubauer. David Basin and his group here in Freiburg were, more often than not, good coffee-break company. Ina Eckert, the secretary and Martin Preen, the system administrator took my questions and complaints on a regular basis. Beate Maurer was a very helpful librarian. Other fine colleagues were Ulf Schuenemann, Adrian Fiech, Jochen Walter and Olga Levchuk. I wish to thank Georg, Matthias and my father for making the German abstract both *German* and an *abstract*.

On a personal level, I would like to thank many people from many countries for their friendship, most of them residing in Belgium and Germany. I do not attempt to name them here since such a list would be too long and incomplete. Particular grateful I am to the KHG-choir in Tübingen as well as the Heinrich-Schütz-Kantorei and Herdmer-vokalensemble in Freiburg for many wonderful hours of singing. The WGs in Zähringen and the Zassiusstrasse have been very pleasant and I thank Ulrike and Lydia for teaching me that you cannot take anything for granted in life. A bathroom in the Stusie and *toastbrot* did some wonderful things too.

My parents I cannot thank enough. Without their unconditional and lasting support, a lifetime long, I would not be writing any of this.

Kate, thank you for being there, you make it all worth it!

Freiburg, June 2002

Contents

List of Figures

1 INTRODUCTION

Programming language research is an important discipline in computer science: it deals with the design, description, semantics and implementation of formal languages suitable for software development. High-level programming languages have come a long way since the invention of Cobol and Fortran in the early 50's. To facilitate program development, researchers and practitioners enhanced and invented new programming languages, all with the same goal: to improve their abstraction and design capabilities.

At all times, programmers are (or should be) concerned about two things: *program correctness* and *execution speed*. To obtain correct programs, a programmer is ideally supported by suitable abstraction mechanisms in the language. Achieving acceptable execution speed on top of this, is mostly the job of a compiler for that language.

While language abstractions and compiler implementations are important means to achieve the above mentioned goals, other tools can be employed to aid the software developer with her quest for correctness and speed. One such tool is a *program specializer*. Very much like a compiler, it transforms a source program into another program. However, unlike a compiler, a specializer is not primarily concerned with the translation of the program into a low-level representation such as machine language. Rather, it produces a new program by shifting abstraction (input data) into execution speed (program text), obtaining a hopefully faster program. In other words, a specializer trades abstraction at the programmer level for speed at the implementation level.

In this thesis, we deal with the formal design and correctness of a specializer for (a side-effect free subset of) the functional programming language *Standard ML* [103]. Standard ML, abbreviated by SML or ML, is a call-by-value imperative-functional programming language with a Hindley-Milner style polymorphic type system [31, 100]. A fairly large part of the thesis deals with a formal *correctness* proof of the specializer, *i.e.* the property that the program specializer produces a valid executable target program that has the same semantics as the original source program.

1.1 Context and Motivation

Before we discuss the individual results of this dissertation, we sketch context and motivation of three topics in programming language research that form the foundations for this thesis: the formal semantics of programming languages, regions and their use in memory management, and program specialization as an optimizing program transformation.

1.1.1 Programming Language Semantics

Every program has a *meaning* or *semantics*. Often, the intended meaning of a program is defined by an implementation of the language or an informal description. Either approach is unsatisfactory: every time a programming language is re-implemented, its semantics may change. Worse, it is really hard to figure out what the intended meaning of a language construct is without executing it in *every* possible context. Modern languages often come with an informal description of what every feature of the language "means". While this is an improvement over an implementation, it still gives rise to many misunderstandings and inconsistencies. Moreover, it is impossible to prove program correctness in a mathematical sense since there is no mathematical agreement on the meaning of the language.

The desire to formally describe the semantics of a program and its execution goes back to Landin [85] and has been a major branch in programming research ever since. Roughly speaking, there are three main approaches to formal language semantics:

The Denotational Approach A denotational semantics is directly tied to the syntax of the language. It maps a program construct onto some mathematical object and describes meaning by looking at the mathematical model. In other words, the abstract values in the model *denote* a syntactic construct of the language. The theory of mathematical objects needed to describe programming language semantics is known as *domain theory* [56]. The denotational approach was invented by Scott and Strachey [129] and has been refined and extended to a wide variety of languages and programming systems. For example, the Scheme language, a dialect of Lisp [132], is standardized by means of a denotational description [84].

The Axiomatic Approach There are many forms of axiomatic semantics. They all consist of a set of axioms and logical deduction rules for deriving properties of programs. The best known axiomatic proof system is Floyd-Hoare logic [40, 74]. It can be used by programmers to reason about the correctness of a particular program, for example by writing loop invariants. Another flavor of an axiomatic semantics is an equational proof system, which employs syntactic equality rules. It makes expressions equal if they belong to the same equivalence class induced by the equational system.

The Operational Approach An operational semantics is probably the closest to an actual implementation of a programming language. However, unlike a compiler, it describes the meaning of a program by evaluating it in an abstract machine. An early example of such an operational model is the SECD abstract machine semantics by Landin [85, 86]. In an operational setting, semantic equivalence is usually defined as Morris-style *contextual equivalence* [108]. Two expressions are contextual equivalent whenever no amount of programming can tell them apart, *i.e.* arbitrary contexts of the two expressions both yield the same value.

There are two important flavors of operational semantics: the *big-step* (also known as *evaluation-style* or *natural*) semantics and the *small-step* (or *transition*) semantics. The big-step semantics is reminiscent of the axiomatic approach in that it

is defined by axioms and deduction rules. In such a system, the semantics of an expression is described by a judgment relating it to its value. The relation is inductively constructed from the relations between the expression's subparts and their values. An extensive example of a natural semantics is found in *the definition of Standard ML* [103]. In contrast, a Plotkin-style [122] transition semantics describes the meaning of an expression by one-step syntactic rewrite rules. The transitive closure of the transition relation relates a program to its value, that is, whenever it exists.

The three approaches above usually deal with the *dynamic* or *execution* semantics of a programming language. However, many modern languages also have a *static* semantics, that is often formulated by means of a *type system*. A type system is really a flavor of an axiomatic semantics. A *type* is a property of an expression and conservatively approximates its value. Type systems are usually designed to be decidable at compile-time, *i.e.* a type checker or type inference engine can validate the type properties of a program before it is executed. This explains the term "static" semantics.

In Section 2.2, we give an overview of the operational techniques used in this thesis by means of the simply-typed lambda calculus with a recursion operator. For further reading on formal language semantics, we refer to the many introductory and advanced textbooks on the subject [55, 104, 133, 152, 157].

1.1.2 Regions for Memory Management

Almost all general-purpose programming languages today cater for *dynamic data structures*. Such structures contain data of unbounded size that cannot be predicted at compile time. Memory management for dynamic data structures is a difficult problem in programming. While memory allocation is dictated by the problem at hand, there is considerable freedom in memory deallocation. If deallocation happens too late, the program suffers from memory bloat and space leaks, which impede performance. If deallocation happens too early, there might be dangling pointers into deallocated memory. Dereferencing a dangling pointer is unsafe and can lead to a crash, or worse, to wrong results.

Some languages, like C, C++ or Pascal, leave the deallocation problem entirely to the programmer, whereas others, like Lisp, Smalltalk, Java, and ML, perform automatic deallocation by incorporating a *trace-based garbage collector* into the runtime system [156]. While the programmer-based solution is immensely error-prone, programs can in principle be tuned for optimal memory use. Traditional garbage collection avoids a large class of program errors, but it has some problems too. Since the garbage collector is, in general, unaware of the semantics of the running program, it must preserve all pointers reachable from a given set of root pointers. This set is a conservative approximation of the set of pointers that will actually be used by the program. As a consequence, deallocation might happen too late, which can lead to space leaks. In addition, trace-based garbage collection takes extra, non-productive time and can cause erratic pauses in the execution of programs, hampering its use for real-time applications. Finally, interoperability between garbage collected languages and non-garbage collected languages is

difficult.

The desire to have compile-time control of memory management, retaining deallocation safety, motivates the use of *regions* [44, 83, 136, 149]. The basic idea of region-based memory management is to split memory into different parts or regions that are allocated in a stack-like manner, directed by a construct of the language. Deallocation is instantaneous, it just pops the topmost region from the stack. Region annotations can be automatically inferred by means of a type and effect analysis [11, 146, 147, 150]. Such a program analysis eliminates the runtime cost of trace-based garbage collection without losing deallocation safety. Based on a region calculus introduced by Tofte and Talpin [150], the ML-kit [148], a compiler for Standard ML, implements a practical implementation of region inference for a realistic language. The ML-kit does not rely on runtime garbage collection[1] to obtain automatic memory management.

Since its introduction, region-based memory management has been improved [6, 12] and transferred to other languages [80, 94, 153]. In addition, many different flavors of the region calculus have been developed [7, 19, 34, 42, 54, 72, 154, 155], usually to cater for more flexible and powerful memory management.

Region inference conservatively approximates data dependencies, where a region can be considered as an abstract program point [112]. Region analysis is fairly accurate due to its region polymorphism and the fact that it builds on the polymorphic Hindley-Milner type system. This is an attractive feature for a program specializer as we shall explain below. In Section 2.3, we give a formal description of the Tofte-Talpin region calculus as it forms the basis for our ML specializer.

Research on regions as an alternative for dynamic memory management is in a fairly young stage. There are no text books available on the subject, however, comprehensive material can be found in the paper by Tofte and Talpin [150] and the user manual of the ML-kit [148].

1.1.3 Program Specialization

The execution of a program p can be abstracted by the following simple equation: $[p](in) = out$, where in is its input and out its output. In many real world problems, the parameter in can be further split into several input parameters. Consider the case where we have two inputs, so the program equation is now as follows: $[p](in_1, in_2) = out$. Whenever the first input in_1 *varies less frequently* than the second input in_2, it may be worthwhile to *specialize* the source program p with respect to in_1. The specialization process evaluates the parts of p that only depend on in_1 and generates code for the remaining operations. The input in_1 is then called the *static* input, whereas in_2 is referred to as the *dynamic* input. The specialized program resulting from p and in_1 is the *residual* program. Running the residual program, say p_{in_1}, more than once, each time with a different input in_2, usually yields an overall speedup. Of course, we have to make sure that the equation $[p_{in_1}](in_2) = out$ holds as well. This process is called *program specialization*, although the term *partial evaluation* is also quite common in the

[1] The latest version 4 of the ML-kit actually incorporates a two-space copying collector to optimize memory management within one region.

literature [10, 25, 82].

There are many applications for program specialization. A typical example is the generation of a compiler by specializing an interpreter with respect to the interpreted program. In this scenario, the source program p is an interpreter `int`, the static input in_1 is a program p', and the dynamic input in_2 is the input $in_{p'}$ to the program p'. Let us call the specializer itself `mix`[2]. We then have the equations $[\texttt{mix}](\texttt{int}, p') = \texttt{int}_{p'}$, where $\texttt{int}_{p'}$ is the residual program and $[\texttt{int}_{p'}](in_{p'}) = out$ also holds. Under proper conditions, the program $\texttt{int}_{p'}$ is in A-Normal Form [39], which is composable with the back-end of a realistic compiler [130]. The equations for the interpreter are known as the *first Futamura Projection* after Yoshihito Futamura, who formulated them in the early seventies [41]. The projections were independently discovered by Turchin [151].

There is a plethora of applications for partial evaluation and we name only a few for exemplification: specialization of domain-specific languages [8], automatic parser generation [131], enforcing security properties in programs [143], faster Fourier transformations [89], etc.

A partial evaluator obtains the residual program by executing those operations that only depend on the static input, whereas it generates code for the remaining program text. Most techniques for program specialization are *syntax-directed*, *i.e.* specialization is guided by the text of the source program. Recently, non syntax-directed methods have been introduced such as type specialization [79] and type-directed specialization [32]. Since it is unclear how well these techniques scale to realistic examples, we do not consider them in our work. For a comparison of Danvy's type-directed partial evaluator and the syntax-directed approach, see [66].

Syntax-directed partial evaluation for functional programming can be further classified into two categories: the *online* and *offline* systems [59]. The online method makes the decision of whether an expression in the source program has to be reduced *on the fly*, *i.e.* while specializing. In contrast, offline partial evaluation precedes the specialization phase with a *Binding-Time Analysis* or BTA [70, 110, 140]. A BTA is a program analysis that determines what constructs in a program can be executed at specialization time. It yields an annotation of each construct with a binding time indicating in which phase it is to be executed. Because the offline approach is bound to make conservative approximations, it is inherently less powerful than the online method. However, the main problem with the online approach is that it is much more inefficient and its more aggressive nature makes it hard to generate compact residual code.

In the early days of program specialization, it was common to specify a BTA by abstract interpretation [27, 109, 110]. Following the reformulation of the BTA in a type-based setting with efficient inference algorithms [47, 71], most modern offline partial evaluators use type inference for the binding-time analysis of their specialization [14, 142]. In our work, we also follow the latter approach as we intend to reuse region and effect inference, which is based on type inference methods. A binding-time annotated expression is often called a *two-level* term [111] and a binding-time analysis can be understood as a static semantics[3] for offline program specialization.

[2]This name is historical and due to Ershov [37] who introduced the notion of *mixed computation*.

[3]The term "static semantics" is not to be confused with "static phase". In fact, the dynamic semantics

A BTA that assigns each construct a fixed binding time is a *monovariant* analysis. Many existing partial evaluators rely on a monovariant BTA [21, 70, 110] and many applications have been studied in the context of these systems. A more expressive policy would consider a binding-time annotation that depends on the context of each particular call of a function, say. This is basically a *polyvariant* analysis [22]. Several applications benefit from binding-time polyvariance, for instance modular program specialization [35], separate compilation via specialization [63, 67] and enforcing security properties by specialization [143]. However, in its most general form, a polyvariant analysis cannot know in advance how many differently annotated versions of a function are required. It relies on user guidance or ad-hoc techniques to ensure a finite number of annotations. Another alternative is to consider a *polymorphic* binding-time analysis [36, 73] which transfers the idea of parametric polymorphism from types to binding times. The program specializer developed in our thesis is binding-time polymorphic.

Extensive coverage of partial evaluation in general can be found in a textbook by Jones et al. [82], a collection of articles in [45] and [61], and also two tutorials [25, 81]

1.1.4 Operational Methods for Specialization

All the newly introduced dynamic semantics in this thesis follow the small-step operational approach. Although the choice of a transition semantics over an evaluation-style formulation is usually a matter of taste, we have been influenced by the work of Hatcliff and Danvy [60] who formalize program specialization with syntactic operational techniques. Moreover, compared to denotational methods, operational methods do not require a lot of mathematical background. This is one reason why the first formulation of the region calculus was operational [149]. Also, it is currently far from clear how to give a denotational description for a region calculus. To the best of the author's knowledge, only the paper by Banerjee, et al. [9] formulate type soundness for a region calculus by relating it to the denotational model of an extended polymorphic lambda calculus.

1.2 Main Results

The main contribution of this dissertation is the development of a novel and entirely operational model for polymorphic offline partial evaluation of a pure subset of an ML-like functional programming language. Moreover, our work provides a complete and formal correctness proof of the specialization model presented. We discuss specific original results in the subsequent part of this section.

1.2.1 Type Soundness for a Storeless Region Calculus

Type soundness for a region calculus implies that the deallocation annotations inserted by the region and effect system are safe with respect to the execution semantics, *i.e.* deallocation only takes place for objects which are provably dead. Tofte and Talpin give such a soundness proof, however, it is complicated and un-intuitive, partly due to the

of offline partial evaluation coincides with the static or specialization phase

fact that they combine type soundness with translation soundness. The latter property expresses that the execution behavior of an annotated program coincides with the semantics of the un-annotated program. Because of the excellent program dependency analysis properties of region inference, we intend to design a binding-time analysis as an extension of region inference. However, to prove the specializer correct with respect to the region-based binding-time analysis, we have to augment the "standard" type soundness proof in one way or another. Considering the complexity of the original Tofte-Talpin proof, we decided to pursue a purely syntactic approach based on a Plotkin-style transition semantics.

We reformulate the store-based big-step operational semantics of the region calculus in a relatively simple storeless transition semantics. Type soundness of the storeless model is proven with syntactic methods due to Wright, Felleisen and Harper [57, 158].

The storeless formulation of the region calculus and its syntactic type soundness proof have been published in [68].

1.2.2 Imperative Extensions

While the storeless formulation is extremely elegant and gives rise to perspicuous proofs, it is desirable to model a calculus with references and destructive updates. Therefore, we introduce a new calculus with an explicit store that extends the Tofte-Talpin calculus with operations on references as they are actually implemented in the ML-kit [148]. We also use syntactic proof techniques to prove the calculus sound with respect to region and effect inference, although we only consider a monomorphic fragment. Adding polymorphism and recursion makes the proofs more technical, but it does not require new insights with respect to the proofs for the storeless calculus.

Moreover, we prove the store-passing transition semantics of the calculus with references equivalent to the evaluation-style formulation of Tofte and Talpin. This result is then used to formulate an alternative soundness proof for the original version of the region calculus on which the ML-kit is based. Finally, we also relate the store-passing semantics to the storeless semantics.

This work has been done in collaboration with Cristiano Calcagno and has been published in [18].

1.2.3 An Equational Theory for a Region Calculus

Although type soundness is an important property for a region calculus, it does not provide for a semantical framework to reason about region-annotated programs. The translation correctness result of Tofte and Talpin [150] addresses this issue to some degree, but it does not give the implementor of a compiler or partial evaluator a usable means for proving the correctness of program transformations on region-annotated terms.

To prove our program specializer correct, we have to show that it preserves the call-by-value semantics of the region calculus. As far as we know, there exists to date no semantic model for the region calculus we could exploit to prove this part of the specialization correctness. Hence, we formulate a syntactic equational theory for a variation of the region calculus and prove it sound with respect to contextual equivalence. Soundness

is achieved by introducing an appropriate notion of *bisimilarity* [101] (a technique we address in Section 2.2.5). It provides a co-inductive proof principle for showing equivalences in our region calculus. We show that bisimilarity includes the equational theory and that it coincides with contextual equivalence.

These results are submitted for publication [65].

1.2.4 Polymorphic Specialization for ML-like languages

We specify a binding-time analysis and specialization semantics for a call-by-value lambda calculus with ML-style polymorphic types. The analysis supports polymorphism over binding times and allows for polymorphic recursion, that is, the binding times of recursive calls of a function may be different at each call. Our binding-time analysis is built as a constraint analysis on top of the region type system. There are several advantages to this approach. First, it allows for a clean separation between the underlying flow analysis and the additional well-formedness requirements, which is a desirable factorization for a BTA [140]. Secondly, the flow analysis part of our BTA is almost as accurate as the powerful data dependency analysis obtained by region inference. Finally, since we can reuse the polynomial time region reconstruction algorithms by Birkedal and Tofte [11, 146], it is only necessary to design and implement a constraint analysis algorithm on top of region inference.

The result of the constraint analysis is a constraint-annotated region expression, where constraints express binding-time dependencies between regions. Since region reconstruction with polymorphic recursion at the level of region variables is decidable[4], our approach implements a binding-time analysis with a high degree of binding-time polyvariance while guaranteeing the finiteness of binding-time descriptions [22]. Our technique builds on a polymorphically typed language, thus extending work by others that builds on the simply typed lambda calculus [36, 73].

Specialization is described with a small-step operational semantics. We again use syntactic type soundness techniques [57, 158] to show the constraint analysis sound with respect to the specialization semantics. Additionally, the specialization semantics is proven correct with respect to the standard semantics of the region calculus. We do so by showing that each reduction step taken by the specializer is in fact a contextual equivalence in the region calculus.

These results are submitted for publication [69].

1.3 Overview

The core of this thesis builds on variations of the Tofte-Talpin region calculus, which we will henceforth refer to as the λ_{tt}^{region}-calculus. These variations are introduced for technical and presentation reasons and we list them below in Section 1.3.1 together with their occurrence in the different chapters. Additionally, we give an overview of their respective type systems in Section 1.3.2 and reduction relations in Section 1.3.3. The introductory chapter concludes with a road map for the rest of the thesis.

[4] In contrast, type reconstruction with polymorphic recursion at the level of types is undecidable [71].

1.3.1 Region Calculi

A region calculus is uniquely determined by its syntax and dynamic semantics. Only the λ_{tt}^{region}-calculus is defined with a big-step semantics, all the other calculi have a small-step semantics.

The λ_{tt}^{region}-calculus of Tofte and Talpin is the original calculus by Tofte and Talpin and presented in Section 2.3.

The monomorphic $\lambda_{tt\text{-}m}^{region}$-calculus is a monomorphic subset of the λ_{tt}^{region}-calculus.

The storeless λ_{s}^{region}-calculus is a storeless variation of the λ_{tt}^{region}-calculus and is presented in Section 3.2.

The monomorphic storeless $\lambda_{s\text{-}m}^{region}$-calculus is a monomorphic subset of the λ_{s}^{region}-calculus.

The imperative λ_{i}^{region}-calculus is a variation of the λ_{s}^{region}-calculus with an explicit store. It is presented in Section 4.2.

The pure λ_{pi}^{region}-calculus is a pure subset of the λ_{i}^{region}-calculus.

The recursive λ_{f}^{region}-calculus is a variation of the λ_{s}^{region}-calculus with first-class recursive functions and is presented in Section 5.2.

The constraint-annotated λ_{C}^{region}-calculus extends the λ_{s}^{region}-calculus with constraint annotations. It is presented in Section 6.2.1.

The two-level λ_{2}^{region}-calculus extends the λ_{C}^{region}-calculus with two-level terms and is presented in Section 6.3.1.

Each of the above region calculi models a different set of constructs as listed in Figure 1.1. We briefly describe the different row entries:

- The *lambda calculus* constructs are the usual ones: variables, lambda abstractions and function applications.
- *Constants* are unspecified values of base type.
- A *canonical primitive* is the identity on constants and models a primitive function.
- *Region abstractions* allocate new memory. Details are discussed in Section 2.3.
- A `let`*-binding* is the usual local binder. However, except for the λ_{f}^{region}-calculus, it is always monomorphic.
- A `letrec`*-binding* binds recursive type- and region polymorphic functions. Next to normal variables, these functions also abstract over region variables.

	λ_{tt}^{region}	$\lambda_{tt\text{-}m}^{region}$	λ_{s}^{region}	$\lambda_{s\text{-}m}^{region}$	λ_{i}^{region}	λ_{pi}^{region}	λ_{f}^{region}	λ_{C}^{region}	λ_{2}^{region}
lambda calculus	X	X	X	X	X	X	X	X	X
constants	X	X	X	X	X	X		X	X
canonical primitive	X		X		X			X	X
region abstractions	X	X	X	X	X	X	X	X	X
`let`*-binding*	X		X				X	X	X
`letrec`*-binding*	X		X					X	X
recursive functions							X		
region applications	X		X				X	X	X
reference cells					X				
constraint sets								X	X
two-level terms									X

Figure 1.1: Region Calculi

- As with `letrec`-bound functions, *recursive functions* abstract over regions. In addition, they are first-class objects in the language and together with a Hindley-Milner polymorphic `let`-construct, they subsume `letrec`-bound functions.
- *Region applications* make it possible to substitute abstracted region variables for actual regions.
- *Reference cells* model ML-style operations on reference cells.
- A *constraint set* collects binding-time constraints at region abstractions. See Section 6.2 for details.
- A *two-level expression* hardwires binding times in the syntax and is required for specialization. See Section 6.3 for details.

1.3.2 Type Systems

Each of the region calculi from Section 1.3.1 has a static semantics, expressed by a type system of type judgments. We distinguish different type judgments by their annotations:

The $\vdash_t^t$ typing judgment types the λ_{tt}^{region}-calculus, the $\lambda_{tt\text{-}m}^{region}$-calculus, the λ_{s}^{region}-calculus and the $\lambda_{s\text{-}m}^{region}$-calculus. It is defined in Figure 2.6 on page 27 and extended in Figure 3.3 on page 33.

The $\vdash_t$ typing judgment types the same region terms as the $\vdash_t^t$ typing judgment. The only difference is that it has a slightly more conservative type rule for lambda abstractions, which is given on page 118.

The $\vdash_i$ typing judgment types the λ_{i}^{region}-calculus and the λ_{pi}^{region}-calculus. It uses the auxiliary judgments $\vdash_i^h$, $\vdash_i^s$ and $\vdash_i^e$. The $\vdash_i$ typing judgment is defined by rule

(I-Conf) on page 49 and rule *(I-Heap)* on page 49. Auxiliary type rules are found in Figure 4.3 on page 48 and Figure 4.4 on page 49.

The $\vdash_f$ typing judgment types the λ_f^{region}-calculus. It is defined in Figure 5.4 on page 75.

The $\vdash_c$ typing judgment types the λ_C^{region}-calculus and the λ_2^{region}-calculus. It is defined in Figure 6.2 on page 115 and extended in Figure 6.8 on page 129 and Figure 6.9 on page 131.

1.3.3 Reduction Relations

Each of the region calculi also has a dynamic reduction relation:

Relation $\Downarrow$ specifies a big-step evaluation semantics for the λ_{tt}^{region}-calculus and the $\lambda_{tt\text{-}m}^{region}$-calculus. It is defined in Figure 2.5 on page 24.

Relation $\rightarrow$ specifies a small-step transition semantics for the λ_s^{region}-calculus and the $\lambda_{s\text{-}m}^{region}$-calculus. It is defined in Figure 3.2 on page 31.

Relation $\rightarrowtail$ specifies a small-step transition semantics for the λ_i^{region}-calculus and the λ_{pi}^{region}-calculus. It is defined in Figure 4.2 on page 46.

Relation $\rightharpoonup$ specifies a small-step transition semantics for the λ_f^{region}-calculus. It is defined in Figure 5.2 on page 73 and Figure 5.3 on page 74.

Relation $\rightsquigarrow$ specifies a small-step transition semantics for the λ_2^{region}-calculus. It is defined in Figure 6.5 on page 123, Figure 6.6 on page 124 and Figure 6.7 on page 125.

1.3.4 Chapter Outline

The rest of this thesis is structured as follows:

Chapter 2 presents preliminary technical background, which is used to develop the operational theory of polymorphic offline partial evaluation. It introduces some notational conventions (Section 2.1), discusses operational techniques in semantics (Section 2.2), and defines the static and dynamic semantics of the λ_{tt}^{region}-calculus (Section 2.3).

Chapter 3 formulates a syntactic type soundness proof for the λ_s^{region}-calculus. It discusses syntax (Section 3.2.1), dynamic semantics (Section 3.2.2), static semantics (Section 3.2.3) and illustrates small-step evaluation with an example (Section 3.2.4). After formulating auxiliary lemmas (Section 3.3.1), a type preservation (Section 3.3.2) and progress property (Section 3.3.3) are proven. Finally, type soundness is established (Section 3.3.4).

Chapter 4 introduces the λ_i^{region}-calculus (Section 4.2), which adds an explicitely passed store to the λ_s^{region}-calculus to provide for operations on references. After discussing the dynamic and static semantics (Section 4.2.2 and Section 4.2.3), we prove type soundness (Section 4.3). We need several auxiliary lemmas (Section 4.3.1), a type preservation (Section 4.3.2) and progress (Section 4.3.3) property. Then, we formally relate the λ_{pi}^{region}-calculus with the $\lambda_{tt\text{-}m}^{region}$-calculus (Section 4.4) and the λ_{pi}^{region}-calculus with the $\lambda_{s\text{-}m}^{region}$-calculus (Section 4.5).

Chapter 5 develops an equational theory for the λ_f^{region}-calculus (Section 5.2). After introducing the notions of typed relations and open extensions (Section 5.3), a contextual equivalence relation is defined for the λ_f^{region}-calculus (Section 5.4). Then, the syntactic equational theory is formulated (Section 5.5) and a notion of applicative bisimilarity introduced (Section 5.6). After proving a congruence property for bisimilarity (Section 5.7), we show that the equational theory is a bisimulation (Section 5.8). We conclude by proving that the bisimilarity relation is equivalent to contextual equivalence (Section 5.9).

Chapter 6 deals with polymorphic specialization for ML. It starts out with a detailed description of the binding-time or constraint analysis (Section 6.2), where we discuss the λ_C^{region}-calculus (Section 6.2.1), the constraint system (Section 6.2.2 and Section 6.2.3), some properties (Section 6.2.4) and the notion of constraint completion (Section 6.2.5). To formalize specialization (Section 6.3), we introduce the λ_2^{region}-calculus (Section 6.3.1), a small-step operational semantics (Section 6.3.2), and some examples (Section 6.3.3). The extension of the static semantics (Section 6.3.5) is required to prove the soundness of the constraint analysis (Section 6.4). Finally, we also show soundness of the specializer (Section 6.5).

Chapter 7 The concluding chapter assesses our work (Section 7.1), discusses future work (Section 7.2) and concludes (Section 7.3).

2 FORMAL BACKGROUND

This chapter establishes the formal background for the forthcoming chapters. In Section 2.1, we fix some notational conventions. We continue to elaborate a few techniques in operational semantics in Section 2.2. This section only provides an introduction and may be skipped by the knowledgeable reader. Finally, Section 2.3 presents the syntax and semantics of the λ_{tt}^{region}-calculus. Although, Section 2.3 bares no original results – the material presented is mostly taken from the paper by Tofte and Talpin [150] – it introduces technical material for later chapters. We also fix some notation related to region calculi.

2.1 Notation

Let A and B be sets. We write$A \hookrightarrow B$ for the set of partial functions from A to B. The *domain* of a function $f \in A \hookrightarrow B$ is $Dom\,(f) = \{x \in A \mid \exists y(y \in B \wedge f(x) = y)\}$ and its *range* is $Ran\,(f) = \{y \in B \mid \exists x(x \in A \wedge f(x) = y)\}$.

We write $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ for a function f with $Dom\,(f) = \{x_1, \ldots, x_n\}$ and $\{\,\}$ for the empty function. A partial function $f \in A \hookrightarrow B$ is *total* whenever $Dom\,(f) = A$ and we write $A \to B$ for the set of total functions from A to B.

The extension $f + \{x \mapsto a\}$ of $f \in A \hookrightarrow B$ is a partial function where $(f + \{x \mapsto a\})(x') = f(x')$ if $x' \in Dom\,(f) \setminus \{x\}$ and $(f + \{x \mapsto a\})(x) = a$. The application $(f + \{x \mapsto a\})(x')$ is undefined whenever $x' \notin Dom\,(f) \cup \{x\}$. The function $f|_X \in A \hookrightarrow B$ is the *restriction* of f to X and defined as follows: $f|_X(x) = f(x)$, if $x \in X \cap Dom\,(f)$, and undefined otherwise. The *co-restriction* $f - X$ of a partial function f is defined by $(f - X)(x) = f(x)$ if $x \in Dom\,(f) \setminus X$ and undefined otherwise. We write $f - x$ as an abbreviation for $f - \{x\}$. Given two partial functions f_1 and f_2 such that $Ran\,(f_1) \cup Ran\,(f_2) \subseteq A \hookrightarrow B$. Then the function f_2 *extends* f_1 ($f_1 \leq f_2$) if $Dom\,(f_1) \subseteq Dom\,(f_2)$ and for all $x \in Dom\,(f_1)$ it holds that $Dom\,(f_1(x)) \subseteq Dom\,(f_2(x))$ and $f_2(x)|_{Dom\,(f_1(x))} = f_1(x)$.

For two partial functions f and g, we define the *composition* $f \circ g$ such that $(f \circ g)(x) = f(g(x))$ provided $x \in Dom\,(g)$ and $g(x) \in Dom\,(f)$. It is undefined otherwise.

A finite sequence of symbols a_i is written $\vec{a}$ and the set of the elements in a sequence $\vec{a}$ is denoted by $\{\vec{a}\}$. We write $\mathcal{P}(A)$ for the set of all subsets of A.

We use the symbol $=$ to denote equality of terms modulo α-conversion. If the terms e and e' are syntactically equal, we write $e \equiv e'$. The notation $e[x \mapsto e']$, stands for the term e with e' substituted for each free occurrence of x. As usual, substitution avoids capture of variables by renaming.

Some semantic objects are indexed by the calculus they belong to. For example, the

syntactic category of terms Term exists for each calculus. Whenever we want to avoid ambiguity we may write λ_{tt}^{region}-Term for the terms of the λ_{tt}^{region}-calculus and similarly for the other calculi. When no confusion is possible, we omit the annotation.

2.2 Operational Techniques in Semantics

The main attractiveness of operational techniques in semantics is the relatively small mathematical overhead required to develop the theory. In this section, we explain how an operational theory can be built from a transition semantics and an appropriate notion of bisimilarity. It is largely based on Section 2.1, Section 2.2 and Section 2.4 from a study by Gordon [50]. He discusses a theory of bisimilarity for a call-by-name PCF[1] with lazy streams. In this section, we only consider a simply-typed call-by-value lambda calculus with a recursion operator. All the region calculi coming up later in the thesis extend it in one way or another. Additionally, we discuss the syntactic proof method for type soundness by Wright and Felleisen [158], in a variation pioneered by Harper [57]. Material and more references on operational methods in semantics can be found in the HOOTS book [52] and also in an article by Pitts [117].

2.2.1 Induction and Co-Induction

The foundations for operational reasoning in semantics are given by the dual notions of *induction* and *co-induction*.

For some universal set U, consider the *complete lattice* $(\mathcal{P}(U), \subseteq)$ and the function $f \in \mathcal{P}(U) \to \mathcal{P}(U)$. The function f is *monotone* whenever it preserves the partial order, *i.e.* $X \subseteq Y$ implies $f(X) \subseteq f(Y)$. We say that a set $X \subseteq U$ is *f-closed* if and only if $f(X) \subseteq X$. A set $X \subseteq U$ is *f-dense* if and only if $X \subseteq f(X)$. A *fixpoint* of f is a solution of the equation $X = f(X)$, that is, a set which is f-closed *and* f-dense.

We define the following subsets of U:

$$\textit{(Infimum)} \quad \mathsf{Inf}(f) = \bigcap \{X \mid f(X) \subseteq X\}$$

$$\textit{(Supremum)} \quad \mathsf{Sup}(f) = \bigcup \{X \mid X \subseteq f(X)\}$$

These sets have a special property as formulated by the *Knaster-Tarski*Fixpoint theorem:

Theorem 2.2.1 (Knaster-Tarski) *For a monotone function $f \in \mathcal{P}(U) \to \mathcal{P}(U)$, it holds that*

1. $\mathsf{Inf}(f)$ *is the least fixpoint of f.*

2. $\mathsf{Sup}(f)$ *is the greatest fixpoint of f.*

[1] PCF is a programming model introduced by Dana Scott [127] and stands for *Programming Computable Functions*.

The set $\mathsf{Inf}(f)$ is *inductively* defined by the function f and the set $\mathsf{Sup}(f)$ is *co-inductively* defined by the function f. The Knaster-Tarski theorem implies the following two proof principles:

Induction	$f(X) \subseteq X$ *implies* $\mathsf{Inf}(f) \subseteq X$
Co-induction	$X \subseteq f(X)$ *implies* $X \subseteq \mathsf{Sup}(f)$

Induction comes in many flavors. *Mathematical induction* functions as follows: take an element $0 \in U$ (where U is a universal set) and an injective function $S \in U \to U$. Now, define the monotone function $f(X) = \{0\} \cup \{S(x) \mid x \in X\}$ and take the set of natural numbers $\mathbb{N}$ to be the infimum $\mathsf{Inf}(f)$. Whenever we define a number theoretic property as a set X, we have to show that $\mathbb{N} \subseteq X$, *i.e.* the property must hold for all natural numbers. The induction principle says that it is sufficient to show that $0 \in X$ and that for all $x \in X$, it holds that $S(x) \in X$.

The notion of mathematical induction can be extended to any monotone function f. A typical example is a monotone axiomatic deduction system. Since such a system is defined by a set of rules, the associated inductive proof principles is often called *rule induction*. Operational and type semantics are all defined by rule induction and we use it plenty throughout the thesis. A more rigorous explanation on different induction techniques can be found in books and articles by Aczel [4], Pitts [115], Mitchell [104] and Winskel [157]. In contrast, the material on co-induction is rather scarce.

In our thesis, we use co-inductively defined relations [102] to relate a big-step with a small-step semantics (see Section 4.4) and in the definition of bisimilarity, as explained below in Section 2.2.5.

Both the inductive and co-inductive proof principles also have a *strong* variant, which are slightly more flexible (see Section 2.2 of Gordon's article [50] for proofs).

Strong Induction	$f(X \cap \mathsf{Inf}(f)) \cap \mathsf{Inf}(f) \subseteq X$ *implies* $\mathsf{Inf}(f) \subseteq X$
Strong Co-induction	$X \subseteq f(X \cup \mathsf{Sup}(f)) \cup \mathsf{Sup}(f)$ *implies* $X \subseteq \mathsf{Sup}(f)$

2.2.2 A Simply-Typed Lambda Calculus with Recursion

We explore operational methods by means of a simple example calculus: the simply-typed lambda calculus with a recursion operator, the λ^{cbv}-calculus. Its syntax is given by the following grammar:

$$x, f \in \mathsf{Var} \qquad c \in \mathsf{Constant}$$

$$\mathsf{Type} \ni \tau ::= \mathtt{int} \mid \tau \longrightarrow \tau$$

$$\mathsf{Term} \ni e ::= x \mid c \mid \lambda\, x.\, e \mid e \mathbin{@} e \mid \mathtt{fix}\, f.e$$

The sets Var and $\mathsf{Constant}$ are denumerable disjoint sets of variable names and constant symbols respectively. The type language caters for one base type `int` and a function arrow $\longrightarrow$. The grammar of expressions contains variables, constants, lambda abstractions, function applications, and a recursive binder $\mathtt{fix}\, f.e$ which binds the variable f in e recursively to $\mathtt{fix}\, f.e$.

$$\textit{(S-Var)} \qquad \frac{TE(x) = \tau}{TE \vdash_s x : \tau}$$

$$\textit{(S-Const)} \qquad TE \vdash_s c : \texttt{int}$$

$$\textit{(S-Lam)} \qquad \frac{TE + \{x \mapsto \tau_1\} \vdash_s e : \tau_2}{TE \vdash_s \lambda x.\, e : \tau_1 \longrightarrow \tau_2}$$

$$\textit{(S-App)} \qquad \frac{TE \vdash_s e_1 : \tau_1 \longrightarrow \tau_2 \quad TE \vdash_s e_2 : \tau_1}{TE \vdash_s e_1 \mathbin{@} e_2 : \tau_2}$$

$$\textit{(S-Fix)} \qquad \frac{TE + \{f \mapsto \tau\} \vdash_s e : \tau}{TE \vdash_s \texttt{fix}\, f.e : \tau}$$

Figure 2.1: Static semantics of the λ^{cbv}-calculus

The notion of free variables of an expression e, written $fv\,(e)$, is defined as usual. A *closed term* is an expression e for which $fv\,(e) = \emptyset$. If a term is not closed, we say it is *open*.

Static Semantics of the λ^{cbv}-calculus

A *type environment* $TE \in \mathsf{Var} \hookrightarrow \mathsf{Type}$ is a finite function from variables to types. Type assignment for the λ^{cbv}-calculus is a ternary relation $TE \vdash_s e : \tau$ inductively defined by the rules in Figure 2.1.

The type rules are standard (see for example [104]): rule *(S-Var)* looks up the type of a variable in the type environment. Rule *(S-Const)* assigns type `int` to the constant c. A lambda abstraction is assigned a functional type with rule *(S-Lam)*. It type checks the body by extending the environment with the type of the bound parameter x. Function application is typed with rule *(S-App)* and requires an expression of functional type as the operator. This is the only place where a *type error* can occur: whenever e_1 is of type `int`, the application $e_1 \mathbin{@} e_2$ is not typeable. Finally, the rule *(S-Fix)* types a recursive binding.

Define the following sets:

$$\mathit{Prog}(\tau) = \{e \mid \{\,\} \vdash_s e : \tau\}$$

$$\mathit{BinRel}(\tau) = \{(e_1, e_2) \mid e_1 \in \mathit{Prog}(\tau) \wedge e_2 \in \mathit{Prog}(\tau)\}$$

$$\mathit{BinRel} = \bigcup \{\mathit{BinRel}(\tau) \mid \textit{ for any type } \tau\}$$

A *program* is a well-typed closed term and the set *BinRel* contains all pairs of programs with the same type.

Beta-value reduction:

$$(\lambda\, x.\, e) \mathbin{@} v \mapsto e[x \mapsto v] \tag{2.1}$$

Recursive unfolding:

$$\texttt{fix}\, f.e \mapsto e[f \mapsto \texttt{fix}\, f.e] \tag{2.2}$$

Reductions in Context:

$$\frac{e_1 \mapsto e_1'}{e_1 \mathbin{@} e_2 \mapsto e_1' \mathbin{@} e_2} \tag{2.3}$$

$$\frac{e \mapsto e'}{v \mathbin{@} e \mapsto v \mathbin{@} e'} \tag{2.4}$$

Figure 2.2: Dynamic semantics of the λ^{cbv}-calculus

Dynamic Semantics of the λ^{cbv}-calculus

The small-step transition relation $\mapsto$ is a syntactic rewrite relation defined on programs in λ^{cbv}-Term. Values in the λ^{cbv}-calculus are constants or lambda abstractions, so we introduce an additional syntactic class:

$$\mathsf{Value} \ni v ::= c \mid \lambda\, x.\, e$$

A call-by-value semantics for the λ^{cbv}-calculus is defined by the rules in Figure 2.2. Reduction rule (2.1) implements a beta-value reduction. Recursive definitions are unfolded by rule (2.2). The rules (2.3) and (2.4) define a left-to-right call-by-value semantics for the λ^{cbv}-calculus. The latter two rules are *context rules* and sometimes presented by means of a grammar for *evaluation contexts* [38]. In this thesis, we follow the more direct Plotkin-style formulation [122].

The reduction relation $\mapsto$ is deterministic:

Lemma 2.2.2 *If $e \mapsto e'$ and $e \mapsto e''$, then $e' \equiv e''$.*

The relation $\overset{*}{\mapsto}$ is the *reflexive* and *transitive closure* of $\mapsto$. We say that e *converges* ($e \searrow$) whenever there exists a value v such that $e \overset{*}{\mapsto} v$. If for all e' where $e \overset{*}{\mapsto} e'$, there exists an e'' such that $e' \mapsto e''$, then we say that e *diverges* ($e \nearrow$).

2.2.3 Syntactic Type Soundness proofs

A static type system is *sound* with respect to a dynamic semantics if well-typed programs cannot cause type errors. The literature is abundant with material on type soundness. We follow the syntactic approach by Wright and Felleisen [158], although we present it in a variation due to Harper [57]. The paper by Wright and Felleisen gives a good overview of alternative operational and denotational methods.

Syntactic type soundness requires three main steps. We omit the proofs for the λ^{cbv}-calculus as they are standard.

Substitution Lemma For every substitution in the rewrite rules, we formulate a *substitution lemma*. In the case of the λ^{cbv}-calculus, we need:

Lemma 2.2.3 *If* $TE \vdash_s e' : \tau'$ *and* $TE + \{x \mapsto \tau'\} \vdash_s e : \tau$, *then* $TE \vdash_s e[x \mapsto e'] : \tau$.

Type Preservation states that type assignment is invariant under reduction. For the λ^{cbv}-calculus it is as follows:

Proposition 2.2.4 *If* $TE \vdash_s e : \tau$ *and* $e \mapsto e'$, *then* $TE \vdash_s e' : \tau$.

The proposition is proven by induction and making use of the substitution lemma.

The type preservation property is sometimes known as *subject reduction* after its analog in combinatory logic [30]. However, since we only show type preservation for the transition relation $\mapsto$, it is slightly weaker than subject reduction. The latter implies type preservation for a compatible extension of the reduction relation, *i.e.* reduction preserves typing in arbitrary contexts, not only in evaluation contexts.

Progress implies that it is always possible to reduce a program unless it is a value. For the λ^{cbv}-calculus, progress is as follows:

Proposition 2.2.5 *If* $\{\,\} \vdash_s e : \tau$, *then either* e *is a value* v *or* $e \mapsto e'$.

Progress is usually proven with an auxiliary result, known as the *canonical forms lemma*. It describes the form of a well-typed value:

Lemma 2.2.6 *Suppose* $TE \vdash_s v : \tau$. *Whenever* $\tau = \texttt{int}$, *then* v *is a constant of the form* c. *Whenever* $\tau = \tau_1 \longrightarrow \tau_2$, *then* v *is a lambda abstraction of the form* $\lambda\, x.\, e$.

Combining type preservation and progress leads to a (*strong* [158]) type soundness theorem:

Theorem 2.2.7 *Suppose* $\{\,\} \vdash_s e : \tau$. *Then either* $e \overset{*}{\mapsto} v$ *such that* $\{\,\} \vdash_s v : \tau$ *or* $e \not\nearrow$.

The proof strategy as presented here is the variation due to Harper [57]. Wright and Felleisen [158] do not prove a progress property, but consider the contraposition. This requires the introduction of *stuck* expressions: an expression e is stuck if e is not a value v and there is no e' for which $e \to e'$. Then, we approximate the set of expressions that become stuck by a countable set of *faulty expressions*, for which the following can be shown: if a closed expression cannot be reduced to a faulty expression, evaluation either does not terminate or returns a value. If faulty expressions coincide with untypable expressions, type soundness follows by type preservation.

2.2.4 Contextual Equivalence

Contextual equivalence is a powerful notion of program equivalence: intuitively, two program fragments are contextual equivalent if no program context can tell them apart. The notion is due to Morris [108], who specified it for a compatible extension of the lambda calculus reduction semantics. Plotkin [121] simplified contextual equivalence to weak-head reduction. It has become a standard notion to reason about operational program equivalence. Contextual equivalence is sometimes known as *observational* or *operational equivalence* (the term *observational congruence* is used as well [98], but not as common).

We introduce a context as an expression with one hole, where we take the hole to be a distinguished variable, say $\diamond$.

Definition 2.2.1 *A* context[2] *is an expression e such that $fv\,(e) \subseteq \{\diamond\}$. If e is a context, we write $e[e']$ for $e[\diamond \mapsto e']$.*

The use of a substitution instead of *hole filling with capture* does not pose any problems for our simplified presentation as the definition of contextual equivalence below only substitutes closed expressions.

Given the dynamic semantics of Figure 2.2, here is the definition of contextual equivalence for the λ^{cbv}-calculus:

Definition 2.2.2 Contextual order, $\sqsubseteq \subseteq$ *BinRel, and* contextual equivalence, $\backsim \subseteq$ *BinRel, are as follows:*

- $e_1 \sqsubseteq e_2$ *iff whenever* $\{\,\} \vdash_s e_1 : \tau$ *and* $\{\,\} \vdash_s e_2 : \tau$ *and for some context e such that* $\{\diamond \mapsto \tau\} \vdash_s e : \mathtt{int}$, *the convergence* $e[e_1] \searrow$ *implies the convergence* $e[e_2] \searrow$.
- $e_1 \backsim e_2$ *iff* $e_1 \sqsubseteq e_2$ *and* $e_2 \sqsubseteq e_1$.

So, we basically say that two programs are not equivalent when we have a context that distinguishes them. The definition is formulated in terms of termination behavior. It is possible to reformulate contextual equivalence in terms of convergence towards a specific constant. However, since convergence holds for *all* contexts, it is easy to see that a definition with convergence to a constant is equivalent to the one above. For example, if a context $e[e_1]$ reduces to a constant c, we only have to extend the context e such that it converges whenever $e[e_1]$ reduces to c and that it diverges otherwise[3].

2.2.5 Bisimilarity

Bisimulation is a notion originating from process calculi, notably Milner's CCS [101]. In many cases, it allows an alternative co-inductive characterization for contextual equivalence of functional programming languages [3, 48–50]. Here, we formulate a notion of

[2]This is not the most general definition of context [38], but sufficient for the simplified presentation of this section. In Section 5.4, we use a more refined notion of contexts for a region calculus [87].

[3]To actually do this, we need a consumer function for constants, which is currently not present in the λ^{cbv}-calculus. However, it is trivial to add such a consumer and we omit the details.

$$\textit{(Obs-Red-cbv)} \qquad \frac{\{\,\} \vdash_s e : \tau \quad e \mapsto e'' \quad e'' \xrightarrow{\delta} e'}{e \xrightarrow{\delta} e'}$$

$$\textit{(Obs-Lam-cbv)} \qquad \frac{\{\,\} \vdash_s \lambda x.\, e : \tau_1 \longrightarrow \tau_2 \quad \{\,\} \vdash_s v : \tau_1}{\lambda x.\, e \xrightarrow{@\, v} e[x \mapsto v]}$$

$$\textit{(Obs-Const-cbv)} \qquad c \xrightarrow{c} \texttt{fix}\, x.x$$

Figure 2.3: A labeled transition system for the λ^{cbv}-calculus

bisimilarity for the λ^{cbv}-calculus, but we omit theorems and proofs. In Chapter 5, we introduce bisimilarity for a region calculus and prove it equivalent to contextual equivalence. There, we also use it to show that an equational theory for a region calculus is sound with respect to contextual equivalence.

A Labeled Transition System

We describe bisimilarity with a *labeled transition system* since we use a small-step transition semantics for the dynamic semantics. We also follow this approach in Chapter 5, however, this is largely a matter of taste. A bisimilarity theory can be developed without such a transition system, see for example [49].

The idea is to characterize the observations one can make of a program. In the case of the λ^{cbv}-calculus, we only have two observations:

$$\mathsf{Observation} \ni \delta ::= @\, v \mid c$$

We explain the observations below. Define a labeled transition system ($\dot{\longrightarrow} \in \mathsf{Term} \times \mathsf{Observation} \times \mathsf{Term}$) as a ternary relation inductively defined by the rules in Figure 2.3. Rule *(Obs-Red-cbv)* expresses that in a call-by-value language, an expression is essentially characterized by its value, whenever it exists. Rule *(Obs-Lam-cbv)* observes a lambda abstraction and unfolds it with some value. A constant in the λ^{cbv}-calculus is computationally not interesting. Hence, rule *(Obs-Const-cbv)* makes a transition into $\texttt{fix}\, x.x$ which does not terminate and therefore yields no new observations (by rule *(Obs-Red-cbv)*).

The *derivation tree* of a program is the tree whose nodes are programs and whose arcs are labeled transitions. Intuitively, we expect two programs to be equivalent whenever their derivations trees have the same structure and labels, except for the syntax in the nodes.

Simulation and Bisimulation

A notion of program equivalence based on derivation trees can be formalized by considering the following two monotone functions $[\cdot] \in \mathcal{P}(\mathit{BinRel}) \to \mathcal{P}(\mathit{BinRel})$ and $\langle\cdot\rangle \in$

$\mathcal{P}(BinRel) \to \mathcal{P}(BinRel)$, where we have for a relation $\mathcal{R} \in BinRel$ that $\mathcal{R}^{-1}$ defines its inverse.

$$[\mathcal{R}] = \{(e_1, e_2) \mid \textit{whenever } e_1 \xrightarrow{\delta} e_1', \textit{ then there is an } e_2' \textit{ such that } e_2 \xrightarrow{\delta} e_2' \textit{ and } (e_1', e_2') \in \mathcal{R}\}$$

$$\langle \mathcal{R} \rangle = [\mathcal{R}] \cap [\mathcal{R}^{-1}]^{-1}$$

These monotone functions *build* pre-ordered and equivalent derivation trees respectively. What interests us are the fixpoints of these functions. However, there are many of these, for example, the empty relation. Following Milner [101], we are after the greatest fixpoints:

- A *simulation* is a $[\cdot]$-dense relation.
- A *bisimulation* is a $\langle\cdot\rangle$-dense relation.
- *Similarity*, $\prec \in BinRel$, is the greatest simulation, *i.e.* $\prec = \mathsf{Sup}([\cdot])$.
- *Bisimilarity*, $\sim \in BinRel$, is the greatest bisimulation, *i.e.* $\sim = \mathsf{Sup}(\langle\cdot\rangle)$.

From the Knaster-Tarski fixpoint Theorem 2.2.1, we have that $\prec = [\prec]$ and $\sim = \langle\sim\rangle$. Additionally, it allows us to use the co-inductive proof principles from Section 2.2.1 to prove similarity and bisimilarity equivalences.

A theory of bisimilarity as depicted above is only useful if it is at least sound with respect to contextual equivalence. Preferably it is also shown complete with respect to contextual equivalence in order to make it an alternative method for reasoning about program equivalence.

Additionally, the usefulness of bisimilarity very much depends on the design of the labeled transition system. The key issue is to figure out what observations and transitions are computationally relevant, *i.e.* what observations could influence the termination behavior of a program. For example, one could include labeled transitions for all context rules: this would lead to an alternative characterization of Milner's Context Lemma [99] and is called *experimental equivalence* by Gordon [48]. On the other extreme, omitting relevant observations can lead to an incomplete notion of bisimilarity, see for instance [53]. A definition of bisimilarity that only considers applicative contexts, for instance the one for the λ^{cbv}-calculus, is called *applicative bisimilarity* [2, 3].

Finally, we stress that the exposition on operational equivalence and bisimilarity in this section is kept to a minimum. In particular, we have glossed over important details such as how to deal with open expressions. These details are thoroughly handled in Chapter 5, where we develop a bisimilarity theory for a variation of the region calculus.

2.3 The λ_{tt}^{region}-calculus: its syntax and semantics

All the region calculi in this thesis are variations of the λ_{tt}^{region}-calculus as it was originally specified by Tofte and Talpin [149, 150]. In this section, we present its syntax and semantics.

$$\mathsf{Term} \ni e ::= x \mid c\ \mathtt{at}\ \varrho \mid \lambda x.\, e\ \mathtt{at}\ \varrho \mid e\ @\ e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{copy}\,[\varrho_1, \varrho_2]\, e \mid \mathtt{new}\ \varrho.\, e \mid f\ [\varrho_1, \ldots, \varrho_n]\ \mathtt{at}\ \varrho \mid \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e\ \mathtt{at}\ \varrho\ \mathtt{in}\ e$$

Figure 2.4: Syntax of the λ_{tt}^{region}-calculus

2.3.1 Syntax of the λ_{tt}^{region}-calculus

The syntax of the λ_{tt}^{region}-calculus is defined in the grammar of Figure 2.4. We take $\varrho \in \mathsf{RegionVar}$, a denumerable and disjoint set with respect to Var and Constant (see Section 2.2.2).

In comparison to the lambda calculus, constants and lambda abstractions carry a region annotation $\mathtt{at}\ \varrho$, which indicates the region in which the value is to be allocated. Since constants also carry this annotation, the λ_{tt}^{region}-calculus formalizes a fully boxed implementation strategy. The lambda abstraction $\lambda x.\, e\ \mathtt{at}\ \varrho$ yields a lambda closure which is allocated in the region bound to ϱ. Variables, function application and `let`-bindings are standard. The term $\mathtt{copy}\,[\varrho_1, \varrho_2]\, e$ copies a value of base type from the region bound to ϱ_1 to the region bound to ϱ_2. It stands for a prototypical primitive operation.

Memory is divided in *regions* of unbounded size, which are allocated and deallocated in a stack-like manner. The term $\mathtt{new}\ \varrho.\, e$ allocates a new region, binds it to ϱ, evaluates e, and finally deallocates the region. Hence, the lifetime of a region corresponds with the lexical scope of ϱ. The set of free region variables for a λ_{tt}^{region}-Term is standard and denoted by $frv\,(e)$. Bound occurrences of ϱ in e are subject to α-conversion as usual.

The term $\mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1\ \mathtt{at}\ \varrho\ \mathtt{in}\ e_2$ recursively binds the variable f to the closure of a region abstraction $\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1$, which is put into the region bound to ϱ. A region abstraction is a lambda abstraction, which is abstracted over zero or more region variables. The region application $f\ [\varrho_1, \ldots, \varrho_n]\ \mathtt{at}\ \varrho$ binds its region arguments to the formal region parameters of the closure of the region abstraction bound to f. The resulting lambda closure is put in the region bound to ϱ.

Like in the λ^{cbv}-calculus, a *program* is a term without free (expression) variables, but it may contain free occurrences of region variables. In our framework, programs typically have free region variables, for instance, the initial memory region in which the program stores its result.

2.3.2 Dynamic Semantics of the λ_{tt}^{region}-calculus

The dynamic semantics of the λ_{tt}^{region}-calculus is formulated as a big-step operational evaluation relation. We need several semantic objects.

First, assume the following two denumerable sets, both disjoint with respect to Var, Constant and RegionVar:

$r \in \mathsf{RegionName}$	region names
$l \in \mathsf{Location}$	store locations

A *value* $v \in \mathsf{Value}$ is a pair of a region name r and a location l, *i.e.* an *address*. A location can be understood as an offset in the potentially unbounded region referred to by r.

A *region environment*, $RE \in \mathsf{RegionVar} \hookrightarrow \mathsf{RegionName}$, is a finite function from region variables to region names. A *value environment*, $VE \in \mathsf{Var} \hookrightarrow \mathsf{Value}$, is a finite function from variables to values. A *storable value*, $w \in \mathsf{Storable}$, is either a stored constant $\langle c \rangle$, a function closure $\langle x, e, VE, RE \rangle$ or a region function closure $\langle \varrho_1, \ldots, \varrho_n, x, e, VE, RE \rangle$. A function closure $\langle x, e, VE, RE \rangle$ consists of a formal parameter x, the body of the stored lambda e, a value environment VE, and a region environment RE. Storing the region and value environment implements lexical binding. A region function closure $\langle \varrho_1, \ldots, \varrho_n, x, e, VE, RE \rangle$ is very much like a function closure, except that it registers the formal region parameters $\varrho_1, \ldots, \varrho_n$ as well. A *region*, $p \in \mathsf{Location} \hookrightarrow \mathsf{Storable}$, is a finite function from locations to storable values. A *store*, $S \in \mathsf{RegionName} \hookrightarrow (\mathsf{Location} \hookrightarrow \mathsf{Storable})$, is a finite function from region names to regions.

With these definitions, the big-step evaluation relation $S, VE, RE \vdash e \Downarrow v, S'$ relates the expression e, using the store S, the value environment VE, and the region environment RE, to its value v and the resulting store S'. Figure 2.5 shows the individual rules. Rule *(BS-Var)* evaluates a variable by looking it up in the value environment. A constant c `at` ϱ is allocated with rule *(BS-Const)* by looking up the region bound to ϱ in the region environment and fetching a *fresh* location l. Freshness is guaranteed by demanding that $l \notin Dom\,(S(r))$. The resulting store is extended accordingly. Rule *(BS-Lam)* functions similarly to *(BS-Const)*, except that it allocates a function closure. For function application, rule *(BS-App)* first evaluates the operator to a function closure, somewhere allocated in the store. Then, it evaluates the operand and evaluates the body of the lambda abstraction after binding the operand value to the formal parameter x. Note that the big-step semantics also enforces a left-to-right evaluation, since the evaluation of the operand needs the resulting store of the evaluation of the operator to be in its context. The evaluation of a `let`-expression is expressed with rule *(BS-Let)* and standard. Copying a value from one region r to another region r' is specified with rule *(BS-Copy)*. A new memory cell is allocated in the target region r' by having $l' \notin Dom\,(S'(r'))$ and the constant stored in region r is now allocated at the address (r', l').

A region abstraction `new` $\varrho.\, e$ allocates a new region and is evaluated with rule *(BS-Reglam)*. It takes a fresh region r from the store S and binds it to the abstracted region variable ϱ. The new region is initially the empty map. Rule *(BS-Letrec)* allocates a region function closure very much like constants and lambdas in the rules *(BS-Const)* and *(BS-Lam)* respectively. Note the recursive binding of the variable f in the value environment VE', which is also put in the region function closure. Finally, rule *(BS-Regapp)* evaluates $f\ [\varrho_1, \ldots, \varrho_n]$ `at` ϱ by looking up the region function closure bound to f in the value environment. It binds the regions bound to the region variables $\varrho_1, \ldots, \varrho_n$ to the formal region parameters in the closure and allocates the resulting function closure to the region bound to ϱ.

The final store of an evaluation extends the store it starts out with, but without altering the domain. The domain is invariant because every region allocation is paired with a region deallocation. Of course, the initial regions in S may have grown.

(BS-Var)
$$\frac{VE(x) = v}{S,\, VE,\, RE \vdash x \Downarrow v, S}$$

(BS-Const)
$$\frac{RE(\varrho) = r \quad l \notin Dom\,(S(r)) \quad w = \langle c \rangle}{S,\, VE,\, RE \vdash c\ \mathtt{at}\ \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto w\}\}}$$

(BS-Lam)
$$\frac{RE(\varrho) = r \quad l \notin Dom\,(S(r)) \quad w = \langle x, e, VE, RE \rangle}{S,\, VE,\, RE \vdash \lambda\, x.\, e\ \mathtt{at}\ \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto w\}\}}$$

(BS-App)
$$\frac{\begin{array}{c} S,\, VE,\, RE \vdash e_1 \Downarrow (r_1, l_1), S_1 \quad S_1(r_1)(l_1) = \langle x, e, VE', RE' \rangle \\ S_1,\, VE,\, RE \vdash e_2 \Downarrow v_2, S_2 \quad S_2,\, VE' + \{x \mapsto v_2\},\, RE' \vdash e \Downarrow v, S' \end{array}}{S,\, VE,\, RE \vdash e_1 \mathbin{@} e_2 \Downarrow v, S'}$$

(BS-Let)
$$\frac{S,\, VE,\, RE \vdash e_1 \Downarrow v_1, S_1 \quad S_1,\, VE + \{x \mapsto v_1\},\, RE \vdash e_2 \Downarrow v_2, S_2}{S,\, VE,\, RE \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow v_2, S_2}$$

(BS-Copy)
$$\frac{\begin{array}{c} S,\, VE,\, RE \vdash e \Downarrow (r, l), S' \quad S'(r)(l) = \langle c \rangle \\ RE(\varrho) = r \quad RE(\varrho') = r' \quad l' \notin Dom\,(S'(r')) \end{array}}{S,\, VE,\, RE \vdash \mathtt{copy}\,[\varrho, \varrho']\, e \Downarrow (r', l'), S' + \{r' \mapsto S'(r') + \{l' \mapsto \langle c \rangle\}\}}$$

(BS-Reglam)
$$\frac{r \notin Dom\,(S) \quad S + \{r \mapsto \{\,\}\},\, VE,\, RE + \{\varrho \mapsto r\} \vdash e \Downarrow v, S'}{S,\, VE,\, RE \vdash \mathtt{new}\ \varrho.\, e \Downarrow v, S' - r}$$

(BS-Letrec)
$$\frac{\begin{array}{c} r = RE(\varrho) \quad l \notin Dom\,(S(r)) \quad VE' = VE + \{f \mapsto (r, l)\} \\ S + \{(r, l) \mapsto \langle \varrho_1, \ldots, \varrho_n, x, e_1, VE', RE \rangle\},\, VE',\, RE \vdash e_2 \Downarrow v, S' \end{array}}{S,\, VE,\, RE \vdash \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1\ \mathtt{at}\ \varrho\ \mathtt{in}\ e_2 \Downarrow v, S'}$$

(BS-Regapp)
$$\frac{\begin{array}{c} r = RE(\varrho) \quad l \notin Dom\,(S(r)) \\ VE(f) = (r', l') \quad S(r')(l') = \langle \varrho'_1, \ldots, \varrho'_n, x, e, VE', RE' \rangle \\ w = \langle x, e, VE', RE' + \{\varrho'_1 \mapsto RE(\varrho_1), \ldots, \varrho'_n \mapsto RE(\varrho_n)\} \rangle \end{array}}{S,\, VE,\, RE \vdash f\ [\varrho_1, \ldots, \varrho_n]\ \mathtt{at}\ \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto w\}\}}$$

Figure 2.5: Dynamic semantics of the λ_{tt}^{region}-calculus

Lemma 2.3.1 *$S_1, VE, RE \vdash e \Downarrow v, S_2$ implies $S_1 \leq S_2$ and $Dom(S_1) = Dom(S_2)$*

This lemma coincides with Lemma 4.1. in [150].

2.3.3 Static semantics of the λ_{tt}^{region}-calculus

For notational convenience, we introduce a meta-symbol: *the region placeholder* $\rho \in \mathsf{Placeholder}$. It ranges over region variables and perhaps other semantic objects, depending on the calculus of discourse. A placeholder is not a semantic object itself, but a notational convenience which allows us to "reuse" type rules. In the static semantics of the λ_{tt}^{region}-calculus, region placeholders only range over region variables. We stress that a *region*, *region variable* and *region placeholder* are three different things. However, to maintain readability and whenever no confusion is possible, we may use those three terms interchangeably.

Semantics Objects

The type language of the λ_{tt}^{region}-calculus is generated by the following grammar, where $\mathsf{EffectVar}$ and $\mathsf{TypeVar}$ are disjoint denumerable sets of effect and type variables respectively:

$$\epsilon \in \mathsf{EffectVar} \qquad \varphi \subseteq \mathsf{Effect} = \mathsf{Placeholder} \cup \mathsf{EffectVar} \qquad \alpha \in \mathsf{TypeVar}$$

$$\mathsf{Type} \ni \tau ::= \alpha \mid \texttt{int} \mid \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$$

$$\mathsf{TypeWPlace} \ni \mu ::= (\tau, \rho) \qquad \mathsf{TypeScheme} \ni \sigma ::= \forall \vec{\alpha}.\forall \vec{\epsilon}.\forall \vec{\varrho}.\tau$$

An *effect*, φ, is a finite set of region placeholders ρ and effect variables ϵ. The effect of a term, e, indicates the set of regions that may be affected when evaluating e. Tofte and Talpin distinguish between *get(ϱ)* and *put(ϱ)* effects to denote whether an object is read from a region or written into a region, respectively. This qualification is irrelevant for the purposes of this thesis and can easily be added if desired.

A *type* τ is either a type variable α, the integer type `int`, or an arrow type $\pi_1 \xrightarrow{\epsilon.\varphi} \pi_2$. The *arrow effect* $\epsilon.\varphi$ of a function type describes the effect of the function's body and is discharged on function application. Such arrow effects are advantageous for type and region reconstruction. The effect variable ϵ stands for the arrow effect of the function. It implements a mechanism for parametric polymorphism over arrow effects, particularly useful in the presence of higher-order functions [147]. A *type with place* μ is a pair of a type τ and a region placeholder ρ, where the placeholder indicates in what region the object is allocated. A *type scheme* σ abstracts a type τ over type, effect and region variables.

A substitution, $S_s = (S_t, S_e, S_r)$, is a triple of functions: a type substitution $S_t \in \mathsf{TypeVar} \to \mathsf{Type}$, an effect substitution $S_e \in \mathsf{EffectVar} \to \mathsf{EffectVar} \times \mathsf{Effect}$, and a region substitution $S_r \in \mathsf{RegionVar} \to \mathsf{Placeholder}$. The substitution S_s distributes operations componentwise over its three substitutions. The definition of S_s on types, types with

place, placeholders and effects is as follows:

$$S_s(\varphi) = \{S_s(\rho) \mid \rho \in \varphi\} \cup \{\eta \mid \epsilon \in \varphi \wedge S_e(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\}$$

$$S_s(\mu \xrightarrow{\epsilon.\varphi} \mu') = S_s(\mu) \xrightarrow{\epsilon'.\varphi'} S_s(\mu') \quad \textit{where } S_e(\epsilon) = \epsilon'.\varphi'' \textit{ and } \varphi' = \varphi'' \cup S_s(\varphi)$$

$$S_s(\alpha) = S_t(\alpha) \quad S_s(\texttt{int}) = \texttt{int} \quad S_s(\varrho) = S_r(\varrho) \quad S_s(\tau, \rho) = (S_s(\tau), S_s(\rho))$$

$$S_s(\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\forall\, \vec{\varrho}\,.\tau) = \forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\forall\, \vec{\varrho}\,.(S_s - (\{\vec{\alpha}\} \cup \{\vec{\epsilon}\} \cup \{\vec{\varrho}\}))(\tau)$$

The substitutions I_t and I_r are the identities on type and region variables, respectively, and the substitution I_e maps every effect variable ϵ to $\epsilon.\{\}$. We write S_r as a shorthand for (I_t, I_e, S_r) and similarly for the other substitutions. The *support* of a type substitution, $\mathit{Supp}\,(S_t)$, is the set $\{\alpha \mid S_t(\alpha) \neq \alpha\}$. By abuse of notation, we sometimes write $\{\alpha_1 \mapsto S_t(\alpha_1), \ldots, \alpha_n \mapsto S_t(\alpha_n)\}$ whenever the $\mathit{Supp}\,(S_t) = \{\alpha_1, \ldots, \alpha_n\}$. The $\mathit{Supp}\,(S_r)$, $\mathit{Supp}\,(S_e)$ and $\mathit{Supp}\,(S_s)$ are defined analogously. Similarly, we define the *image* of a type substitution S_t by $\mathit{Img}\,(S_t) = \{S_t(\alpha) \mid \alpha \in \mathit{Supp}\,(S_t)\}$ and analogously for S_e, S_r and S_s.

Since region placeholders may occur in expression syntax, we define the substitution $S_s(e) = S_r(e)$, where the region substitution propagates homomorphically in the expression, ignoring locally bound region variables as usual. For a semantic object O (and similarly for a λ_{tt}^{region}-Term e), we may write $O[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n]$ for the region substitution $\{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}(O)$ in analogy to the notation for expression substitution.

A type τ is an *instance* of a type scheme $\sigma = \forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\forall\, \vec{\varrho}\,.\tau'$ via substitution S_s, written $\tau \prec \sigma$ *via* S_s, if $\mathit{Supp}\,(S_t) \subseteq \{\vec{\alpha}\}$, $\mathit{Supp}\,(S_e) \subseteq \{\vec{\epsilon}\}$ and $\mathit{Supp}\,(S_r) \subseteq \{\vec{\varrho}\}$, such that $S_s(\tau') = \tau$. The instance relation extends to type schemes by $\sigma \prec \sigma'$ iff $\tau \prec \sigma$ *via* S_s implies $\tau \prec \sigma'$ *via* S'_s.

A type environment, $TE \in \mathsf{TypeEnv}$, is similar to the type environment in the λ^{cbv}-calculus. It is a finite function that maps expression variables to pairs of the form (σ, ρ). Substitutions extend to type environments, avoiding capture by renaming. The free variables of a semantic object O are defined in the usual way: we write $\mathit{ftv}\,(O)$, $\mathit{frv}\,(O)$, and $\mathit{fev}\,(O)$ for O's free type, region[4] and effect variables respectively. Also, $\mathit{fv}\,(O) = \mathit{ftv}\,(O) \cup \mathit{frv}\,(O) \cup \mathit{fev}\,(O)$. Moreover, $\mathit{frv}\,(TE) = \bigcup \{\mathit{frv}\,(TE(x)) \mid x \in \mathit{Dom}\,(TE)\}$ and analogous for type and effect variables.

Type Rules

The type rules for the λ_{tt}^{region}-calculus are expressed in terms of the judgment $TE \vdash_t^t e : \mu, \varphi$. Its intended meaning is that in type environment TE the term e has type with place μ and effect φ. Figure 2.6 lists the individual rules.

Variables are typed with rule *(TT-Var)*, which looks up the type in the type environment. The effect of a variable is empty since an environment lookup does not require

[4]Here, we mean by *region*, the objects derivable from a region placeholder.

(TT-Var)
$$\frac{TE(x) = \mu}{TE \vdash_t^t x : \mu, \emptyset}$$

(TT-Const)
$$TE \vdash_t^t c \ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}$$

(TT-Lam)
$$\frac{TE + \{x \mapsto \mu_1\} \vdash_t^t e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_t^t \lambda x.\, e \ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\}}$$

(TT-App)
$$\frac{TE \vdash_t^t e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \quad TE \vdash_t^t e_2 : \mu_1, \varphi_2}{TE \vdash_t^t e_1 \mathbin{@} e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

(TT-Let)
$$\frac{TE \vdash_t^t e_1 : \mu_1, \varphi_1 \quad TE + \{x \mapsto \mu_1\} \vdash_t^t e_2 : \mu_2, \varphi_2}{TE \vdash_t^t \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mu_2, \varphi_1 \cup \varphi_2}$$

(TT-Copy)
$$\frac{TE \vdash_t^t e : (\mathtt{int}, \rho), \varphi}{TE \vdash_t^t \mathtt{copy}\,[\rho, \rho']\, e : (\mathtt{int}, \rho'), \varphi \cup \{\rho, \rho'\}}$$

(TT-Reglam)
$$\frac{TE \vdash_t^t e : \mu, \varphi \quad \varrho \notin frv\,(TE, \mu)}{TE \vdash_t^t \mathtt{new}\ \varrho.\, e : \mu, \varphi \setminus \{\varrho\}}$$

(TT-Letrec)
$$\frac{\begin{array}{c}(\{\varrho_1, \ldots, \varrho_n\} \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n .\tau \quad \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_t^t \lambda x.\, e_1 \ \mathtt{at}\ \rho : (\tau, \rho), \varphi' \\ \sigma = \forall\, \vec{\alpha}\, .\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\} \vdash_t^t e_2 : \mu, \varphi\end{array}}{TE \vdash_t^t \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1 \ \mathtt{at}\ \rho\ \mathtt{in}\ e_2 : \mu, \varphi \cup \varphi'}$$

(TT-Regapp)
$$\frac{\begin{array}{c}TE(f) = (\sigma, \rho) \quad \sigma = \forall\, \vec{\alpha}\, .\forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n .\tau \\ \tau' \prec \sigma \ \ via\ \ (S_t, S_e, \{\varrho_1 \mapsto \rho_1', \ldots, \varrho_n \mapsto \rho_n'\})\end{array}}{TE \vdash_t^t f\ [\rho_1', \ldots, \rho_n']\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

(TT-Effect)
$$\frac{TE \vdash_t^t e : \mu, \varphi \quad \epsilon \notin fev\,(TE, \mu)}{TE \vdash_c e : \mu, \varphi \setminus \{\epsilon\}}$$

Figure 2.6: Static semantics of the λ_{tt}^{region}-calculus

a store access[5]. A constant is of base type and typed with rule *(TT-Const)*. Its effect is only the region in which the constant is to be stored. Lambda abstractions are typed with *(TT-Lam)*. As with rule *(S-Lam)*, the body is typed with an extended type environment. The resulting effect φ may be arbitrarily enlarged into an effect φ' that is attached to an effect variable ϵ to form the arrow effect. The enlargement of the body effect is called *sub-effecting* and in principle, one could add a general sub-effecting rule for all type judgments. This is only a matter of presentation and does not affect the theory in this thesis.

The effect of evaluating the lambda abstraction is only the region in which the closure is allocated. In comparison to the region inference system of Tofte and Talpin [150], *(TT-Lam)* does not have the extra condition $frv(e) \subseteq frv(TE, \mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2)$. This condition is no principle hindrance to region inference, since it is possible to satisfy it by repeated applications of *(TT-Reglam)*, which is explained below. Tofte and Talpin introduce this side-condition to facilitate their consistency proof. However, it turns out that we do not need it for the development of the theory in this thesis.

Type rule *(TT-App)* types function application and is standard in its non-effect parts. The effect of evaluating the application includes the effect of evaluating the operator and operand, as well as the region where the function closure is stored and its entire arrow effect. Rule *(TT-App)* discharges the arrow effect because it evaluates the body of the lambda.

The rules *(TT-Let)* and *(TT-Copy)* are self-explaining. A region allocating expression $\texttt{new}\,\varrho.\,e$ is typed with rule *(TT-Reglam)*. It requires a well-typed e and ϱ may not occur free in the type environment or its type. This condition expresses that any objects in ϱ are semantically *dead* outside the lambda abstraction. However, it is possible that ϱ still occurs in the effect of e. This is safe since the deallocation only takes place after e is evaluated. Rule *(TT-Effect)* similarly allows effect variables to be taken out of the effect whenever they are dead.

Recursively bound polymorphic functions are typed with rule *(TT-Letrec)*. The conditions on the abstracted region, type, and effect variables are standard in polymorphic type systems. The `letrec`-bound lambda expression is typed with a type environment that assigns f a region- and effect-polymorphic type. However, it does *not* abstract over type variables since type reconstruction for polymorphic recursive types is undecidable [71]. Type polymorphism is only introduced in the continuation expression e_2. Finally, a region application fetches the type scheme associated with the variable f and instantiates it with appropriate types and effects. The instantiated regions have to coincide with the regions in the region application. The effect consists of the original region ρ and a new region ρ'. This is because the region application has to fetch the region function closure from the store, instantiate the regions and allocate a new function closure.

[5]In an actual implementation, the value of a variable is either stored in machine registers or in the stack. Henceforth, the term *store* always refers to dynamic memory, for example a heap.

3 A STORELESS REGION CALCULUS

3.1 Introduction

The proof of consistency for the λ_{tt}^{region}-calculus as it is given by Tofte and Talpin [150] is quite complicated. The source of the complication is twofold. First, Tofte and Talpin prove two properties at the same time: type soundness and *translation soundness*. The latter property guarantees that there is some relation between a non-region annotated program and its region-based counterpart.

The second source of complication is due to the co-inductive definition of their consistency relation, required because of the loss of information when deleting a region from the store in their big-step semantics. Apart from the somewhat awkward co-inductive formulation, their proof is very technical and lacks intuition of why deallocation safety is obtained.

In this chapter, we focus on the problem of type soundness, *i.e.* the property which guarantees that regions are not deallocated while they are still in use. We do so by reformulating the big-step operational semantics of the λ_{tt}^{region}-calculus in a relatively simple storeless transition semantics in the style of Plotkin [122]. Type soundness of the storeless model is proven with syntactic methods due to Wright, Felleisen and Harper [57, 158].

The results of this chapter have been published in [68].

3.2 The λ_{s}^{region}-calculus

The *computational* λ_{s}^{region}-calculus is very similar to the λ_{tt}^{region}-calculus, except that is defined by means of a small-step semantics. Since a transition semantics requires that intermediate computational state is expressed *within* the syntax, the λ_{s}^{region}-calculus has some additional constructs compared to the λ_{tt}^{region}-calculus.

3.2.1 Syntax

The syntax of λ_{s}^{region}-terms is reminiscent of the syntax of the λ_{tt}^{region}-calculus (see Figure 2.4 on page 22). It is presented in Figure 3.1 and we only discuss the additional computational constructs since λ_{tt}^{region}-$\mathsf{Term} \subseteq \lambda_{s}^{region}$-$\mathsf{Term}$.

The computational constructs in λ_{s}^{region}-Term are constructed by observing that terms like $c\ \mathtt{at}\ \rho$ cannot be used as values. This is because evaluation of the term $c\ \mathtt{at}\ \rho$ allocates memory, stores the constant c in it, and returns a pointer to the stored constant. As a consequence, the storeless formulation of the λ_{s}^{region}-calculus must include terms to

$$
\begin{array}{rl}
\mathsf{Term} \ni e ::= & x \mid v \mid c\ \mathtt{at}\ \rho \mid \lambda\, x.\, e\ \mathtt{at}\ \rho \mid e\ @\ e \mid \mathtt{copy}\ [\rho_1, \rho_2]\ e \mid \mathtt{new}\ \varrho.\, e \mid \\
& \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e\ \mathtt{at}\ \rho\ \mathtt{in}\ e \mid \\
& \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e \rangle_\rho\ \mathtt{in}\ e \mid f\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \\
& \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho'
\end{array}
$$

$$
\mathsf{Value} \ni v ::= \langle c \rangle_\rho \mid \langle \lambda\, x.\, e \rangle_\rho \qquad \rho \in \mathsf{Placeholder} = \mathsf{RegionVar} \cup \{\bullet\}
$$

Figure 3.1: Syntax of the λ_s^{region}-calculus

express pointers to constants and pointers to functions. These are the values $\langle c \rangle_\rho$ and $\langle \lambda\, x.\, e \rangle_\rho$ respectively. Their region annotation indicates the region where the value is living.

Similarly, we have a region closure pointer, $\langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e \rangle_\rho$, however, unlike a value v, it cannot occur in isolation[1] within a λ_s^{region}-Term. It is either bound to a variable in a `letrec`-expression or applied to some region arguments. Note that we do not have variables x as values since we only consider programs, *i.e.* closed λ_s^{region}-Terms.

In the λ_s^{region}-calculus, a region placeholder $\rho \in \mathsf{Placeholder}$ does not only include region variables, but also a distinct *dead* region $\bullet$. It is used to express region deallocation or a dangling pointer in the syntax. For example, evaluating the expression $c\ \mathtt{at}\ \bullet$ attempts to allocate a constant in a freed region of memory. Of course, this operation fails and hence, such an expression should only occur in dead code. Similarly, the value $\langle c \rangle_\bullet$ expresses a dangling pointer, which was once referring to the constant c. Having a reference to a past state of the store hardwired in the syntax is crucial to prove type soundness as we shall see later in this chapter.

3.2.2 The Dynamic Semantics

The execution semantics of the λ_s^{region}-calculus is formulated as a small-step transition semantics, designed using a set of syntactic rewrite rules. The one-step reduction rules are presented in Figure 3.2.

Rule (3.1) and rule (3.2) implement constant and lambda allocation respectively. Similarly, rule (3.3) allocates a region closure pointer within a `letrec`-expression. Region *deallocation* is specified with rule (3.4). It requires the body to be a value, which means that the body-expression has been evaluated. Deallocation replaces the region variable ϱ in the resulting value with the dead region $\bullet$ to guarantee that is not used anymore. Region *allocation* is not explicitely present in the reduction relation, except that context rule (3.13) evaluates under the region abstraction.

The `copy`-function moves a constant value from one region into the other with rule (3.5). Beta-value reduction is implemented by the rules (3.6) and (3.7) as usual (see also the reduction rules for the λ^{cbv}-calculus in Figure 2.2 on page 17). A region clo-

[1] It is possible to enforce this restriction on a type level too. Alternatively, it might be beneficial to have a more general notion of region application, an option we explore in Chapter 5.

Beta Reduction rules:

$$c\ \texttt{at}\ \varrho \to \langle c \rangle_\varrho \tag{3.1}$$

$$\lambda\, x.\, e\ \texttt{at}\ \varrho \to \langle \lambda\, x.\, e \rangle_\varrho \tag{3.2}$$

$$\begin{array}{c}\texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1\ \texttt{at}\ \varrho\ \texttt{in}\ e_2 \to \\ \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\varrho\ \texttt{in}\ e_2\end{array} \tag{3.3}$$

$$\texttt{new}\ \varrho.\, v \to v[\varrho \mapsto \bullet] \tag{3.4}$$

$$\texttt{copy}\ [\varrho_1, \varrho_2]\ \langle c \rangle_{\varrho_1} \to \langle c \rangle_{\varrho_2} \tag{3.5}$$

$$\langle \lambda\, x.\, e \rangle_\varrho\ @\ v \to e[x \mapsto v] \tag{3.6}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e \to e[x \mapsto v] \tag{3.7}$$

$$\begin{array}{c}\texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_2 \to \\ e_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho]\end{array} \tag{3.8}$$

$$\begin{array}{c}\langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e \rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \varrho' \to \\ \langle \lambda\, x.\, e[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n] \rangle_{\varrho'}\end{array} \tag{3.9}$$

Reductions in Context:

$$\frac{e \to e'}{e\ @\ e'' \to e'\ @\ e''} \tag{3.10}$$

$$\frac{e \to e'}{v\ @\ e \to v\ @\ e'} \tag{3.11}$$

$$\frac{e \to e'}{\texttt{let}\ x = e\ \texttt{in}\ e'' \to \texttt{let}\ x = e'\ \texttt{in}\ e''} \tag{3.12}$$

$$\frac{e \to e'}{\texttt{new}\ \varrho.\, e \to \texttt{new}\ \varrho.\, e'} \tag{3.13}$$

$$\frac{e \to e'}{\texttt{copy}\ [\rho_1, \rho_2]\ e \to \texttt{copy}\ [\rho_1, \rho_2]\ e'} \tag{3.14}$$

Figure 3.2: Dynamic semantics of the λ_s^{region}-calculus

sure pointer is unfolded once by rule (3.8). Rule (3.9) implements region application by substituting the regions for the formal region parameters and constructs a lambda closure.

Left-to-right call-by-value evaluation is encoded by the rules (3.10) and (3.11) for function application and in rule (3.12) for a `let`-expression. The primitive `copy` function is also reduced in a call-by-value fashion by rule (3.14). Observe that the one-step reductions, which explicitely access a region, require a region variable ϱ to ensure that deallocated pointers cannot be used in a computation. The reflexive transitive closure of the $\rightarrow$ relation is denoted by $\xrightarrow{*}$.

The crucial observation is that $\rightarrow$ does not need to propagate a store because there are no destructive update operations. Region allocation is implicit and only deallocation is visible in the reduction rules. Intuitively, every region variable, $\varrho \in \mathsf{RegionVar}$, in a term can be understood as some fragment of the heap, either local and scoped by a binder, or otherwise initial and global. Since region variables denote some region in the heap, renaming a region variable actually implies that we consider some different part in memory. This can happen by α-renaming a bound region variable or by substituting one region variable for another.

As with the $\mapsto$ reduction relation of the λ^{cbv}-calculus, we say that a λ_s^{region}-Term converges ($e \downarrow$) whenever there exists a value v such that $e \xrightarrow{*} v$. Likewise, we say that a λ_s^{region}-Term diverges ($e \uparrow$) whenever for each e' where $e \xrightarrow{*} e'$, there exists e'' with $e' \rightarrow e''$.

3.2.3 The Static Semantics

Since the λ_s^{region}-calculus is an extension of the λ_{tt}^{region}-calculus, we enrich the $\vdash_t^t$ typing judgment to accommodate for the computational terms of λ_s^{region}-Term, not present in λ_{tt}^{region}-Term.

The semantic objects found in Section 2.3.3 all retain their definitions. However, since a λ_s^{region}-$\mathsf{Placeholder}$ includes the dead region $\bullet$, we have to extend the substitution S_s as follows: $S_s(\bullet) = \bullet$.

It is often desirable to know whether a set of semantic objects contains region placeholders that are only region variables. In the case of the λ_s^{region}-calculus, this question boils down to asking whether the dead region is in the set. To this end, define the predicate $\mathit{Clean}\,(A)$ for some set A to be true iff $\mathit{frv}\,(A) \subseteq \mathsf{RegionVar}$ and false otherwise. Recall that for any object O, the operation $\mathit{frv}\,(O)$ collects all placeholders free in O. The negation of $\mathit{Clean}\,(\cdot)$ is $\mathit{Dirty}\,(\cdot)$, *i.e.* $\mathit{Dirty}\,(A)$ holds iff *not* $\mathit{Clean}\,(A)$.

For the non-computational type rules of the $\vdash_t^t$ typing judgment we refer to Section 2.3.3 for its definition and a discussion. Note that the type rules of Figure 2.6 on page 27 automatically extend to expressions with the dead region, due to the extended definition of λ_s^{region}-$\mathsf{Placeholder}$. The remaining type rules can be found in Figure 3.3.

The type rule *(TT-Constv)* is almost identical to type rule *(TT-Const)*, except that the effect is empty. This is due to the fact that values, which are pointers, do not have an effect on the store. Likewise, rule *(TT-Lam)*, which allocates a closure, affects the region of allocation, whereas the corresponding pointer reference in rule *(TT-Lamv)* has

$$\textit{(TT-Constv)} \qquad TE \vdash^t_t \langle c \rangle_\rho : (\mathtt{int}, \rho), \emptyset$$

$$\textit{(TT-Lamv)} \qquad \frac{TE + \{x \mapsto \mu_1\} \vdash^t_t e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash^t_t \langle \lambda\, x.\, e \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset}$$

$$\textit{(TT-Letrecv)} \qquad \frac{\begin{array}{c}(\{\varrho_1, \ldots, \varrho_n\} \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\, .\tau \quad \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash^t_t \langle \lambda\, x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset \\ \sigma = \forall\, \vec{\alpha}\, .\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\} \vdash^t_t e_2 : \mu, \varphi\end{array}}{TE \vdash^t_t \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \mathtt{in}\ e_2 : \mu, \varphi}$$

$$\textit{(TT-Regappv)} \qquad \frac{\begin{array}{c}\{\varrho_1, \ldots, \varrho_n\} \cap (frv\,(TE) \cup \{\rho, \rho'\}) = \emptyset \\ TE \vdash^t_t \langle \lambda\, x.\, e \rangle_{\rho'} : (\tau, \rho'), \emptyset \quad \tau' = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}(\tau)\end{array}}{TE \vdash^t_t \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

Figure 3.3: Static semantics of the λ_s^{region}-calculus

no effect at all. Analogously, rule *(TT-Letrec)* types the allocation of a recursive region closure whereas rule *(TT-Letrecv)* only types the pointer to the closure. This implies that only the effect of the `letrec`-continuation expression e_2 is included in the overall effect. Rule *(TT-Regapp)* types a region application before substituting a region closure for f whereas *(TT-Regappv)* applies after the substitution. The effect contains both the regions ρ and ρ' since such an expression is a redex (see reduction rule (3.9)) involving both regions.

3.2.4 Example

It is instructive to look at an example of a small-step evaluation and see how it preserves typing. Suppose a global region variable ϱ_1:

$$\begin{array}{l}(\mathtt{new}\, \varrho_2.\, \mathtt{let}\ \ f_1 = \lambda\, x.\, x\ \mathtt{at}\, \varrho_2 \\ \qquad\quad \mathtt{in\ let}\ \ f_2 = \lambda\, f.\, (4\ \mathtt{at}\, \varrho_1) \\ \qquad\qquad \mathtt{in}\ \ \lambda\, y.\, (f_2 \mathbin{@} f_1)\ \mathtt{at}\, \varrho_1) \mathbin{@} (2\ \mathtt{at}\, \varrho_1) \\ \xrightarrow{*} (\mathtt{new}\, \varrho_2.\, \lambda\, y.\, (\langle \lambda\, f.\, 4\ \mathtt{at}\, \varrho_1 \rangle_{\varrho_1} \mathbin{@} \langle \lambda\, x.\, x \rangle_{\varrho_2})\ \mathtt{at}\, \varrho_1) \mathbin{@} (2\ \mathtt{at}\, \varrho_1) \\ \xrightarrow{*} \langle \lambda\, y.\, (\langle \lambda\, f.\, 4\ \mathtt{at}\, \varrho_1 \rangle_{\varrho_1} \mathbin{@} \langle \lambda\, x.\, x \rangle_\bullet) \rangle_{\varrho_1} \mathbin{@} \langle 2 \rangle_{\varrho_1} \\ \xrightarrow{*} \langle \lambda\, f.\, 4\ \mathtt{at}\, \varrho_1 \rangle_{\varrho_1} \mathbin{@} \langle \lambda\, x.\, x \rangle_\bullet \\ \xrightarrow{*} \langle 4 \rangle_{\varrho_1}\end{array}$$

All intermediate terms are typeable with the rules of Figure 2.6 on page 27 and Figure 3.3 on page 33. The lambda abstraction $\lambda\, x.\, x\ \mathtt{at}\, \rho_2$ is first allocated in ρ_2, which is safely deallocated after evaluation of $\lambda\, y. \ldots\ \mathtt{at}\, \varrho_1$, since ϱ_2 neither occurs free in the type of this lambda, nor in its environment. The dangling pointer $\langle \lambda\, x.\, x \rangle_\bullet$ remains visible in the syntax, but it is never dereferenced and disappears eventually.

3.3 Type Soundness for the λ_s^{region}-calculus

We now prove type soundness for the small-step transition relation $\rightarrow$ with respect to the type system in Figure 2.6, using the additional rules from Figure 3.3.

The proof follows the structure from Section 2.2.3: we first formulate some standard lemmas. Then, we prove *type preservation*, which states that a well-typed term remains well-typed under the small-step transition relation $\rightarrow$. The second result is the *progress* property, which states that a well-typed closed term is either a value or it can be further reduced. Taken together, these two results imply type soundness.

3.3.1 Auxiliary Lemmas

First, we observe that syntactic values have no effect.

Lemma 3.3.1 *For all TE, values v, and types μ, if $TE \vdash_t^t v : \mu, \varphi$ then $\varphi = \emptyset$.*

Proof. Trivial by inspecting the rules *(TT-Constv)* and *(TT-Lamv)*. □
We will use this lemma often implicitly.

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

Lemma 3.3.2 (Canonical Forms Lemma) *The following hold:*

1. *If $TE \vdash_t^t v : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \varphi$ then there exist some x and e such that $v = \langle \lambda\, x.\, e \rangle_\rho$.*
2. *If $TE \vdash_t^t v : (\texttt{int}, \rho), \varphi$ then there exists some c such that $v = \langle c \rangle_\rho$.*

A correct type judgment yields a valid type, region or effect substituted judgment:

Lemma 3.3.3 *If $TE \vdash_t^t e : \mu, \varphi$ then, for all substitutions S_s, we have that $S_s(TE) \vdash_t^t S_s(e) : S_s(\mu), S_s(\varphi)$.*

Proof. By rule induction on the derivation of $TE \vdash_t^t e : \mu, \varphi$. Similar to Lemma 5.3 in [150] and Lemma 4.5 in [158]. □

Type schemes obey a *weakening* lemma:

Lemma 3.3.4 *If $TE + \{f \mapsto (\sigma, \rho)\} \vdash_t^t e : \mu, \rho'$, and $\sigma \prec \sigma'$, then $TE + \{f \mapsto (\sigma', \rho)\} \vdash_t^t e : \mu, \rho'$*

Proof. A straightforward induction on the depth of the proof of $TE + \{f \mapsto (\sigma, \rho)\} \vdash_t^t e : \mu, \rho'$. Proven analogously to Lemma 5.4 in [150] and Lemma 4.6 in [158]. □

Substituting a value of the correct type for a variable of the same type preserves the type and the effect of the enclosing term. This property is the *first substitution lemma*:

Lemma 3.3.5 (First Substitution Lemma) *Suppose $TE + \{x \mapsto \mu\} \vdash_t^t e : \mu', \varphi'$ and $TE \vdash_t^t v : \mu, \varphi$, then $TE \vdash_t^t e[x \mapsto v] : \mu', \varphi'$.*

Proof. By rule induction on the derivation of $TE + \{x \mapsto \mu\} \vdash_t^t e : \mu', \varphi'$. The only interesting case is the application of *(TT-Var)* to (free occurrences of) x. If $TE' = TE + \{x \mapsto \mu\}$, then since $TE'(x) = TE + \{x \mapsto \mu\}(x) = \mu$, rule *(TT-Var)* yields $TE' \vdash_t^t x : \mu, \emptyset$. On the other hand, $x[x \mapsto v] = v$ as well. By assumption, $TE \vdash_t^t v : \mu, \emptyset$ since $\varphi = \emptyset$, by Lemma 3.3.1.

All other cases are simple appeals to the inductive hypothesis. □

Since the λ_s^{region}-calculus also substitutes a polymorphic region closure pointer (see reduction rule (3.8)), we have a second substitution lemma. It is slightly more technical:

Lemma 3.3.6 (Second Substitution Lemma) *Suppose*

1. $\{\varrho_1 \ldots, \varrho_n, \vec{\epsilon}, \vec{\alpha}\} \cap (fv(TE) \cup \{\rho\}) = \emptyset$ *and* $\sigma = \forall \vec{\alpha}.\hat{\sigma}$ *and* $\hat{\sigma} = \forall \vec{\epsilon}.\forall \varrho_1 \ldots \varrho_n.\tau$
2. $TE + \{f \mapsto (\sigma, \rho)\} \vdash_t^t e_2 : \mu, \varphi$
3. $TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_t^t \langle \lambda x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset$

then $TE \vdash_t^t e_2[f \mapsto \langle \Lambda \varrho_1, \ldots, \varrho_n.\lambda x.\, \texttt{letrec}\ f = \langle \Lambda \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho] : \mu, \varphi$.

Proof. By induction on the derivation of $TE + \{f \mapsto (\sigma, \rho)\} \vdash_t^t e_2 : \mu, \varphi$. All cases are straightforward applications of the induction hypothesis, except for free occurrences of f in e_2.

In this case, Item 2 is as follows.

$$TE + \{f \mapsto (\sigma, \rho)\} \vdash_t^t f\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' : \mu, \varphi \tag{3.15}$$

This judgment must be due to rule *(TT-Regapp)*, that is, there exists $S_s = (S_t, S_r, S_e)$ such that $\mu = (\tau', \rho')$; $\varphi = \{\rho, \rho'\}$; $(TE + \{f \mapsto (\sigma, \rho)\})(f) = (\sigma, \rho)$; $\tau' \prec \sigma$ *via* S_s; and $S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}$.

Hence, we must show the following judgment:

$$\begin{array}{rl} TE \vdash_t^t & \langle \Lambda \varrho_1, \ldots, \varrho_n.\lambda x.\, \texttt{letrec}\ f = \langle \Lambda \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' \\ & : (\tau', \rho'), \{\rho, \rho'\} \end{array} \tag{3.16}$$

By α-renaming the ϱ_i, we can extend Item 1 to

$$\{\varrho_1 \ldots, \varrho_n, \vec{\epsilon}, \vec{\alpha}\} \cap (fv(TE) \cup \{\rho, \rho'\}) = \emptyset \tag{3.17}$$

Define the α-renaming region substitution $\chi = \{\varrho_1 \mapsto \varrho'_1, \ldots, \varrho_n \mapsto \varrho'_n\}$, such that

$$\{\varrho'_1, \ldots, \varrho'_n\} \cap (frv(TE) \cup Img(S_s) \cup \{\rho, \rho', \varrho_1, \ldots, \varrho_n\}) = \emptyset \tag{3.18}$$

Using the substitution χ and assumption (3.17), judgment (3.16) can be reformulated as follows:

$$\begin{array}{rl} TE \vdash_t^t & \langle \Lambda \varrho'_1, \ldots, \varrho'_n.\lambda x.\, \texttt{letrec}\ f = \langle \Lambda \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1 \rangle_\rho\ \texttt{in}\ \chi(e_1) \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' \\ & : (\tau', \rho'), \{\rho, \rho'\} \end{array}$$

where we now have that the type scheme $\sigma = \forall \vec{\alpha}.\forall \vec{\epsilon}.\forall \varrho'_1 \ldots \varrho'_n.\chi(\tau)$, the substitution $S_r = \{\varrho'_1 \mapsto \rho'_1, \ldots, \varrho'_n \mapsto \rho'_n\}$.

By type rule *(TT-Regappv)*, we must show that for some τ'', the following holds:

- $TE \vdash_t^t \langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ \chi(e_1) \rangle_{\rho'} : (\tau'', \rho'), \emptyset$;
- $\{\varrho'_1, \ldots, \varrho'_n\} \cap (frv\,(TE) \cup \{\rho, \rho'\}) = \emptyset$ (this is immediate by the assumption (3.18));
- $\tau' = S_r(\tau'')$.

A suitable choice for τ'' is $S'_s(\chi(\tau))$ where $S'_s = (S_t, I_r, S_e)$. Because of the assumptions in (3.18), we have $Supp\,(S_r) \cap Img\,(S'_s) = \emptyset$ and hence that $S_s = S_r \circ S'_s$. Since $\tau' \prec \sigma$ *via* S_s, the type τ'' satisfies the requirement that $\tau' = S_r(\tau'')$.

It remains to show that

$$TE \vdash_t^t \langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ \chi(e_1) \rangle_{\rho'} : (S'_s(\chi(\tau')), \rho'), \emptyset \tag{3.19}$$

This is the same as showing the judgment

$$TE \vdash_t^t \langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_{\rho'} : (S'_s(\tau'), \rho'), \emptyset \tag{3.20}$$

since we can apply χ on (3.20), using the assumptions in (3.17) and (3.18) and Lemma 3.3.3.

To prove (3.20), we first show that

$$TE \vdash_t^t \langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{3.21}$$

To this end, suppose that $\tau = \mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2$. Then judgment (3.21) follows from Item 3, type rule *(TT-Lamv)*, and the following implication:

If, for some $\varphi'' \subseteq \varphi'$,

$$TE + \{f \mapsto (\hat{\sigma}, \rho)\} + \{x \mapsto \mu_1\} \vdash_t^t e_1 : \mu_2, \varphi'' \tag{3.22}$$

then

$$TE + \{x \mapsto \mu_1\} \vdash_t^t \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 : \mu_2, \varphi'' \tag{3.23}$$

In order to show the latter judgment, look at Item 3 of the lemma. From this, we trivially obtain that

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_t^t \langle \lambda\, x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{3.24}$$

Applying Lemma 3.3.4 to judgment (3.22) for $\hat{\sigma} \prec \sigma$ and observing that $TE + \{f \mapsto (\hat{\sigma}, \rho)\} + \{x \mapsto \mu_1\} = TE + \{x \mapsto \mu_1\} + \{f \mapsto (\hat{\sigma}, \rho)\}$ because $x \neq f$ yields

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\sigma, \rho)\} \vdash_t^t e_1 : \mu_2, \varphi'' \tag{3.25}$$

Applying rule *(TT-Letrecv)* to Item 1, judgment (3.24), and judgment (3.25) yields the claim in (3.23).

Applying Lemma 3.3.3 to judgment (3.21) for S'_s yields

$$S'_s(TE) \vdash_t^t S'_s(\langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho) : S'_s(\tau, \rho), S'_s(\emptyset) \tag{3.26}$$

The domain of $S'_s = (S_t, I_r, S_e)$ is a subset of $\{\vec{\epsilon}, \vec{\alpha}\}$ since $\tau' \prec \sigma$ *via* (S_t, S_r, S_e). By assumption (1), $\{\vec{\epsilon}, \vec{\alpha}\}$ is disjoint from $fv\,(TE) \cup \{\rho\}$. Therefore,

- $S'_s(TE) = TE$;
- $S'_s(\langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho) =$ due to I_r; $\langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_1 \rangle_\rho$
- $S'_s(\tau, \rho) = (S'_s(\tau), \rho)$.

Using the type rule *(TT-Lamv)* once backwards and once forwards replaces ρ by ρ' and transforms judgment (3.26) into judgment (3.20), which proves the claim. □

3.3.2 Type Preservation

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

Proposition 3.3.7 (Type Preservation) *Suppose $TE \vdash_t^t e : \mu, \varphi$. If $e \to e'$ then there exists an effect φ' for which $TE \vdash_t^t e' : \mu, \varphi'$ and $\varphi' \subseteq \varphi$.*

Proof. By induction on the definition of $\to$ and the number of subsequent uses of *(TT-Effect)* at the end of e's type derivation.

If $TE \vdash_t^t e : \mu, \varphi$ derives from $TE \vdash_t^t e : \mu, \varphi \cup \{\epsilon\}$ by rule *(TT-Effect)*, for some $\epsilon \notin fev\,(TE, \mu, \varphi)$, then induction yields that $TE \vdash_t^t e' : \mu, \varphi'$ where $\varphi' \subseteq \varphi \cup \{\epsilon\}$. If $\epsilon \notin \varphi'$ then the claim holds anyway. Otherwise, use *(TT-Effect)* to get $TE \vdash_t^t e' : \mu, \varphi' \setminus \{\epsilon\}$ where $\varphi' \setminus \{\epsilon\} \subseteq \varphi \cup \{\epsilon\} \setminus \{\epsilon\} = \varphi$, as required.

If the last rule in the proof of $TE \vdash_t^t e : \mu, \varphi$ is not *(TT-Effect)* then perform a case analysis. Most cases are straightforward applications of the inductive hypothesis and/or one of the preceding lemmas.

Case $c\ \texttt{at}\ \varrho \to \langle c \rangle_\varrho$**.** Is obvious by rules *(TT-Const)* and *(TT-Constv)*.

Case $\lambda\, x.\, e\ \texttt{at}\ \varrho \to \langle \lambda\, x.\, e \rangle_\varrho$**.** Is obvious by rules *(TT-Lam)* and *(TT-Lamv)*.

Case $\begin{array}{l} \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1\ \texttt{at}\ \varrho\ \texttt{in}\ e \to \\ \quad \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, e_1 \rangle_\varrho\ \texttt{in}\ e \end{array}$.
Is also obvious by rules *(TT-Letrec)* and *(TT-Letrecv)*.

Case $\texttt{new}\ \varrho.\, v \to v[\varrho \mapsto \bullet]$**.**
The last rule in the type derivation of the redex must be *(TT-Reglam)*:

$$\frac{TE \vdash_t^t v : \mu, \varphi' \quad \varrho \notin frv\,(TE, \mu)}{TE \vdash_t^t \texttt{new}\ \varrho.\, v : \mu, \varphi' \setminus \{\varrho\}}$$

By Lemma 3.3.1, $\varphi' = \emptyset$, hence $\varphi' \setminus \{\varrho\} = \emptyset$. By Lemma 3.3.3, $TE[\varrho \mapsto \bullet] \vdash_t^t v[\varrho \mapsto \bullet] : \mu[\varrho \mapsto \bullet], \varphi'[\varrho \mapsto \bullet]$. Since $\varrho \notin frv\,(TE, \mu)$ and $\varphi' = \emptyset$, this amounts to $TE \vdash_t^t v[\varrho \mapsto \bullet] : \mu, \emptyset$, which verifies the claim.

Case $\mathtt{copy}\,[\varrho_1,\varrho_2]\,\langle c\rangle_{\varrho_1} \to \langle c\rangle_{\varrho_2}$**.** By assumption, $TE \vdash^t_t \mathtt{copy}\,[\varrho_1,\varrho_2]\,\langle c\rangle_{\varrho_1} : \mu,\varphi$, for some μ and φ. Hence, by rules *(TT-Copy)* and *(TT-Constv)*, $\mu = (\mathtt{int},\varrho_2)$ and $\varphi = \{\varrho_1,\varrho_2\}$. For the reduct, rule *(TT-Constv)* yields $TE \vdash^t_t \langle c\rangle_{\varrho_2} : (\mathtt{int},\varrho_2),\emptyset$, which verifies the claim.

Case $\langle\lambda\, x.\, e\rangle_\varrho \mathbin{@} v \to e[x \mapsto v]$**.** The last rule in the type derivation of the redex must be *(TT-App)*, hence (using Lemma 3.3.1)

$$\frac{TE \vdash^t_t \langle\lambda\, x.\, e\rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2,\varrho),\emptyset \quad TE \vdash^t_t v : \mu_1,\emptyset}{TE \vdash^t_t (\langle\lambda\, x.\, e\rangle_\varrho) \mathbin{@} v : \mu_2,\varphi \cup \{\epsilon,\varrho\}}$$

By rule *(TT-Lamv)* it must be that $TE + \{x \mapsto \mu_1\} \vdash^t_t e : \mu_2,\varphi'$ for some $\varphi' \subseteq \varphi$. By the First Substitution Lemma 3.3.5, it follows that $TE \vdash^t_t e[x \mapsto v] : \mu_2,\varphi'$. Clearly, $\varphi' \subseteq \varphi \cup \{\epsilon,\varrho\}$, which proves the claim.

Case $\mathtt{let}\; x = v\; \mathtt{in}\; e \to e[x \mapsto v]$**.** Immediate by inspection of the type rule *(TT-Let)* and the First Substitution Lemma 3.3.5.

Case
$$\mathtt{letrec}\; f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e_1\rangle_\rho\; \mathtt{in}\; e_2 \to e_2[f \mapsto \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, \mathtt{letrec}\; f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e_1\rangle_\rho\; \mathtt{in}\; e_1\rangle_\rho]\;.$$
The last step in the typing derivation for the redex must apply rule *(TT-Letrecv)*:

$$\frac{\begin{array}{c}\{\varrho_1\ldots,\varrho_n,\vec{\epsilon}\} \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \quad \hat{\sigma} = \forall\,\vec{\epsilon}.\forall\,\varrho_1\ldots\varrho_n\,.\tau \\ TE + \{f \mapsto (\hat{\sigma},\rho)\} \vdash^t_t \langle\lambda\, x.\, e_1\rangle_\rho : (\tau,\rho),\emptyset \quad \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \\ \sigma = \forall\,\vec{\alpha}\,.\hat{\sigma} \quad TE + \{f \mapsto (\sigma,\rho)\} \vdash^t_t e_2 : \mu,\varphi\end{array}}{TE \vdash^t_t \mathtt{letrec}\; f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e_1\rangle_\rho\; \mathtt{in}\; e_2 : \mu,\varphi}$$

By the Second Substitution Lemma 3.3.6, this yields
$TE \vdash^t_t e_2[f \mapsto \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, \mathtt{letrec}\; f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e_1\rangle_\rho\; \mathtt{in}\; e_1\rangle_\rho] : \mu,\varphi$ as required.

Case $\langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho\;[\rho'_1,\ldots,\rho'_n]\;\mathtt{at}\;\varrho' \to \langle\lambda\, x.\, e[\varrho_1 \mapsto \rho'_1,\ldots,\varrho_n \mapsto \rho'_n]\rangle_{\varrho'}$**.**
The last step of the typing derivation for the redex is *(TT-Regappv)*:

$$\frac{\begin{array}{c}S_r = \{\varrho_1 \mapsto \rho'_1,\ldots,\varrho_n \mapsto \rho'_n\} \quad \tau' = S_r(\tau) \\ \{\varrho_1,\ldots,\varrho_n\} \cap (frv\,(TE) \cup \{\rho,\rho'\}) = \emptyset \quad TE \vdash^t_t \langle\lambda\, x.\, e\rangle_{\varrho'} : (\tau,\varrho'),\emptyset\end{array}}{TE \vdash^t_t \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho\;[\rho'_1,\ldots,\rho'_n]\;\mathtt{at}\;\varrho' : (\tau',\varrho'),\{\varrho,\varrho'\}}$$

So, by Lemma 3.3.3, we have $S_r(TE) \vdash^t_t S_r(\langle\lambda\, x.\, e\rangle_{\varrho'}) : S_r(\tau,\varrho'),\emptyset$, but since $S_r(\tau) = \tau'$ and $\{\varrho_1,\ldots,\varrho_n\} \cap (frv\,(TE) \cup \{\rho,\rho'\}) = \emptyset$, we have that $TE \vdash^t_t \langle\lambda\, x.\, S_r(e)\rangle_{\varrho'} : (\tau',\varrho'),\emptyset$.

Case $\dfrac{e \to e'}{e \mathbin{@} e'' \to e' \mathbin{@} e''}$**.** The last step in the typing derivation of the left term is the rule *(TT-App)*:

$$\frac{TE \vdash^t_t e : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2,\rho),\varphi_1 \quad TE \vdash^t_t e'' : \mu_1,\varphi_2}{TE \vdash^t_t e \mathbin{@} e'' : \mu_2,\varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon,\rho\}}$$

By induction, we have that $TE \vdash_t^t e' : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1'$ where $\varphi_1' \subseteq \varphi_1$. Again by rule *(TT-App)*, this yields

$$\frac{TE \vdash_t^t e' : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1' \quad TE \vdash_t^t e'' : \mu_1, \varphi_2}{TE \vdash_t^t e \mathbin{@} e'' : \mu_2, \varphi \cup \varphi_1' \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

and $(\varphi \cup \varphi_1' \cup \varphi_2 \cup \{\epsilon, \rho\}) \subseteq (\varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\})$, proving the case.

The cases (3.11) through (3.14).
All by similarly simple appeal to the inductive hypothesis.

□

3.3.3 Progress

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

Proposition 3.3.8 (Progress) *If* $\{\ \} \vdash_t^t e : \mu, \varphi$ *and* $Clean\,(\varphi)$ *then either*

1. *there exists a* λ_s^{region}-Term e' *such that* $e \to e'$ *or*
2. e *is a* λ_s^{region}-Value v.

Proof. By rule induction on the type derivation $\{\ \} \vdash_t^t e : \mu, \varphi$. Note that we only consider closed expressions, hence, we do not treat the cases dealing with free variables.

Case $\{\ \} \vdash_t^t c\ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}$. Since $Clean\,(\{\rho\})$ by assumption, we have $\rho \neq \bullet$ and can reduce with rule (3.1). Hence Item 1 applies.

Case $\{\ \} \vdash_t^t \lambda\, x.\, e\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\}$. Again by assumption we have that $Clean\,(\{\rho\})$ or $\rho \neq \bullet$, so we reduce with rule (3.2).

Case $\{\ \} \vdash_t^t e_1 \mathbin{@} e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$. Hence, by rule *(TT-App)*, we have $\{\ \} \vdash_t^t e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1$ and $\{\ \} \vdash_t^t e_2 : \mu_1, \varphi_2$. We also have $Clean\,(\varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\})$, so by induction, we consider the following two cases for e_1:

- Either we have that $e_1 \to e_1'$. This allows application of reduction rule (3.10) and Item 1 applies.
- Or, we have that e_1 is a value v. Again by induction, we now make a case distinction for e_2:
  - Suppose we have that $e_2 \to e_2'$. Then, we can reduce with rule (3.11).
  - Alternatively, we have that e_2 is a value v'. In that case, we have by the canonical forms Lemma 3.3.2 that v has the form $\langle \lambda\, x.\, e \rangle_\rho$. Moreover, since we have that $Clean\,(\{\epsilon, \rho\})$, we know that $\rho \neq \bullet$. Hence, we can reduce the application with rule (3.6) and Item 1 applies.

Case $\{\ \} \vdash_t^t \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mu_2, \varphi_1 \cup \varphi_2$**.** Via type rule *(TT-Let)*, we have $\{\ \} \vdash_t^t e_1 : \mu_1, \varphi_1$. As *Clean* $(\varphi_1 \cup \varphi_2)$ holds, by induction we either have that e_1 is a value v in which we case we reduce with rule (3.7). Or, if $e_1 \to e_1'$, then we can apply context rule (3.12).

Case $\{\ \} \vdash_t^t \texttt{copy}\,[\rho, \rho']\, e : (\texttt{int}, \rho'), \varphi \cup \{\rho, \rho'\}$**.** Type rule *(TT-Copy)* makes that $\{\ \} \vdash_t^t e : (\texttt{int}, \rho), \varphi$. Also *Clean* $(\varphi \cup \{\rho, \rho'\})$, hence, by induction we either have that e is a value v. Moreover, by the canonical forms Lemma 3.3.2 it must be that v is of the form $\langle c \rangle_\rho$. Since *Clean* (ρ, ρ'), we can then apply reduction rule (3.5).

Otherwise, we have that $e \to e'$ and so, again Item 1 applies, since we can reduce with context rule (3.14).

Case $\{\ \} \vdash_t^t \texttt{new}\,\varrho.\, e : \mu, \varphi \setminus \{\varrho\}$**.** Analogously, we apply the induction hypothesis because of judgment $\{\ \} \vdash_t^t e : \mu, \varphi$ due to type rule *(TT-Reglam)* (where obviously also *Clean* (φ) since $\varrho \neq \bullet$). So, whenever e is a value v, we reduce with rule (3.4). Otherwise, we have the one-step reduction $e \to e'$ and context rule (3.13) applies.

Case $\{\ \} \vdash_t^t \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e_1\ \texttt{at}\ \rho\ \texttt{in}\ e_2 : \mu, \varphi \cup \varphi'$**.**
Since $\rho \in \varphi$ and *Clean* $(\varphi \cup \varphi')$, we can reduce this expression with rule (3.3) and hence, Item 1 applies.

Case $\{\ \} \vdash_t^t \langle c \rangle_\rho : (\texttt{int}, \rho), \emptyset$**.** The expression $\langle c \rangle_\rho$ is a value and hence Item 2 applies.

Case $\{\ \} \vdash_t^t \langle \lambda\, x.\, e \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset$**.** The expression $\langle \lambda\, x.\, e \rangle_\rho$ is a value and hence Item 2 applies.

Case $\{\ \} \vdash_t^t \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e_1 \rangle_\rho\ \texttt{in}\ e_2 : \mu, \varphi$**.** This expression is immediately reducible with transition rule (3.8) and so Item 1 applies once more.

Case $\{\ \} \vdash_t^t \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\rho\ [\rho_1', \ldots, \rho_n']\ \texttt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}$**.**
Since *Clean* $(\{\rho, \rho'\})$ holds by assumption, we can reduce this expression with rule (3.9) and Item 1 applies.

□

3.3.4 Type Soundness

Using Type Preservation (Proposition 3.3.7) and Progress (Proposition 3.3.8), it is relatively straightforward to prove a strong type soundness result. It says that a well-typed closed term either gives rise to an infinite reduction sequence, or eventually reduces to a value of the same type.

Theorem 3.3.9 (Type Soundness of the λ_s^{region}-calculus) *Suppose that* $\{\ \} \vdash_t^t e : \mu, \varphi$ *with* *Clean* (φ)*. Then, either*

1. *there exists some value* v *such that* $e \xrightarrow{*} v$ *and* $\{\ \} \vdash_t^t v : \mu, \emptyset$ *or;*

2. $e \uparrow$.

Proof. Consider an arbitrary e' such that $e \xrightarrow{*} e'$ (such an e' always exists), now it remains to be proven that

1. the expression e' is a value v such that $\{\ \} \vdash_t^t v : \mu, \emptyset$ or;
2. $e' \rightarrow e''$ (by definition of divergence).

We prove by induction on the length of the reduction and have two cases:

- Suppose $e' = e$. Then by Progress, we either have that e is a value v. In that case Item 1 applies due to Lemma 3.3.1 and the assumption that $\{\ \} \vdash_t^t e : \mu, \varphi$. Otherwise, Progress says that $e' \rightarrow e''$ and Item 2 applies.
- Suppose $e \rightarrow e''$ and $e'' \xrightarrow{*} e'$. By Type Preservation, we have that $\{\ \} \vdash_t^t e'' : \mu, \varphi'$, where $\varphi' \subseteq \varphi$. Hence, we also have that $\mathit{Clean}\,(\varphi')$ and we can apply the induction hypothesis to obtain the result.

□

3.4 Related Work

The λ_{tt}^{region}-calculus was originally introduced by Tofte and Talpin [149,150]. The formal correctness proof, however, is very complex and therefore not very intuitive or scalable. This situation has stimulated other researchers to pursue alternative type soundness proofs:

1. Crary, Walker, and Morrisett [28,154] provide an indirect soundness proof by translating the region calculus into their capability calculus. The capability calculus has a sophisticated type-and-effect system that supports safe allocation and deallocation of regions in an arbitrary order. This added flexibility may lead to a better use of memory at runtime, since there are cases where a region may be de-allocated earlier than in the region calculus. Their soundness proof for the capability calculus is also syntactic.

2. Banerjee, Heintze, and Riecke [9] translate the region calculus into $F_{\#}$, an extension of the polymorphic lambda calculus with a special type constructor for encapsulation. They construct an original denotational model for their calculus and give a semantic soundness proof based on the model.

3. Dal Zilio and Gordon [159, 160] modify the operational semantics of Tofte and Talpin so that it also keeps track of deallocated regions. This extra information allows an inductive definition of the consistency relation and an inductive correctness proof. They go on to show that this result is a consequence of a more general result for a typed π-calculus with name groups. This is shown by using a translation from the region calculus into their extended π-calculus.

4. Calcagno [17] defines a *high-level* big-step operational semantics and proves type soundness for it. Instead of marking deallocated pointers, he propagates a set of live regions. He also formally relates the high-level semantics to the original *low-level* semantics of the λ_{tt}^{region}-calculus.

Compared to these approaches, we give a direct and syntactic soundness proof that is extremely concise and simple. With the exception of the work by Calcagno [17], the above mentioned alternatives do not treat a polymorphic region calculus and are indirect via another presentation. As noted by Tofte and Talpin [150], type polymorphism does not add conceptual problems to the type soundness result, but polymorphic recursion gives rise to some subtle twists as witnessed by the Second Substitution Lemma 3.3.6 on page 35. Except for the big-step versus small-step approach, Calcagno's work comes closest to ours and was carried out independently and at the same time[2].

3.5 Chapter Summary

In Section 3.2, we presented the λ_{s}^{region}-calculus as an alternative characterization of the λ_{tt}^{region}-calculus, by giving it a small-step operational semantics. Following syntactic proof methods [57, 158], Section 3.3 detailed a type soundness proof for the transition semantics of the λ_{s}^{region}-calculus. Since it is solely based on rewriting and induction, the proof is considerably easier than the original soundness proof of Tofte and Talpin. We were able to elide an explicit store from our presentation of the semantics because the λ_{tt}^{region}-calculus (and hence also the λ_{s}^{region}-calculus) never updates the contents of the store. Including references in our framework is possible at the price of including an explicit store component in the transition relation. We explore this possibility in Chapter 4.

[2]Personal Communication, August 2000

4 AN IMPERATIVE REGION CALCULUS

4.1 Introduction

As already mentioned in the last chapter, neither the λ_{tt}^{region}-calculus nor the λ_{s}^{region}-calculus model update operations on the store. In the case of the λ_{tt}^{region}-calculus, Tofte and Talpin [150] simply add such primitives in the type environment by giving them appropriate type schemes. However, since no dynamic semantics is formulated for these operations, there is no proof for their soundness.

In the case of the λ_{s}^{region}-calculus, adding destructive store operators is impossible due to the absence of a store in the transition semantics. In this chapter, we extend the λ_{s}^{region}-calculus with a reference constructor, a destructive update operator as well as a dereferencing primitive. However, we only consider a non-recursive monomorphic version of the region calculus since the extension to polymorphism and recursion is tedious, but standard along the lines explained in Chapter 3.

Since the update operations require an explicit store in the semantics, the challenge is to come up with a store-based small-step operational semantics that again admits a syntactic type soundness proof. Henceforth, we refer to our monomorphic region calculus with destructive store operations and associated small-step semantics as the λ_{i}^{region}-calculus. Operationally, the λ_{i}^{region}-calculus closely matches the actual intermediate language used in the ML-kit [148].

Apart from giving a direct type soundness proof for a region calculus with store operations, another goal of this chapter is to relate the λ_{s}^{region}-calculus with the λ_{tt}^{region}-calculus. The λ_{i}^{region}-calculus is an excellent intermediate representation, which we first relate to the λ_{tt}^{region}-calculus, and then also to the λ_{s}^{region}-calculus. However, in relating these calculi, we only treat their monomorphic and *pure* subsets, *i.e.* without destructive store operations.

The contributions of this chapter are the result of joint work with Cristiano Calcagno and have been published [18].

4.2 The Imperative λ_{i}^{region}-calculus

We first describe the syntax of the λ_{i}^{region}-calculus, which adds destructive store operators and additional syntax to cater for intermediate computational terms. Then, we formulate a small-step reduction semantics and a type system for the λ_{i}^{region}-calculus.

$$\begin{array}{rl} \mathsf{Term} \ni e ::= & v \mid x \mid c\ \mathtt{at}\ \rho \mid \lambda x.\, e\ \mathtt{at}\ \rho \mid e\ @\ e \mid \mathtt{copy}\ [\rho_1, \rho_2]\ e \mid \mathtt{new}\ \varrho.\, e \mid \\ & \mathtt{region}\ r\ \mathtt{in}\ e \mid \mathtt{ref}\ e\ \mathtt{at}\ \rho \mid\ !\ e \mid e := e \end{array}$$

$$\mathsf{Value} \ni v ::= (r, l) \mid (\bullet, l) \qquad \mathsf{Placeholder} = \mathsf{RegionVar} \cup \mathsf{RegionName} \cup \{\bullet\}$$

Figure 4.1: Syntax of the λ_i^{region}-calculus

4.2.1 Syntax of the λ_i^{region}-calculus

The syntax for the λ_i^{region}-calculus is derived from a monomorphic subset of the λ_{tt}^{region}-calculus and is presented in Figure 4.1.

As with the syntax of the λ_s^{region}-calculus, we introduce intermediate computational terms. Similar to values for the big-step semantics of the λ_{tt}^{region}-calculus, a λ_i^{region}-Value is a pair of a *region name* $r \in \mathsf{RegionName}$ and location $l \in \mathsf{Location}$ (see Section 2.3 for the definitions of these syntactic categories). However, since a value v is *part* of the syntax, we have to model a dangling pointer similarly to dereferenced pointers in the λ_s^{region}-calculus. Hence, the value $(\bullet, l)$ represents a deallocated object. In theory we could have used $\bullet$ instead of the pair $(\bullet, l)$ since the offset l plays no role anymore after region deallocation. However, it would complicate the reduction semantics, since we wish to make substitutions of the form $e[r \mapsto \bullet]$, as defined below.

Compared to the λ_s^{region}-calculus, we also extend region placeholders to include region names, next to region variables and the dead region. This is due to the fact that in the λ_i^{region}-calculus, we make a distinction between an abstracted region variable and an actual region in the store, which is represented by a region name. One can understand a region variable as a *not yet allocated* region, a region name as an *already allocated* region and the dead region as a *deallocated* region.

Variables, constants, lambdas, function application, the primitive copy operator and region abstraction have a similar semantics as the analogous constructs of the λ_{tt}^{region}-calculus in Section 2.3.1. In addition, there is a new intermediate term, $\mathtt{region}\ r\ \mathtt{in}\ e$. The binding construct, $\mathtt{new}\ \varrho.\, e$, reduces to $\mathtt{region}\, r\, \mathtt{in}\, e[\varrho \mapsto r]$ as soon as the evaluation has committed to a particular region. It is important to note that $\mathtt{region}\ r\ \mathtt{in}\ e$ does not bind r, but merely records the region used in the store. This annotation is required to successfully deallocate r after finishing the evaluation of e.

Finally, the set of λ_i^{region}-Terms now includes the Standard ML operations on references: creation of a reference, $\mathtt{ref}\ e\ \mathtt{at}\ \rho$, dereferencing a reference, $!\ e$, and updating a reference, $e := e'$.

4.2.2 Dynamic Semantics of the λ_i^{region}-calculus

The extra semantic objects for the transition semantics of the λ_i^{region}-calculus are very similar to those of the big-step semantics of the λ_{tt}^{region}-calculus (see Section 2.3.2). We define a storable value $w \in \lambda_i^{region}$-Storable, either as a constant $\langle c \rangle$, a lambda closure

$\langle \lambda\, x.\, e \rangle$ or a reference to a value $\langle \texttt{ref}\ v \rangle$. In contrast with the λ_{tt}^{region}-calculus, a lambda closure does not need to keep track of a region and value environment, since our transition semantics is a substitution semantics. As with the big-step semantics of the λ_{tt}^{region}-calculus, we define a *region*, $p \in \mathsf{Location} \hookrightarrow \mathsf{Storable}$, as a finite function from locations to storable values. A *store*, $s \in \mathsf{RegionName} \hookrightarrow (\mathsf{Location} \hookrightarrow \mathsf{Storable})$, is a finite function from region names to regions.

Recall that $frv\,(e)$ collects all the free *region placeholders* in the expression e. However, in the λ_i^{region}-calculus, region placeholders include region names $r \in \mathsf{RegionName}$. We define the free region variables of an expression e as usual and the free region *names* in e are *all* occurrences of r in e, where we stress that the expression $\texttt{region}\ r\ \texttt{in}\ e$ does not bind r in e. The definition of $frv\,(-)$ is extended to all other semantic objects O in the usual way. In the case of a store s, we define $frv\,(s) = \bigcup\{frv\,(s(r)(l)) \mid r \in Dom\,(s), l \in Dom\,(s(r))\}$. For the purposes of this chapter, we redefine the $Clean\,(A)$ predicate to be true iff $frv\,(A) \subseteq \mathsf{RegionName}$, *i.e.* we are interested in regions which are already allocated but not deallocated.

With some abuse of notation, $e[r \mapsto \bullet]$ denotes the term e after substitution of $\bullet$ for all free occurrences of the region name r in e. Similarly, for a semantic object O, the substitution $O[r \mapsto \bullet]$ denotes the object after substitution of $\bullet$ for all $r \in frv\,(O)$.

The dynamic semantics is defined as a transition relation on configurations. A *configuration* s, e is a pair of an $s \in \lambda_i^{region}$-Store, and a closed expression $e \in \lambda_i^{region}$-Term. Figure 4.2 shows the small-step reduction semantics on configurations.

By convention, we say that a reduction is not in the $\rightarrowtail$ relation whenever the right-hand side of the transition is not defined.

Rule (4.1) *allocates* a fresh region in the store s, by picking a free region $r \notin Dom\,(s)$. It is initialized with the empty function and the `region`-construct registers this commitment in the syntax. In contrast, rule (4.2) *deallocates* the region r from the store, once the body expression e has been evaluated with context rule (4.10). It substitutes all free occurrences of r in both the store s and the value v for $\bullet$ and removes the region function r from the store.

Constant values and lambda abstractions are allocated with the rules (4.3) and (4.4) respectively. They fetch a new offset $l \notin Dom\,(s(r))$ in the committed region r and put the respective storable value in the new cell. The result of evaluating these allocating expressions is the address (r, l), referring to the allocated storable.

Beta-value reduction is formalized with rule (4.5), where the operator is fetched from the store and then applied on a value v as usual, by substituting it for the formal parameter x in the body of the closure. The beta rule for the copy primitive is given in (4.6) and retrieves a constant from the given address. It generates a new offset in the target region r_2 and copies the constant into that cell. The result of the evaluation is the newly generated reference.

Similarly to the allocation for constants in rule (4.3) and lambdas in rule (4.4), a reference storable $\langle \texttt{ref}\ v \rangle$ is allocated by the rule (4.7). It is dereferenced by a lookup in the store as specified by rule (4.8). A destructive update is implemented with rule (4.9): it replaces the cell with the pointer to the new storable.

The context rules (4.11) and (4.12) implement a left-to-right call-by-value semantics

Beta Reduction rules:

$$s, \texttt{new}\ \varrho.\, e \rightarrowtail s + \{r \mapsto \{\ \}\}, \texttt{region}\ r\ \texttt{in}\ e[\varrho \mapsto r] \quad \textit{if } r \notin \textit{Dom}\,(s) \tag{4.1}$$

$$s, \texttt{region}\ r\ \texttt{in}\ v \rightarrowtail s[r \mapsto \bullet] - r, v[r \mapsto \bullet] \tag{4.2}$$

$$s, c\ \texttt{at}\ r \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l) \quad \textit{if } l \notin \textit{Dom}\,(s(r)) \tag{4.3}$$

$$s, \lambda\, x.\, e\ \texttt{at}\ r \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}, (r, l) \quad \textit{if } l \notin \textit{Dom}\,(s(r)) \tag{4.4}$$

$$s, (r, l)\ @\ v \rightarrowtail s, e[x \mapsto v] \quad \textit{if } s(r)(l) = \langle \lambda\, x.\, e \rangle \tag{4.5}$$

$$s, \texttt{copy}\,[r_1, r_2]\,(r_1, l_1) \rightarrowtail s + \{r_2 \mapsto s(r_2) + \{l_2 \mapsto \langle c \rangle\}\}, (r_2, l_2) \quad \textit{if } s(r_1)(l_1) = \langle c \rangle \textit{ and } l_2 \notin \textit{Dom}\,(s(r_2)) \tag{4.6}$$

$$s, \texttt{ref}\ v\ \texttt{at}\ r \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v \rangle\}\}, (r, l) \quad \textit{if } l \notin \textit{Dom}\,(s(r)) \tag{4.7}$$

$$s, \texttt{!}\,(r, l) \rightarrowtail s, v \quad \textit{if } s(r)(l) = \langle \texttt{ref}\ v \rangle \tag{4.8}$$

$$s, (r, l)\ \texttt{:=}\ v \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle \texttt{ref}\ v \rangle\}\}, v \tag{4.9}$$

Reductions in Context:

$$\frac{s, e \rightarrowtail s', e'}{s, \texttt{region}\ r\ \texttt{in}\ e \rightarrowtail s', \texttt{region}\ r\ \texttt{in}\ e'} \tag{4.10}$$

$$\frac{s, e_1 \rightarrowtail s', e_1'}{s, e_1\ @\ e_2 \rightarrowtail s', e_1'\ @\ e_2} \tag{4.11}$$

$$\frac{s, e \rightarrowtail s', e'}{s, v\ @\ e \rightarrowtail s', v\ @\ e'} \tag{4.12}$$

$$\frac{s, e \rightarrowtail s', e'}{s, \texttt{copy}\,[r_1, r_2]\,e \rightarrowtail s', \texttt{copy}\,[r_1, r_2]\,e'} \tag{4.13}$$

$$\frac{s, e \rightarrowtail s', e'}{s, \texttt{ref}\ e\ \texttt{at}\ r \rightarrowtail s', \texttt{ref}\ e'\ \texttt{at}\ r} \tag{4.14}$$

$$\frac{s, e \rightarrowtail s', e'}{s, \texttt{!}\ e \rightarrowtail s', \texttt{!}\ e'} \tag{4.15}$$

$$\frac{s, e_1 \rightarrowtail s', e_1'}{s, e_1\ \texttt{:=}\ e_2 \rightarrowtail s', e_1'\ \texttt{:=}\ e_2} \tag{4.16}$$

$$\frac{s, e \rightarrowtail s', e'}{s, v\ \texttt{:=}\ e \rightarrowtail s', v\ \texttt{:=}\ e'} \tag{4.17}$$

Figure 4.2: Dynamic semantics of the λ_i^{region}-calculus

and similarly, rule (4.13) imposes a call-by-value primitive copy operator. Finally, the rules (4.14) through (4.17) implement a call-by-value semantics for the destructive store operators.

As usual, we define the relation $\overset{*}{\rightarrowtail}$ as the reflexive and transitive closure of $\rightarrowtail$.

The set of λ_i^{region}-Terms can be divided into *pure term* (terms that do not contain sub-terms of the form $\texttt{region}\, r\, \texttt{in}\, e$) and *intermediate terms* (terms that do contain sub-terms of the form $\texttt{region}\, r\, \texttt{in}\, e$). A λ_i^{region}-Storable term $\langle \lambda\, x.\, e \rangle$ or a λ_i^{region}-Term of the form $\texttt{new}\, \varrho.\, e$ only makes sense if e is pure.

4.2.3 The Static Semantics of the λ_i^{region}-calculus

The type language of the λ_i^{region}-calculus is a little different from that of the λ_{tt}^{region}-calculus since we only consider the monomorphic subset and have first-class references. It is generated by the following grammar:

$$\mathsf{TypeWPlace} \ni \mu ::= (\tau, \rho) \qquad \mathsf{Type} \ni \tau ::= \texttt{int} \mid \mu \xrightarrow{\varphi} \mu \mid \texttt{ref}\, \mu$$

As before, a type with place μ is a pair of a type and a region placeholder. A type τ is either a base type, a function arrow, or a reference type. Since the λ_i^{region}-calculus is monomorphic, we do not need effect variables and hence the arrow effect φ of the arrow $\mu \xrightarrow{\varphi} \mu$ is not tied to an effect variable ϵ. In addition, we type a reference with the type $\texttt{ref}\, \mu$, where the type with place μ is the type of referenced storable. We extend substitutions to work with the new definition of Placeholder, so we have $S_s(r) = r$ for $r \in \mathsf{RegionName}$ and $S_s(\bullet) = \bullet$.

Due to the explicit store with destructive cells, we need some way to type the store. To this end, we define two additional finite functions: a *region type*, $K \in \mathsf{Location} \hookrightarrow \mathsf{Type}$, mapping locations to types and a *heap type*, $H \in \mathsf{RegionName} \hookrightarrow (\mathsf{Location} \hookrightarrow \mathsf{Type})$, mapping a region name to a region type, thus providing the typing for locations in the store. Also define $frv\,(H) = \bigcup \{ frv\,(s(r)(l)) \mid r \in Dom\,(s), l \in Dom\,(H(r)) \}$.

The type system for the λ_i^{region}-calculus is built from the following four judgments:

$H, TE \vdash_i^e e : \mu, \varphi$	expression typing
$H \vdash_i s, e : \mu, \varphi$	configuration typing
$H \vdash_i^h s$	heap typing
$H \vdash_i^s w : \tau$	storable typing

An expression type judgment $H, TE \vdash_i^e e : \mu, \varphi$ gives an expression e the type with place μ and effect φ assuming a type environment TE and heap type H. The expression type rules for the λ_i^{region}-calculus are listed in Figure 4.3. Expression typings are a simple extension to the type judgment for λ_{tt}^{region}-Terms (see Figure 2.6 on page 27). In addition to the $\vdash_t^t$ typing judgment, expression typings "propagate" the heap type H, which is only used to provide a typing for live pointers in rule *(I-Pointer)*. A dead pointer can assume any type, due to rule *(I-Dead)*. The type rules *(I-Ref)*, *(I-Deref)* and *(I-Setref)* are equivalent to the primitive type schemes for reference operations as given by Tofte and Talpin [150, Section 11.1].

$$\textit{(I-Pointer)} \quad \frac{H(r)(l) = \tau}{H,\, TE \vdash_i^e (r, l) : (\tau, r), \emptyset}$$

$$\textit{(I-Dead)} \quad H,\, TE \vdash_i^e (\bullet, l) : (\tau, \bullet), \emptyset$$

$$\textit{(I-Var)} \quad \frac{TE(x) = \mu}{H,\, TE \vdash_i^e x : \mu, \emptyset}$$

$$\textit{(I-Const)} \quad H,\, TE \vdash_i^e c\ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}$$

$$\textit{(I-Lam)} \quad \frac{H,\, TE + \{x \mapsto \mu_2\} \vdash_i^e e : \mu_1, \varphi \qquad \varphi \subseteq \varphi'}{H,\, TE \vdash_i^e \lambda x.\, e\ \mathtt{at}\ \rho : (\mu_2 \xrightarrow{\varphi'} \mu_1, \rho), \{\rho\}}$$

$$\textit{(I-App)} \quad \frac{H,\, TE \vdash_i^e e_1 : (\mu_2 \xrightarrow{\varphi} \mu_1, \rho), \varphi_1 \qquad H,\, TE \vdash_i^e e_2 : \mu_2, \varphi_2}{H,\, TE \vdash_i^e e_1 \mathbin{@} e_2 : \mu_2, \varphi_1 \cup \varphi_2 \cup \varphi \cup \{\rho\}}$$

$$\textit{(I-Copy)} \quad \frac{H,\, TE \vdash_i^e e : (\mathtt{int}, \rho_1), \varphi}{H,\, TE \vdash_i^e \mathtt{copy}\,[\rho_1, \rho_2]\, e : (\mathtt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}}$$

$$\textit{(I-Reglam)} \quad \frac{H,\, TE \vdash_i^e e : \mu, \varphi \qquad \varrho \notin \mathit{frv}\,(TE, \mu)}{H,\, TE \vdash_i^e \mathtt{new}\, \varrho.\, e : \mu, \varphi \setminus \{\varrho\}}$$

$$\textit{(I-Region)} \quad \frac{H,\, TE \vdash_i^e e : \mu, \varphi \qquad r \notin \mathit{frv}\,(TE|_{\mathit{fv}\,(e)}, \mu)}{H,\, TE \vdash_i^e \mathtt{region}\, r\, \mathtt{in}\, e : \mu, \varphi \setminus \{r\}}$$

$$\textit{(I-Ref)} \quad \frac{H,\, TE \vdash_i^e e : \mu, \varphi}{H,\, TE \vdash_i^e \mathtt{ref}\ e\ \mathtt{at}\ \rho : (\mathtt{ref}\ \mu, \rho), \varphi \cup \{\rho\}}$$

$$\textit{(I-Deref)} \quad \frac{H,\, TE \vdash_i^e e : (\mathtt{ref}\ \mu, \rho), \varphi}{H,\, TE \vdash_i^e\ !\, e : \mu, \varphi \cup \{\rho\}}$$

$$\textit{(I-Setref)} \quad \frac{H,\, TE \vdash_i^e e_1 : (\mathtt{ref}\ \mu, \rho), \varphi_1 \qquad H,\, TE \vdash_i^e e_2 : \mu, \varphi_2}{H,\, TE \vdash_i^e e_1 := e_2 : \mu, \varphi_1 \cup \varphi_2 \cup \{\rho\}}$$

Figure 4.3: Expression typing for the λ_i^{region}-calculus

$$\textit{(IS-Const)} \qquad H \vdash_i^s \langle c \rangle : \texttt{int}$$

$$\textit{(IS-Lam)} \qquad \frac{H, \{x \mapsto \mu_2\} \vdash_i^e e : \mu_1, \varphi \qquad \varphi \subseteq \varphi'}{H \vdash_i^s \langle \lambda\, x.\, e \rangle : \mu_2 \xrightarrow{\varphi'} \mu_1}$$

$$\textit{(IS-Ref)} \qquad \frac{H, \{\,\} \vdash_i^e v : \mu, \emptyset}{H \vdash_i^s \langle \texttt{ref}\ v \rangle : \texttt{ref}\ \mu}$$

Figure 4.4: Storable typing for the λ_i^{region}-calculus

There is just one rule for a configuration typing:

$$\textit{(I-Conf)} \qquad \frac{H \vdash_i^h s \qquad H, \{\,\} \vdash_i^e e : \mu, \varphi}{H \vdash_i s, e : \mu, \varphi}$$

A configuration s, e has type with place μ and effect φ under heap type H, if H is a valid heap type for s and e is a closed expression typeable with heap type H.

There is also just one rule for a heap typing:

$$\textit{(I-Heap)} \qquad \frac{\begin{array}{c} Dom\,(H) = Dom\,(s) \\ (\forall r \in Dom\,(H))\ Dom\,(H(r)) = Dom\,(s(r)) \\ (\forall r \in Dom\,(H))\ (\forall l \in Dom\,(H(r)))\ H \vdash_i^s s(r)(l) : H(r)(l) \end{array}}{H \vdash_i^h s}$$

That is, the names of the regions in the heap type and the actual store must agree. Further, the domains of all regions must agree and, for each region r and each location l in r, H must provide a valid type for the contents of $s(r)(l)$. The latter is asserted using a storable typing as formulated in Figure 4.4. Since all storables are closed, a storable typing simply refers to the expression typing in the empty environment. The rules are self explanatory.

4.3 Type Soundness of the λ_i^{region}-calculus

Analogous to the syntactic type soundness proof for the λ_s^{region}-calculus, we prove type soundness for the λ_i^{region}-calculus following the schema from Section 2.2.3.

4.3.1 Auxiliary Results

Before we prove type preservation and progress, we need several technical lemmas.

Values have an empty effect:

Lemma 4.3.1 *If H, $TE \vdash_i^e v : \mu, \varphi$ then $\varphi = \emptyset$.*

We have a weakening lemma for type environments:

Lemma 4.3.2 *If* $H, TE \vdash_i^e e : \mu, \varphi$ *then* $H, TE' + TE \vdash_i^e e : \mu, \varphi$.

We have a weakening lemma for heap types as well, in that we can arbitrarily extend a heap type in an expression typing judgment:

Lemma 4.3.3 *If* $H, TE \vdash_i^e e : \mu, \varphi$ *and* $H \leq H'$ *then* $H', TE \vdash_i^e e : \mu, \varphi$.

In the other direction, it is possible to strengthen a judgment by reducing the heap type with a region that is not "relevant" anymore:

Lemma 4.3.4 *If* $H, TE \vdash_i^e e : \mu, \varphi$ *and* $r \in Dom(H) \setminus frv(H, TE, e, \mu, \varphi)$ *then* $H - r, TE \vdash_i^e e : \mu, \varphi$.

By analogy to the type substitution Lemma 3.3.3 on page 34 for the λ_s^{region}-calculus, we have a region substitution lemma for the λ_i^{region}-calculus:

Lemma 4.3.5 (Region Substitution Lemma) *By abuse of notation, let* S_r *be a substitution that either maps a region name* r *to* $\bullet$ *(*i.e., $S_r^\bullet = \{r \mapsto \bullet\}$*) or that maps a region variable to a region name (*i.e., $S_r^r = \{\varrho \mapsto r\}$*).*

1. *If* $H, TE \vdash_i^e e : \mu, \varphi$ *then* $S_r(H), S_r(TE) \vdash_i^e S_r(e) : S_r(\mu), S_r(\varphi)$.
2. *If* $H \vdash_i^h s$ *then* $S_r(H) \vdash_i^h S_r(s)$.
3. *If* $H \vdash_i s, e : \mu, \varphi$ *then* $S_r(H) \vdash_i S_r(s), S_r(e) : S_r(\mu), S_r(\varphi)$.
4. *If* $H \vdash_i^s w : \tau$ *then* $S_r(H) \vdash_i^s S_r(w) : S_r(\tau)$.

Proof. By simultaneous induction on the derivations. The proof of the case for $S_r^\bullet$ relies crucially on the type rule *(I-Dead)*. □

The Canonical Forms Lemma is formulated on a configuration typing:

Lemma 4.3.6 (Canonical Forms Lemma) *Suppose* $H \vdash_i s, (r, l) : \mu, \emptyset$, *then*

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

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

Case 1**.** By rule *(I-Conf)*, we have that $H \vdash_i^h s$. By rule *(I-Pointer)*, it must be that $H(r)(l) = \texttt{int}$. Rule *(I-Heap)* gives that $H \vdash_i^s s(r)(l) : H(r)(l)$, that is, $H \vdash_i^s s(r)(l) : \texttt{int}$. The only storable that fulfills this requirement is of the form $\langle c \rangle$: by rule *(IS-Const)* $H \vdash_i^s \langle c \rangle : \texttt{int}$. Hence, $s(r)(l) = \langle c \rangle$.

Cases 2 and 3**.** Analogous to case 1.

□

Since the λ_i^{region}-calculus is monomorphic, we only have one value substitution lemma, which is proven by induction on the type derivation:

Lemma 4.3.7 (Substitution Lemma)
Suppose $H, TE + \{x \mapsto \mu'\} \vdash_i^e e : \mu, \varphi$ *and* $H, TE \vdash_i^e v : \mu', \emptyset$. *Then* $H, TE \vdash_i^e e[x \mapsto v] : \mu, \varphi$.

The following lemma is an adaptation of Lemma 2.3.1 on page 23, re-casted to the small-step semantics of the λ_i^{region}-calculus:

Lemma 4.3.8 *Suppose* e_1 *is a pure term (see page 47). Then* $s_1, e_1 \overset{*}{\rightarrowtail} s_2, e_2$ *implies* $s_1 \leq s_2$. *Moreover, if* $e_2 \in \lambda_i^{region}$-Value *then* $Dom(s_1) = Dom(s_2)$.

4.3.2 Type Preservation

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

Proposition 4.3.9 (Type Preservation)
Suppose that for $H \vdash_i s, e : \mu, \varphi$ *we have that* $frv(\mu) \subseteq Dom(H)$ *and* $s, e \mapsto s', e'$. *Then there exist* H' *and* φ' *such that* $H' \vdash_i s', e' : \mu, \varphi'$ *where* $\varphi' \subseteq \varphi$ *and* $frv(\mu) \subseteq Dom(H')$.

The requirement that $frv(\mu) \subseteq Dom(H)$ guarantees that all regions in use are also allocated. Alternatively, we could demand that $frv(\mu) \subseteq Dom(s)$ since $H \vdash_i^h s$ holds.

Proof. By rule induction on the reduction relation $\rightarrowtail$. We only consider the interesting cases:

Case $s, \mathtt{new}\,\varrho.\,e \rightarrowtail s + \{r \mapsto \{\,\}\}, \mathtt{region}\,r\,\mathtt{in}\,e[\varrho \mapsto r]$ *if* $r \notin Dom(s)$**.**
Since $H \vdash_i s, \mathtt{new}\,\varrho.\,e : \mu, \varphi$ is assumed it must be that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e \mathtt{new}\,\varrho.\,e : \mu, \varphi$, by rule *(I-Conf)*. By rule *(I-Reglam)*, it must be that $H, \{\,\} \vdash_i^e e : \mu, \varphi'$ and $\varrho \notin frv(\mu)$ and $\varphi = \varphi' \setminus \{\varrho\}$. By Lemma 4.3.5, $H[\varrho \mapsto r], \{\,\}[\varrho \mapsto r] \vdash_i^e e[\varrho \mapsto r] : \mu[\varrho \mapsto r], \varphi'[\varrho \mapsto r]$. Since $Dom(H) = Dom(s)$, by rule *(I-Heap)* for $H \vdash_i^h s$, it follows that $r \notin Dom(H)$. Therefore, $r \notin frv(\mu)$ and because $\varrho \notin frv(\mu)$, the above judgment is equivalent to $H, \{\,\} \vdash_i^e e[\varrho \mapsto r] : \mu, \varphi'[\varrho \mapsto r]$ where r does not occur free in μ. These are exactly the assumptions for rule *(I-Region)*, hence $H, \{\,\} \vdash_i^e \mathtt{region}\,r\,\mathtt{in}\,e : \mu, \varphi$. Let $H' = H + \{r \mapsto \{\,\}\}$. Since $H \leq H'$, Lemma 4.3.3 yields that $H', \{\,\} \vdash_i^e \mathtt{region}\,r\,\mathtt{in}\,e : \mu, \varphi$. Since $H' \vdash_i^h s + \{r \mapsto \{\,\}\}$, rule *(I-Conf)* yields $H' \vdash_i s + \{r \mapsto \{\,\}\}, \mathtt{region}\,r\,\mathtt{in}\,e : \mu, \varphi$.

Case $s, \mathtt{region}\,r\,\mathtt{in}\,v \rightarrowtail s[r \mapsto \bullet] - r, v[r \mapsto \bullet]$**.**
Since $H \vdash_i s, \mathtt{region}\,r\,\mathtt{in}\,v : \mu, \varphi$ is assumed it must be that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e \mathtt{region}\,r\,\mathtt{in}\,v : \mu, \varphi$, by rule *(I-Conf)*. By rule *(I-Region)*, we have that $H, \{\,\} \vdash_i^e v : \mu, \varphi'$, $r \notin frv(\mu)$ and $\varphi = \varphi' \setminus \{r\}$. By Lemma 4.3.5, $H[r \mapsto \bullet], \{\,\}[r \mapsto \bullet] \vdash_i^e v[r \mapsto \bullet] : \mu[r \mapsto \bullet], \varphi'[r \mapsto \bullet]$ also holds. By Lemma 4.3.1, $\varphi' = \emptyset$.

Taken together with the assumption on the non-occurrence of r in μ it holds that $H[r \mapsto \bullet], \{\,\} \vdash_i^e v[r \mapsto \bullet] : \mu, \emptyset$. As $r \notin \mathit{frv}\,(H[r \mapsto \bullet], v[r \mapsto \bullet], \mu)$, Lemma 4.3.4 implies that $H', \{\,\} \vdash_i^e v[r \mapsto \bullet] : \mu, \emptyset$ where $H' = H[r \mapsto \bullet] - r$. Therefore, because $H' \vdash_i^h s[r \mapsto \bullet] - r$, rule *(I-Conf)* yields that $H' \vdash_i s[r \mapsto \bullet] - r, v[r \mapsto \bullet] : \mu, \emptyset$.

Case $s, \lambda\, x.\, e\ \mathtt{at}\ r \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}, (r, l)$ *if* $l \notin \mathit{Dom}\,(s(r))$**.**
By assumption, $H \vdash_i s, \lambda\, x.\, e\ \mathtt{at}\ r : \mu, \varphi$. By rule *(I-Conf)* it must be that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e \lambda\, x.\, e\ \mathtt{at}\ r : \mu, \varphi$. Rule *(I-Lam)* gives that $\mu = (\mu_1 \xrightarrow{\varphi'} \mu_2, r)$ and $\varphi = \{r\}$. Let $H' = H + \{r \mapsto H(r) + \{l \mapsto \mu_1 \xrightarrow{\varphi'} \mu_2\}\}$ and $s' = s + \{r \mapsto s(r) + \{l \mapsto \langle \lambda\, x.\, e \rangle\}\}$. Since $\lambda\, x.\, e$ is closed and $l \notin \mathit{Dom}\,(s(r)) = \mathit{Dom}\,(H(r))$, rules *(IS-Lam)* and *(I-Heap)* yield $H' \vdash_i^h s'$. Since $H'(r)(l) = \mu_1 \xrightarrow{\varphi'} \mu_2$ rule *(I-Pointer)* yields $H', \{\,\} \vdash_i^e (r, l) : (\mu_1 \xrightarrow{\varphi'} \mu_2, r), \emptyset$. Applying rule *(I-Conf)* yields $H' \vdash_i s', (r, l) : (\mu_1 \xrightarrow{\varphi'} \mu_2, r), \emptyset$. This suffices the claim, since $\emptyset \subseteq \{r\}$.

Case $s, (r, l)\,@\,v \rightarrowtail s, e[x \mapsto v]$ *if* $s(r)(l) = \langle \lambda\, x.\, e \rangle$**.**
By assumption, $H \vdash_i s, (r, l)\,@\,v : \mu, \varphi$. By rule *(I-Conf)* it must be that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e (r, l)\,@\,v : \mu, \varphi$. From rule *(I-App)*, we have that $H, \{\,\} \vdash_i^e (r, l) : (\mu_2 \xrightarrow{\varphi'} \mu, r), \varphi_1$, that $H, \{\,\} \vdash_i^e v : \mu_2, \varphi_2$ and also that $\varphi = \varphi_1 \cup \varphi_2 \cup \varphi' \cup \{r\}$. So, from rule *(I-Pointer)*, it must be that $H(r)(l) = \mu_2 \xrightarrow{\varphi'} \mu$ (this follows from *(I-Heap)*, too) and $\varphi_1 = \emptyset$. By rule *(I-Heap)*, it must be that $H \vdash_i^s s(r)(l) : H(r)(l)$. But this means that $H \vdash_i^s \langle \lambda\, x.\, e \rangle : \mu_2 \xrightarrow{\varphi'} \mu$. Therefore, by rule *(IS-Lam)*, we have that $H, \{x \mapsto \mu_2\} \vdash_i^e e : \mu, \varphi''$, for some $\varphi'' \subseteq \varphi'$. Lemma 4.3.7 then gives $H, \{\,\} \vdash_i^e e[x \mapsto v] : \mu, \varphi''$, and hence, by rule *(I-Conf)*, $H \vdash_i s, e[x \mapsto v] : \mu, \varphi''$. This suffices for the claim because $\varphi'' \subseteq \varphi' \subseteq \varphi$.

Case $s, !\,(r, l) \rightarrowtail s, v$ *if* $s(r)(l) = \langle \mathtt{ref}\ v \rangle$**.**
From the assumption $H \vdash_i s, !\,(r, l) : \mu, \varphi$. So, by rule *(I-Conf)*, it must be that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e\ !\,(r, l) : \mu, \varphi$. From rule *(I-Deref)*, we have that $H, \{\,\} \vdash_i^e (r, l) : (\mathtt{ref}\ \mu, r), \varphi'$ and $\varphi = \varphi' \cup \{\rho\}$. Rule *(I-Pointer)* gives that $\rho = r$. Rule *(I-Heap)* gives $H \vdash_i^s s(r)(l) : H(r)(l)$, that is, $H \vdash_i^s \langle \mathtt{ref}\ v \rangle : \mathtt{ref}\ \mu$. Now, rule *(IS-Ref)* says that $H, \{\,\} \vdash_i^e v : \mu, \emptyset$ and applying rule *(I-Conf)* yields $H \vdash_i s, v : \mu, \emptyset$. Conclude by observing that $\emptyset \subseteq \varphi$.

Case $s, (r, l) := v \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle \mathtt{ref}\ v \rangle\}\}, v$ *if* $s(r)(l) = \langle \mathtt{ref}\ v' \rangle$**.**
By assumption $H \vdash_i s, (r, l) := v : \mu, \varphi$. Again, by rule *(I-Conf)*, it must be that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e (r, l) := v : \mu, \varphi$. By rule *(I-Setref)*, it holds that $H, \{\,\} \vdash_i^e (r, l) : (\mathtt{ref}\ \mu, r), \varphi'$, that $H, \{\,\} \vdash_i^e v : \mu, \varphi''$ and also that $\varphi = \varphi' \cup \varphi'' \cup \{r\}$.

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

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

Case $\dfrac{s, e \rightarrowtail s', e'}{s, \texttt{region}\, r\, \texttt{in}\, e \rightarrowtail s', \texttt{region}\, r\, \texttt{in}\, e'}$.

By assumption, $H \vdash_i s, \texttt{region}\, r\, \texttt{in}\, e : \mu, \varphi$. So, from rule *(I-Conf)*, we have that $H \vdash_i^h s$ and $H, \{\,\} \vdash_i^e \texttt{region}\, r\, \texttt{in}\, e : \mu, \varphi$. By rule *(I-Region)*, it must be that $H, \{\,\} \vdash_i^e e : \mu, \varphi'$ and $\varphi = \varphi' \setminus \{r\}$.

By induction, there exist some H'' and φ'' such that $H'' \vdash_i s', e' : \mu, \varphi''$ where $\varphi'' \subseteq \varphi'$. Rule *(I-Conf)* requires that $H'' \vdash_i^h s'$ and $H'', \{\,\} \vdash_i^e e' : \mu, \varphi''$.

Rule *(I-Region)* is applicable because μ has not changed. It yields that $H'', \{\,\} \vdash_i^e \texttt{region}\, r\, \texttt{in}\, e' : \mu, \varphi'' \setminus \{r\}$. Finally, rule *(I-Conf)* yields $H'' \vdash_i s', \texttt{region}\, r\, \texttt{in}\, e' : \mu, \varphi'' \setminus \{r\}$. This yields the claim because $\varphi'' \setminus \{r\} \subseteq \varphi' \setminus \{r\} = \varphi$.

Remaining Cases. Proven by simple appeal to the induction hypothesis.

□

4.3.3 Progress

The progress property states that a well-typed configuration s, e with clean effect cannot be further reduced when e is a value, or has a transition into a new configuration.

Proposition 4.3.10 (Progress) *Suppose $H \vdash_i s, e : \mu, \varphi$ and $Clean\,(\varphi)$. Then either*

1. *e is a λ_i^{region}-*Value*; or*
2. *there exists a configuration s', e' such that $s, e \rightarrowtail s', e'$.*

Recall that the assumption $Clean\,(\varphi)$ means that $frv\,(\varphi) \subseteq \mathsf{RegionName}$ in this chapter. This has two implications:

1. $\bullet \notin \varphi$, meaning that deallocated regions are not accessed, and
2. $frv\,(\varphi) \cap \mathsf{RegionVar} = \emptyset$, meaning that regions are not accessed before they are allocated.

Proof. The proof is, as usual, a rule induction on the derivation for $H \vdash_i s, e : \mu, \varphi$. In other words, we have $H \vdash_i^h s$ and do the rule induction on $H, \{\,\} \vdash_i^e e : \mu, \varphi$. We only consider some non-trivial cases:

Case $H, \{\,\} \vdash_i^e \lambda\, x.\, e\ \texttt{at}\, \rho : (\mu_2 \xrightarrow{\varphi'} \mu_1, \rho), \{\rho\}$. By assumption we have the $\rho = r$ and hence we can reduce with rule (4.4), applying Item 2.

Case $H, \{\,\} \vdash_i^e e_1 \,@\, e_2 : \mu_2, \varphi_1 \cup \varphi_2 \cup \varphi \cup \{\rho\}$.

By type rule *(I-App)*, we have that $H, \{\,\} \vdash_i^e e_1 : (\mu_2 \xrightarrow{\varphi'} \mu, \rho), \varphi_1$, that $H, \{\,\} \vdash_i^e e_2 : \mu_2, \varphi_2$, and that $\varphi = \varphi_1 \cup \varphi_2 \cup \varphi' \cup \{\rho\}$, where $\rho = r$ since $Clean\,(\varphi)$. If e_1 is not a value then we apply rule *(I-Conf)* to obtain $H \vdash_i s, e_1 : (\mu_2 \xrightarrow{\varphi'} \mu, r), \varphi_1$ where

Clean (φ_1). By induction, e_1 is either a value (which contradicts our assumption) or there exist an s' and e_1' such that $s, e_1 \rightarrowtail s', e_1'$. By reduction rule (4.11), $s, e_1 \mathbin{@} e_2 \rightarrowtail s', e_1' \mathbin{@} e_2$ holds, so Item 2 applies.

On the other hand, if e_1 is a value then application of rule *(I-Conf)* yields $H \vdash_i s, e_2 : \mu_2, \varphi_2$ with *Clean* (φ_2). Applying the induction hypothesis on this judgment yields two cases:

1. If $s, e_2 \rightarrowtail s', e_2'$ then, by reduction rule (4.12), $s, v_1 \mathbin{@} e_2 \rightarrowtail s', v_1 \mathbin{@} e_2'$, which satisfies Item 2.
2. Otherwise, we have that e_2 is a λ_i^{region}-Value, say v_2. By Lemma 4.3.6, $e_1 = (r, l)$ and $s(r)(l) = \langle \lambda\, x.\, e \rangle$, for some x and e. Hence, $s, (r, l) \mathbin{@} v_2$ is reducible with reduction rule (4.5).

Case $H, \{\,\} \vdash_i^e \texttt{ref}\ e\ \texttt{at}\ \rho : (\texttt{ref}\ \mu, \rho), \varphi \cup \{\rho\}$. Since *Clean* $(\varphi \cup \{\rho\})$, we have that $\rho = r$. Hence, we conclude that $s, \texttt{ref}\ e\ \texttt{at}\ r$ can be reduced with rule (4.7).

Case $H, \{\,\} \vdash_i^e\ !\, e : \mu, \varphi \cup \{\rho\}$. By type rules *(I-Deref)*, it must be that $H, \{\,\} \vdash_i^e e : \mu', \varphi'$ where $\mu' = (\texttt{ref}\ \mu, \rho)$ with $\varphi = \varphi' \cup \{\rho\}$ and $\rho = r$ since *Clean* $(\varphi \cup \{\rho\})$. Therefore, by rule *(I-Conf)*, it follows that $H \vdash_i s, e : \mu', \varphi'$ with *Clean* (φ').

By induction, we either have that e is a value or it can be reduced. Whenever e is a value then, by the canonical forms Lemma 4.3.6, it must be that $e = (r_1, l)$ such that $s(r_1)(l) = \langle \texttt{ref}\ v \rangle$. Hence, $s, !\,(r_1, l)$ is a redex for reduction rule (4.8) and Item 2 applies.

Alternatively, we have the reduction $s, e \rightarrowtail s', e'$, from which follows by rule (4.15) that $s, !\, e \rightarrowtail s', !\, e'$, proving the case.

Case $H, \{\,\} \vdash_i^e e_1 := e_2 : \mu, \varphi_1 \cup \varphi_2 \cup \{\rho\}$. By rule *(I-Setref)*, we have that $H, \{\,\} \vdash_i^e e_1 : (\texttt{ref}\ \mu, \rho), \varphi_1$ and $H, \{\,\} \vdash_i^e e_2 : \mu, \varphi_2$, where $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$. Since *Clean* (φ), we must have $\rho = r$.

If e_1 is not a value then applying rule *(I-Conf)* yields $H \vdash_i s, e_1 : (\texttt{ref}\ \mu, r), \varphi_1$ with *Clean* (φ_1). By induction, e_1 is either a value (contradiction), or there exist an s' and e_1' such that $s, e_1 \rightarrowtail s', e_1'$. Hence, we can apply reduction rule (4.16), which yields $s, e_1 := e_2 \rightarrowtail s', e_1' := e_2$.

If e_1 is a value v_1, then applying rule *(I-Conf)* yields $H \vdash_i s, e_2 : \mu, \varphi_2$ with *Clean* (φ_2). We can now apply the induction hypothesis, to obtain two cases:

1. If e_2 is a value v_2, then by Lemma 4.3.6, it holds that $e_1 = (r, l)$ and $s(r)(l) = \langle \texttt{ref}\ v \rangle$, for some v. Hence, $s, (r, l) := v_2$ is a redex for rule (4.9)
2. Whenever $s, e_2 \rightarrowtail s', e_2'$ then, by reduction rule (4.17), $s, v_1 := e_2 \rightarrowtail s', v_1 := e_2'$, which satisfies Item 2.

Case $H, \{\,\} \vdash_i^e \texttt{region}\ r\ \texttt{in}\ e : \mu, \varphi \setminus \{r\}$. By type rule *(I-Region)*, we have that $H, \{\,\} \vdash_i^e e : \mu, \varphi'$ where $\varphi' \subseteq \varphi \cup \{r\}$. Since $r \in$ RegionName, we have immediately *Clean* (φ'). By rule *(I-Conf)*, it follows that $H \vdash_i s, e : \mu, \varphi'$.

By induction, either e is a value or it can be reduced. If e is a value then $s, \texttt{region}\, r\, \texttt{in}\, e$ is a redex for reduction rule (4.2). If we have the reduction $s, e \rightarrowtail s', e'$ then, by context rule (4.10), the reduction $s, \texttt{region}\, r\, \texttt{in}\, e \rightarrowtail s', \texttt{region}\, r\, \texttt{in}\, e'$ can be made.

□

4.3.4 Type Soundness

We can now state the type soundness theorem, which is a consequence of the type preservation and progress propositions:

Theorem 4.3.11 (Type Soundness of the λ_i^{region}-calculus)
Suppose for a term $e \in \lambda_i^{region}$-Term that $H \vdash_i s, e : \mu, \varphi$ such that $frv(\mu) \subseteq Dom(H)$ and $Clean(\varphi)$. Then, either

1. *there exist a value $v \in \lambda_i^{region}$-Value, a heap type H' and a store s' such that $s, e \overset{*}{\rightarrowtail} s', v$ and $H' \vdash_i s', v : \mu, \emptyset$; or*

2. *for each s', e' with $s, e \overset{*}{\rightarrowtail} s', e'$, there exist a configuration s'', e'' such that $s', e' \rightarrowtail s'', e''$.*

Proof. Take an arbitrary configuration s', e' such that $s, e \overset{*}{\rightarrowtail} s', e'$. We prove by induction on the length of this reduction and have two cases:

- Suppose $e' = e$ (and $s = s'$). Then by Progress (Proposition 4.3.10), we either have that e is a value v. In that case Item 1 applies due to Lemma 4.3.1 and the assumption that $H \vdash_i s, e : \mu, \varphi$. Otherwise, Progress says that $s', e' \rightarrowtail s'', e''$ and Item 2 applies.

- Suppose $s, e \rightarrowtail s'', e''$ and $s'', e'' \overset{*}{\rightarrowtail} e'$. By Type Preservation (Proposition 4.3.9), we have that $H'' \vdash_i s'', e'' : \mu, \varphi''$ for some H'' and φ'', where $\varphi'' \subseteq \varphi$ and $frv(\mu) \subseteq Dom(H'')$. Hence, we also have that $Clean(\varphi'')$ and we can apply the induction hypothesis to obtain the result.

□

4.4 Relating the store-based Region Calculi

Until now, we have glossed over the fact that the newly introduced region calculi have a different semantics as the original big-step presentation of the λ_{tt}^{region}-calculus. That these calculi are eventually the *same* is not obvious, even though the two store-based presentations have a lot in common. In fact, we are faced with the technical difficulty of relating a big-step operational semantics with a small-step operational semantics, which

we handle in this section. In Section 4.5, we also relate the two small-step semantics to each other.

For conciseness and brevity, we only handle a pure subset of the λ_i^{region}-calculus, which we will henceforth refer to as the λ_{pi}^{region}-calculus. Therefore, on the other side we only consider an equivalent subset of the λ_{tt}^{region}-calculus (see Section 2.3), which, henceforth, we refer to as the $\lambda_{tt\text{-}m}^{region}$-calculus. Both calculi are monomorphic, exclude operations on references, the primitive `copy`-operator as well as `let`-bindings.

4.4.1 A Greatest Relation

The key technical issue in relating the λ_{pi}^{region}-calculus to the $\lambda_{tt\text{-}m}^{region}$-calculus is defining an appropriate relation. First, we define the following sets:

$$\begin{array}{lcl}
\mathsf{QRel} & = & \mathcal{P}(\lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Store} \times \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{ValueEnv} \times \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{RegionEnv} \times \\
 & & \quad \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Term} \times \lambda_{pi}^{region}\text{-}\mathsf{Store} \times \lambda_{pi}^{region}\text{-}\mathsf{Term}) \\
\mathsf{QRelV} & = & \mathcal{P}(\lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Store} \times \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Value} \times \lambda_{pi}^{region}\text{-}\mathsf{Store} \times \lambda_{pi}^{region}\text{-}\mathsf{Value}) \\
\mathsf{QRelS} & = & \mathcal{P}(\lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Store} \times \lambda_{pi}^{region}\text{-}\mathsf{Store})
\end{array}$$

We call a quadruple

$$(S, VE, RE, e) \in (\lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Store} \times \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{ValueEnv} \times \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{RegionEnv} \times \lambda_{tt\text{-}m}^{region}\text{-}\mathsf{Term})$$

a $\lambda_{tt\text{-}m}^{region}$-configuration. With the equivalences defined in Figure 4.5, define the relation $\mathcal{Q} \in \mathsf{QRel}$, relating $\lambda_{tt\text{-}m}^{region}$-configurations to λ_{pi}^{region}-configurations, an auxiliary relation $\mathcal{Q}_v \in \mathsf{QRelV}$, relating a $\lambda_{tt\text{-}m}^{region}$-$\mathsf{Value}$ to a λ_{pi}^{region}-Value, and an auxiliary relation $\mathcal{Q}_s \in \mathsf{QRelS}$ relating a $\lambda_{tt\text{-}m}^{region}$-$\mathsf{Store}$ to a λ_{pi}^{region}-Store. Observe that the auxiliary definitions for $\mathcal{Q}_s$ and $\mathcal{Q}_v$ can be eliminated by expanding them in the definition of $\mathcal{Q}$.

To show that the relation $\mathcal{Q}$ is well-defined, interpret the equivalences from left to right as the defining clauses of a functional $\mathrm{Q} : \mathsf{QRel} \to \mathsf{QRel}$. For example, in the case of function application the definition is as follows:

$$\begin{array}{lcl}
\mathrm{Q}(\mathcal{Q}') & = & \ldots \\
 & & \cup \quad \{(S, VE, RE, e_1 \mathbin{@} e_2, s, e') \mid e' \equiv (e_1' \mathbin{@} e_2') \wedge \mathcal{Q}_s'(S, s) \wedge \\
 & & \qquad\qquad\qquad\qquad \mathcal{Q}'(S, VE, RE, e_1, s, e_1') \wedge \\
 & & \qquad\qquad\qquad\qquad \mathcal{Q}'(S, VE, RE, e_2, s, e_2')\} \\
 & & \cup \ldots
\end{array}$$

It is easy to verify that the functional Q is monotone in the complete lattice $(\mathsf{QRel}, \subseteq)$, so the existence of fixpoints is guaranteed by the Knaster-Tarski fixpoint theorem (see Theorem 2.2.1 on page 14).

It remains to be decided which fixpoint we are interested in. First, note that neither the empty relation, nor the full relation are a fixpoint, so we have that $\mathcal{Q}$ is not trivial. Since we are only dealing with the λ_{pi}^{region}-calculus, a least fixpoint would be enough, because it can be argued that $\mathcal{Q}$ is well-founded, using a similar argument as Calcagno [17]. However, since it is desirable to develop the theory in a more general way, we want to define the relation to work for the full λ_i^{region}-calculus with updatable references. But

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

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

$$\begin{array}{lll}\mathcal{Q}(S,\, VE, RE, c\ \mathtt{at}\ \varrho, s, e') & \Leftrightarrow & (e' \equiv c\ \mathtt{at}\ \varrho \wedge \mathcal{Q}_s(S,s) \wedge \varrho \notin Dom\,(R)) \quad \vee \\ & & (e' \equiv c\ \mathtt{at}\ r \wedge \mathcal{Q}_s(S,s) \wedge RE(\varrho) = r) \quad \vee \\ & & (e' \equiv c\ \mathtt{at}\ \bullet \wedge \mathcal{Q}_s(S,s))\end{array}$$

$$\begin{array}{lll}\mathcal{Q}(S,\, VE, RE, x, s, e') & \Leftrightarrow & (e' \equiv v \wedge \mathcal{Q}_v(S,\, VE(x), s, v)) \quad \vee \\ & & (e' \equiv x \wedge \mathcal{Q}_s(S,s) \wedge x \notin Dom\,(VE))\end{array}$$

$$\begin{array}{lll}\mathcal{Q}(S,\, VE, RE, \lambda\, x.\, e\ \mathtt{at}\ \varrho, s, e') & \Leftrightarrow & (e' \equiv (\lambda\, x.\, e''\ \mathtt{at}\ \varrho) \wedge \mathcal{Q}_s(S,s) \wedge \\ & & \varrho \notin Dom\,(RE) \wedge \mathcal{Q}(S,\, VE - x, RE, e, s, e'')) \quad \vee \\ & & (e' \equiv (\lambda\, x.\, e''\ \mathtt{at}\ r) \wedge \mathcal{Q}_s(S,s) \wedge \\ & & RE(\varrho) = r \wedge \mathcal{Q}(S,\, VE - x, RE, e, s, e'')) \quad \vee \\ & & (e' \equiv (\lambda\, x.\, e''\ \mathtt{at}\ \bullet) \wedge \mathcal{Q}_s(S,s))\end{array}$$

$$\begin{array}{lll}\mathcal{Q}(S,\, VE, RE, e_1 \mathbin{@} e_2, s, e') & \Leftrightarrow & e' \equiv (e_1' \mathbin{@} e_2') \wedge \mathcal{Q}_s(S,s) \wedge \\ & & \mathcal{Q}(S,\, VE, RE, e_1, s, e_1') \wedge \mathcal{Q}(S,\, VE, RE, e_2, s, e_2')\end{array}$$

$$\mathcal{Q}(S,\, VE, RE, \mathtt{new}\ \varrho.\, e, s, e') \;\Leftrightarrow\; e' \equiv (\mathtt{new}\ \varrho.\, e'') \wedge \mathcal{Q}_s(S,s) \wedge \mathcal{Q}(S,\, VE, RE - \varrho, e, s, e'')$$

Figure 4.5: A relation between $\lambda^{region}_{tt\text{-}m}$-Terms and λ^{region}_{pi}-Terms

destructive updates make it generally possible to have cycles in the store. Since the least fixpoint of Q does not include configurations that have such cycles, we define $\mathcal{Q}$ as the greatest fixpoint of Q.

The reader may find it alarming that we introduce greatest fixpoints, and therefore possibly co-induction, one of the sources of complexity in the original Tofte-Talpin proof. First, note that the λ_{tt}^{region}-calculus does not include destructive updates and hence, the least fixpoint would do as long as we are only interested in relating pure subsets. Secondly, our proofs hardly build on the structure of $\mathcal{Q}$ itself. In fact, it turns out that only one lemma requires a co-inductive argument.

Finally, we have to point out two peculiarities of the relation $\mathcal{Q}$: the reader may have wondered that a $\lambda_{tt\text{-}m}^{region}$-Store and a λ_{pi}^{region}-Store are related by $\mathcal{Q}_s$ as soon as the domains agree. This reflects the idea that values in the stores only need to correspond if a λ_{pi}^{region}-pointer refers to them. Second, the relation $\mathcal{Q}$ does not relate intermediate λ_{pi}^{region}-terms like $\texttt{region}\, r\, \texttt{in}\, e$. This is due to the fact that such terms have no counterpart in the $\lambda_{tt\text{-}m}^{region}$-calculus.

4.4.2 Auxiliary Lemmas

To formulate an equivalence theorem between the $\lambda_{tt\text{-}m}^{region}$-calculus and the λ_{pi}^{region}-calculus, we need several auxiliary lemmas. The first result states that the relation is closed under extension of the stores and is proven by co-induction on the structure of $\mathcal{Q}$.

Lemma 4.4.1 *Suppose $\mathcal{Q}_s(S_2, s_2)$, $S_1 \leq S_2$ and $s_1 \leq s_2$ then*

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

Proof. First, note that item 2 follows immediately from item 1. In order to prove the first item by co-induction on $\mathcal{Q}$, we formulate the above property as a relation $X \in \mathsf{QRel}$:

$$X = \{(S, VE, RE, e, s, e') \mid \mathcal{Q}_s(S, s) \wedge \exists S_1, s_1(S_1 \leq S \wedge s_1 \leq s \wedge \mathcal{Q}(S_1, VE, RE, e, s_1, e'))\}$$

The co-induction principle (see Section 2.2.1) states that if $X \subseteq \mathrm{Q}(X)$, then $X \subseteq \mathcal{Q}$. The latter inclusion obviously proves the claim.

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

$$\begin{aligned}\mathrm{Q}(X) = {} & \ldots \\ & \cup \{(S, VE, RE, e_1 \mathbin{@} e_2, s, e') \mid e' \equiv (e_1' \mathbin{@} e_2') \wedge \mathcal{Q}_s(S, s) \wedge \\ & \qquad \exists S_1, s_1(S_1 \leq S \wedge s_1 \leq s \wedge \\ & \qquad\quad \mathcal{Q}(S_1, VE, RE, e_1, s_1, e_1') \wedge \\ & \qquad\quad \mathcal{Q}(S_1, VE, RE, e_2, s_1, e_2'))\} \\ & \cup \ldots\end{aligned}$$

Now, take $(S, VE, RE, e, s, e') \in X$, then $\mathcal{Q}_s(S, s)$ and $\mathcal{Q}(S_1, VE, RE, e, s_1, e')$ where $S_1 \leq S$ and $s_1 \leq s$ for some S_1 and s_1. Again, we consider the case for function

application, so $e = e_1 \mathbin{@} e_2$. Then, since $\mathcal{Q}(S_1, VE, RE, e_1 \mathbin{@} e_2, s_1, e')$, it must be that $e' = e'_1 \mathbin{@} e'_2$. Moreover, $\mathcal{Q}(S_1, VE, RE, e_1, s_1, e'_1)$ and $\mathcal{Q}(S_1, VE, RE, e_2, s_1, e'_2)$ hold too, so, we can conclude that $(S, VE, RE, e, s, e') \in \mathrm{Q}(X)$. □

Substitution of a value for a variable in the λ^{region}_{pi}-calculus is related to an extension of the type environment in the $\lambda^{region}_{tt\text{-}m}$-calculus.

Lemma 4.4.2 *Suppose* $\mathcal{Q}(S, VE - x, RE, e, s, e')$ *and* $\mathcal{Q}_v(S, v, s, v')$ *then* $\mathcal{Q}(S, VE + \{x \mapsto v\}, RE, e, s, e'[x \mapsto v'])$.

Proof. By induction on the structure of e. The only interesting case is the one for variables:

Case $e \equiv y$**.** So $\mathcal{Q}(S, VE - x, RE, y, s, e')$ and $\mathcal{Q}_v(S, v, s, v')$.

Subcase $x \neq y$**.** Then, either $\mathcal{Q}_v(S, (VE - x)(y), s, e')$ where $e' = v''$. Now, obviously we also have $\mathcal{Q}_v(S, (VE + \{x \mapsto v\})(y), s, v''[x \mapsto v'])$, so $\mathcal{Q}(S, VE + \{x \mapsto v\}, RE, y, s, e'[x \mapsto v'])$.
Alternatively, $\mathcal{Q}(S, VE - x, RE, y, s, y)$, so we have that $y \notin Dom\,(VE - x)$. Then, trivially we also have $y \notin Dom\,(VE + \{x \mapsto v\})$. Hence, $\mathcal{Q}(S, VE + \{x \mapsto v\}, RE, y, s, y[x \mapsto v'])$.

Subcase $x = y$**.** So, $e' = x$ since $x \notin Dom\,(VE - x)$. But $\mathcal{Q}_v(S, v, s, v')$, therefore $\mathcal{Q}_v(S, (VE + \{x \mapsto v\})(y), s, v')$. Hence, $\mathcal{Q}(S, VE + \{x \mapsto v\}, RE, y, s, y[x \mapsto v'])$.

□

The next lemma deals with region allocation.

Lemma 4.4.3 *If* $\mathcal{Q}(S, VE, RE - \varrho, e, s, e')$ *then* $\mathcal{Q}(S + \{r \mapsto \{\,\}\}, VE, RE + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\,\}\}, e'[\varrho \mapsto r])$

Proof. By induction on the structure of e. Note that it is immediate that if $\mathcal{Q}_s(S, s)$, also $\mathcal{Q}_s(S + \{r \mapsto \{\,\}\}, s + \{r \mapsto \{\,\}\})$. We will use this fact implicitly.

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

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

Subcase $VE(x) = v$**.** Hence, $\mathcal{Q}_v(S, v, s, v')$, for some v' where $e' = v'$. Now, the claim holds independent of RE.

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

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

Subcase $e' \equiv c\ \mathtt{at}\ r'$**.** Hence, we have that $(RE' - \varrho)(\varrho') = r'$. Therefore, it must be that $\mathcal{Q}(S + \{r \mapsto \{\ \}\}, VE, RE + \{\varrho \mapsto r\}, c\ \mathtt{at}\ \varrho', s + \{r \mapsto \{\ \}\}, c\ \mathtt{at}\ r')$.

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

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

Case $e_1 @ e_2$**.** Immediate by use of the induction hypothesis.

Case $\mathtt{new}\ \varrho'.\, e$**.** Since $\mathcal{Q}(S, VE, RE - \varrho, \mathtt{new}\ \varrho'.\, e, s, e')$, it must be that $e' = \mathtt{new}\ \varrho'.\, e''$ and $\mathcal{Q}(S, VE, (RE - \varrho) - \varrho', e, s, e'')$.

We can assume without loss of generalization that $\varrho \neq \varrho'$. Since $(RE - \varrho) - \varrho' = (RE - \varrho') - \varrho$, it holds that $\mathcal{Q}(S, VE, (RE - \varrho') - \varrho, e, s, e'')$. By induction, $\mathcal{Q}(S + \{r \mapsto \{\ \}\}, VE, RE - \varrho' + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\ \}\}, e''[\varrho \mapsto r])$. Since $RE - \varrho' + \{\varrho \mapsto r\} = RE + \{\varrho \mapsto r\} - \varrho'$ applying the definition of $\mathcal{Q}$ yields that $\mathcal{Q}(S + \{r \mapsto \{\ \}\}, VE, RE + \{\varrho \mapsto r\}, \mathtt{new}\ \varrho'.\, e, s + \{r \mapsto \{\ \}\}, \mathtt{new}\ \varrho'.\, e''[\varrho \mapsto r])$, which proves the claim.

□

As a final lemma, we prove that region deallocation preserves the relation $\mathcal{Q}$.

Lemma 4.4.4 *The following hold:*

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

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

Item 1**.** By induction on the structure of e:

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

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

Subcase $x \notin \mathit{Dom}\,(VE)$**.** Immediate.

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

Subcase $e' \equiv c\ \mathtt{at}\ \varrho$**.** So, $\varrho \notin \mathit{Dom}\,(RE)$ and $\mathcal{Q}_s(S, s')$. Since $(c\ \mathtt{at}\ \varrho)[r \mapsto \bullet] = c\ \mathtt{at}\ \varrho$, it is immediate that $\mathcal{Q}(S - r, VE, RE, c\ \mathtt{at}\ \varrho, s'[r \mapsto \bullet] - r, c\ \mathtt{at}\ \varrho)$.

Subcase $e' \equiv c\ \mathtt{at}\ r'$**.** So, $RE(\varrho) = r'$ where $r' \in \mathit{Dom}\,(S)$ and $\mathcal{Q}_s(S, s')$. If $r' \neq r$ then the claim holds. If $r' = r$ then $(c\ \mathtt{at}\ r')[r \mapsto \bullet] = c\ \mathtt{at} \bullet$ and the claim holds, too.

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

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

Case $e_1 \mathbin{@} e_2$**.** Simple appeal to the induction hypothesis for e_1 and e_2.

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

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

□

4.4.3 The Equivalence Theorem

The equivalence theorem requires the notion of a *stuck* term (see also [158]).

Definition 4.4.1 *For a λ_{pi}^{region}-configuration s, e, we say that*

- s, e is stuck *iff e is not a value and there does not exist a configuration s', e' such that $s, e \rightarrowtail s', e'$.*
- s, e becomes stuck *iff there exists s', e' such that $s, e \rightarrowtail s', e'$ and s', e' is stuck.*

Now, we can formulate the equivalence theorem:

Theorem 4.4.5 *The following hold:*

1. *Suppose that $S, VE, RE \vdash e \Downarrow v, S'$. For each s, e' such that $\mathcal{Q}(S, VE, RE, e, s, e')$, either s, e' becomes stuck or there exists a configuration s', v' so that $s, e' \overset{*}{\rightarrowtail} s', v'$ and $\mathcal{Q}_v(S', v, s', v')$.*
2. *Suppose $s, e' \overset{*}{\rightarrowtail} s', v'$ and $\mathcal{Q}(S, VE, RE, e, s, e')$. Then the evaluation $S, VE, RE \vdash e \Downarrow v, S'$ exists such that $\mathcal{Q}_v(S', v, s', v')$.*

Part 1 of Theorem 4.4.5 is weaker than we hoped for since a valid big-step evaluation may relate to a stuck small-step configuration. However, the apparent weakness is due to the fact that the theorem deals with an untyped semantics. The problem is that, for some un-typeable expressions, the $\lambda_{tt\text{-}m}^{region}$-calculus semantics is slightly less deterministic than the λ_{pi}^{region}-calculus semantics. Consider the following example:

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

This term definitely gets stuck when evaluated as a λ_{pi}^{region}-Term since the region allocated to ϱ_1 in function f is substituted for the $\bullet$ region after leaving the binding for f. In the big-step semantics for the $\lambda_{tt\text{-}m}^{region}$-calculus, on the other hand, this term may still evaluate to 5 if the region allocated to ϱ_1 in the definition of f is re-allocated to ϱ_2 in the last line.

Below we will show that this subtle difference between the two semantics becomes irrelevant for well-typed terms. First, we proceed with the proof of Theorem 4.4.5.

Proof. (**part 1 of Theorem 4.4.5**) By induction on the derivation of $S, VE, RE \vdash e \Downarrow v, S'$.

Case $\dfrac{VE(x) = v}{S, VE, RE \vdash x \Downarrow v, S}$.

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

Case $\dfrac{RE(\varrho) = r \quad l \notin Dom\,(S(r))}{S, VE, RE \vdash c \texttt{ at } \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto \langle c \rangle\}\}}$.

Let s, e' be such that $\mathcal{Q}(S, VE, RE, c \texttt{ at } \varrho, s, e')$. Then there are two cases:

Subcase $e' \equiv c \texttt{ at } r$**.** Then $\mathcal{Q}_s(S, s)$ where $l \notin Dom\,(s(r))$. As a consequence, $s, c \texttt{ at } r \rightarrowtail s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l)$. It also holds that $\mathcal{Q}_v(S + \{r \mapsto S(r) + \{l \mapsto \langle c \rangle\}\}, (r, l), s + \{r \mapsto s(r) + \{l \mapsto \langle c \rangle\}\}, (r, l))$.

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

Case $\dfrac{RE(\varrho) = r \quad l \notin Dom\,(S(r))}{S, VE, RE \vdash \lambda\, x.\, e \texttt{ at } \varrho \Downarrow (r, l), S + \{r \mapsto S(r) + \{l \mapsto \langle x, e, VE, RE \rangle\}\}}$.

Let s, e'' be such that $\mathcal{Q}(S, VE, RE, \lambda\, x.\, e \texttt{ at } \varrho, s, e'')$. There are two cases.

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

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

Case $\dfrac{S, VE, RE \vdash e_1 \Downarrow (r_1, l_1), S_1 \quad S_1(r_1)(l_1) = \langle x, e, VE', RE' \rangle \quad S_1, VE, RE \vdash e_2 \Downarrow v_2, S_2 \quad S_2, VE' + \{x \mapsto v_2\}, RE' \vdash e \Downarrow v, S'}{S, VE, RE \vdash e_1 \mathbin{@} e_2 \Downarrow v, S'}$.

Let s, e'' be such that $\mathcal{Q}(S, VE, RE, e_1 \mathbin{@} e_2, s, e'')$. Hence, the assertion $\mathcal{Q}_s(S, s)$ holds with $\mathcal{Q}(S, VE, RE, e_1, s, e_1')$ and $\mathcal{Q}(S, VE, RE, e_2, s, e_2')$ where $e'' = e_1' \mathbin{@} e_2'$.

By induction on $S, VE, RE \vdash e_1 \Downarrow (r_1, l_1), S_1$ either s, e_1' becomes stuck (in which case $s, e_1' \mathbin{@} e_2'$ becomes also stuck) or there exist s_1', v_1' such that $s, e_1' \overset{*}{\rightarrowtail} s_1', v_1'$ and $\mathcal{Q}_v(S_1, (r_1, l_1), s_1', v_1')$.

By induction on $S_1, VE, RE \vdash e_2 \Downarrow v_2, S_2$ (using Lemma 2.3.1, Lemma 4.3.8 and Lemma 4.4.1 to establish $\mathcal{Q}$ since $\mathcal{Q}_s(S_1, s_1')$) either s_1', e_2' becomes stuck (in which

case $s_1', v_1' \mathbin{@} e_2'$ also becomes stuck) or there exist s_2', v_2' such that $s_1', e_2' \overset{*}{\rightarrowtail} s_2', v_2'$ and $\mathcal{Q}_v(S_2, v_2, s_2', v_2')$.

From the definition of $\mathcal{Q}_v(S_1, (r_1, l_1), s_1, v_1')$, there are two cases for v_1'. Either $v_1' = (\bullet, l_1)$ and then the λ_{pi}^{region}-configuration $s_2', v_1' \mathbin{@} v_2'$ is stuck; or $v_1' = (r_1, l_1)$ and since $S_1(r_1)(l_1) = \langle x, e, VE', RE' \rangle$ it must be that $s_1'(r_1)(l_1) = \lambda\, x.\, e'$ where $\mathcal{Q}(S_1,\, VE' - x, RE', e, s_1', e')$. Hence, $s_2', v_1' \mathbin{@} v_2' \rightarrowtail s_2', e'[x \mapsto v_2']$.

By Lemma 2.3.1, Lemma 4.3.8 and Lemma 4.4.1, it holds that $\mathcal{Q}(S_2, VE' - x, RE', e, s_2', e')$, since $\mathcal{Q}_s(S_2, s_2')$. Lemma 4.4.2 then states that $\mathcal{Q}(S_2, VE' + \{x \mapsto v_2\}, RE', e, s_2', e'[x \mapsto v_2'])$.

By induction, we obtain that either $s_2', e'[x \mapsto v_2']$ becomes stuck (in which case $s_2', v_1' \mathbin{@} v_2'$ becomes also stuck) or there exists some s', v' such that $s_2', e'[x \mapsto v_2'] \overset{*}{\rightarrowtail} s', v'$ and $\mathcal{Q}_v(S', v, s', v')$.

Putting the parts together using the reduction rules (4.5), (4.11) and (4.12), either $s, e_1' \mathbin{@} e_2'$ becomes stuck or $s, e_1' \mathbin{@} e_2' \overset{*}{\rightarrowtail} s', v'$ where $\mathcal{Q}_v(S', v, s', v')$.

Case $\dfrac{r \notin Dom\,(S) \quad S + \{r \mapsto \{\,\}\}, VE, RE + \{\varrho \mapsto r\} \vdash e \Downarrow v, S'}{S, VE, RE \vdash \mathtt{new}\ \varrho.\, e \Downarrow v, S' - r}$.

Let s, e'' be such that $\mathcal{Q}(S, VE, RE, \mathtt{new}\ \varrho.\, e, s, e'')$. Hence we have that $\mathcal{Q}_s(S, s)$, that $e'' = \mathtt{new}\ \varrho.\, e'$ and also that $\mathcal{Q}(S, VE, RE - \varrho, e, s, e')$. Therefore, $s, \mathtt{new}\ \varrho.\, e' \rightarrowtail s + \{r \mapsto \{\,\}\}, \mathtt{region}\ r\ \mathtt{in}\ e'[\varrho \mapsto r]$. By Lemma 4.4.3, it holds that $\mathcal{Q}(S + \{r \mapsto \{\,\}\}, VE, RE + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\,\}\}, e'[\varrho \mapsto r])$.

By induction, either $s + \{r \mapsto \{\,\}\}, e'[\varrho \mapsto r]$ becomes stuck (in which case $s + \{r \mapsto \{\,\}\}, \mathtt{region}\ r\ \mathtt{in}\ e'[\varrho \mapsto r]$ becomes stuck, too) or there exist s', v' such that $s + \{r \mapsto \{\,\}\}, e'[\varrho \mapsto r] \overset{*}{\rightarrowtail} s', v'$ and $\mathcal{Q}_v(S', v, s', v')$. Because of reduction rule (4.10), we have $s + \{r \mapsto \{\,\}\}, \mathtt{region}\ r\ \mathtt{in}\ e'[\varrho \mapsto r] \overset{*}{\rightarrowtail} s', \mathtt{region}\ r\ \mathtt{in}\ v'$. But the last configuration is a redex of rule (4.2): $s', \mathtt{region} r \mathtt{in} v' \rightarrowtail s'[r \mapsto \bullet] - r, v'[r \mapsto \bullet]$. By Lemma 4.4.4, it follows that $\mathcal{Q}_v(S' - r, v, s'[r \mapsto \bullet] - r, v'[r \mapsto \bullet])$, which concludes the proof.

□

Proof. (**part 2 of Theorem 4.4.5**) By induction on the number of steps in $s_1, e' \overset{*}{\rightarrowtail} s_2, v'$. We only consider two interesting cases:

Case $s, \mathtt{new}\ \varrho.\, e' \rightarrowtail s + \{r \mapsto \{\,\}\}, \mathtt{region}\ r\ \mathtt{in}\ e'[\varrho \mapsto r] \overset{*}{\rightarrowtail} s', v'$ *where* $r \notin Dom\,(s)$**.**

Then $s + \{r \mapsto \{\,\}\}, e'[\varrho \mapsto r] \overset{*}{\rightarrowtail} s_1, v_1'$ where $s_1[r \mapsto \bullet] - r = s'$ and $v_1'[r \mapsto \bullet] = v'$, so that $s_1, \mathtt{region}\ r\ \mathtt{in}\ v_1' \rightarrowtail s', v'$ is the last step.

Suppose now $\mathcal{Q}(S, VE, RE, \mathtt{new}\ \varrho.\, e, s, \mathtt{new}\ \varrho.\, e')$, then Lemma 4.4.3 implies $\mathcal{Q}(S + \{r \mapsto \{\,\}\}, VE, RE + \{\varrho \mapsto r\}, e, s + \{r \mapsto \{\,\}\}, e'[\varrho \mapsto r])$.

By the induction hypothesis, $S + \{r \mapsto \{\,\}\}, VE, RE + \{\varrho \mapsto r\} \vdash e \Downarrow v_1, S_1$ for which $\mathcal{Q}_v(S_1, v_1, s_1, v_1')$. Hence, by evaluation rule *(BS-Reglam)* $S, VE, RE \vdash \mathtt{new}\ \varrho.\, e \Downarrow v_1, S_1 - r$, and $\mathcal{Q}_v(S_1 - r, v_1, s_1, v')$ follows from Lemma 4.4.4.

Case $s, e_1' \mathbin{@} e_2' \overset{*}{\rightarrowtail} s', v'$**.** Then $s, e_1' \mathbin{@} e_2' \overset{*}{\rightarrowtail} s_1, v_1' \mathbin{@} e_2' \overset{*}{\rightarrowtail} s_2, v_1' \mathbin{@} v_2' \rightarrowtail s_2, e'[x \mapsto v_2'] \overset{*}{\rightarrowtail} s', v'$, where $v_1' = (r, l)$ and $s_2(r)(l) = \langle \lambda\, x.e' \rangle$ and $s, e_1' \overset{*}{\rightarrowtail} s_1, v_1'$ and $s_1, e_2' \overset{*}{\rightarrowtail} s_2, v_2'$. Note that all the intermediate $\overset{*}{\rightarrowtail}$ have strictly fewer steps than $s, e_1' \mathbin{@} e_2' \overset{*}{\rightarrowtail} s', v'$.

Suppose now that $\mathcal{Q}(S, VE, RE, e_1 \mathbin{@} e_2, s, e_1' \mathbin{@} e_2')$, then $\mathcal{Q}(S, VE, RE, e_i, s, e_i')$ for $i \in \{1, 2\}$. By the induction hypothesis, $S, VE, RE \vdash e_1 \Downarrow v_1, S_1$ for which $\mathcal{Q}_v(S_1, v_1, s_1, v_1')$.

Since $S \leq S_1$ and $s \leq s_1$ (by Lemma 2.3.1 and Lemma 4.3.8) and because $\mathcal{Q}_s(S_1, s_1)$, Lemma 4.4.1 gives $\mathcal{Q}(S_1, VE, RE, e_2, s_1, e_2')$. By the induction hypothesis, it follows that $S_1, VE, RE \vdash e_2 \Downarrow v_2, S_2$ where $\mathcal{Q}_v(S_2, v_2, s_2, v_2')$.

So, again by Lemma 4.4.1, we have that $\mathcal{Q}_v(S_2, v_1, s_2, v_1')$. Therefore we also have $S_2(r)(l) = \langle x, e, VE', RE' \rangle$ with $\mathcal{Q}(S_2, VE' - x, RE', e, s_2, e')$. Lemma 4.4.2 then implies $\mathcal{Q}(S_2, VE' + \{x \mapsto v_2\}, RE', e, s_2, e'[x \mapsto v_2'])$. By the induction hypothesis, we derive that $S_2, VE' + \{x \mapsto v_2\}, RE' \vdash e \Downarrow v, S'$ for which $\mathcal{Q}_v(S', v, s', v')$. Finally, application of reduction rule *(BS-App)* yields $S, VE, RE \vdash e_1 \mathbin{@} e_2 \Downarrow v, S'$.

□

4.4.4 Type Soundness for the $\lambda_{tt\text{-}m}^{region}$-calculus

To show type soundness of the $\lambda_{tt\text{-}m}^{region}$-calculus via the results for the λ_{pi}^{region}-calculus, we need to strengthen Theorem 4.4.5 to typed terms. To do so, we need to formalize how we *translate* typed $\lambda_{tt\text{-}m}^{region}$-Terms to λ_{pi}^{region}-Terms.

First, define the region substitution $S_r^{RE} \in \mathsf{RegionVar} \rightarrow \mathsf{Placeholder}$ derived from the region environment RE as follows: $S_r^{RE}(\varrho) = RE(\varrho)$, whenever $\varrho \in Dom\,(RE)$ and $S_r^{RE}(\varrho) = \varrho$ otherwise.

Definition 4.4.2 *We say that* (S, RE, e, μ, φ) *is a* typed start configuration *for a closed* $\lambda_{tt\text{-}m}^{region}$-Term e *iff* $S = \{r \mapsto \{\,\} \mid r \in Ran\,(RE)\}$ *and* $\{\,\} \vdash_t^t e : \mu, \varphi$ *with* $frv\,(\mu) \cup frv\,(\varphi) \subseteq Dom\,(RE)$.

The tuple $(S, RE, e, \mu, \varphi, s, e')$ *is a* typed translation *iff* (S, RE, e, μ, φ) *is a typed start configuration,* $e' = S_r^{RE}(e)$ *and* $s = \{r \mapsto \{\,\} \mid r \in Ran\,(RE)\}$.

Note that the condition $frv\,(\mu) \cup frv\,(\varphi) \subseteq Dom\,(RE)$ implies that all regions, which are not solely referred to in dead code, have to be allocated.

We have the following straightforward property for typed translations:

Lemma 4.4.6 *If* $(S, RE, e, \mu, \varphi, s, e')$ *is a typed translation, then* $\mathcal{Q}(S, \{\,\}, RE, e, s, e')$ *holds.*

The typed variant of Theorem 4.4.5 can now be formulated:

Theorem 4.4.7 *Let* $(S, RE, e, \mu, \varphi, s, e')$ *be a typed translation. Then*

$$\textit{(BS-Const-Err)} \quad \frac{\varrho \notin \mathit{Dom}\,(RE)}{S,\, VE, RE \vdash c\ \mathtt{at}\ \varrho \Downarrow \mathsf{err}}$$

$$\textit{(BS-Var-Err)} \quad \frac{x \notin \mathit{Dom}\,(VE)}{S,\, VE, RE \vdash x \Downarrow \mathsf{err}}$$

$$\textit{(BS-Lam-Err)} \quad \frac{\varrho \notin \mathit{Dom}\,(RE)}{S,\, VE, RE \vdash \lambda x.\, e\ \mathtt{at}\ \varrho \Downarrow \mathsf{err}}$$

$$\textit{(BS-App-Err)} \quad \frac{S,\, VE, RE \vdash e_1 \Downarrow (r_1, l_1), S_1\ \wedge\ S_1,\, VE, RE \vdash e_2 \Downarrow v_2, S_2\ \wedge\ (r_1 \notin \mathit{Dom}\,(S_2) \vee S_2(r_1)(l_1) \neq \langle x, e, VE', RE' \rangle)}{S,\, VE, RE \vdash e_1\ @\ e_2 \Downarrow \mathsf{err}}$$

$$\textit{(BS-Copy-Err)} \quad \frac{\{\varrho, \varrho'\} \not\subseteq \mathit{Dom}\,(RE) \vee (S,\, VE, RE \vdash e \Downarrow (r, l), S' \wedge (r \notin \mathit{Dom}\,(S') \vee S'(r)(l) \neq \langle c \rangle))}{S,\, VE, RE \vdash \mathtt{copy}\,[\varrho, \varrho']\, e \Downarrow \mathsf{err}}$$

Figure 4.6: Extending the $\lambda^{region}_{tt\text{-}m}$-calculus semantics with errors

1. *If* $S, \{\,\}, RE \vdash e \Downarrow v, S'$ *then there exist a* v' *and an* s' *such that* $s, e' \overset{*}{\rightarrowtail} s', v'$ *where* $\mathcal{Q}_v(S', v, s', v')$.

2. *If* $s, e' \overset{*}{\rightarrowtail} s', v'$ *then there exist* v *and* S' *such that* $S, \{\,\}, RE \vdash e \Downarrow v, S'$ *where* $\mathcal{Q}_v(S', v, s', v')$.

Proof. Part 2 is a special case of Theorem 4.4.5, part 2. For part 1, take a heap type $H = \{r \mapsto \{\,\} | r \in \mathit{Ran}\,(RE)\}$. Hence $H \vdash^h_i s$ and therefore by rule *(Conf)* also $H \vdash_i s, e : \mu, \varphi$. We can now apply Lemma 4.3.5 to obtain $S^{RE}_r(H) \vdash_i S^{RE}_r(s), S^{RE}_r(e) : S^{RE}_r(\mu), S^{RE}_r(\varphi)$ or simplified $H \vdash_i s, e' : S^{RE}_r(\mu), S^{RE}_r(\varphi)$ since $e' = S^{RE}_r(e)$. Because it also holds that $\mathit{frv}\,(\mu) \cup \mathit{frv}\,(\varphi) \subseteq \mathit{Dom}\,(\mathit{Renv})$, we have that $\mathit{Clean}\,(\mathit{frv}\,(S^{RE}_r(\varphi)))$ and $\mathit{frv}\,(S^{RE}_r(\mu)) \subseteq \mathit{Ran}\,(RE) = \mathit{Dom}\,(H)$. These are the condition for Theorem 4.3.11, which we use to conclude that the configuration s, e' cannot become stuck. Because of Lemma 4.4.6 we have $\mathcal{Q}(S, \{\,\}, RE, e, s, e')$ and so by Theorem 4.4.5, part 1, we know that there exist a v' and an s' such that $s, e' \overset{*}{\rightarrowtail} s', v'$ and $\mathcal{Q}_v(S', v, s', v')$. □

To give a complete soundness proof for the $\lambda^{region}_{tt\text{-}m}$-semantics, we need an equivalent notion of a stuck term in the big-step semantics of the $\lambda^{region}_{tt\text{-}m}$-calculus. We do this by extending the $\lambda^{region}_{tt\text{-}m}$-evaluation relation (see Figure 2.5 on page 24) with error reductions. We extend the λ^{region}_{tt}-Value domain with an error value err. The extra evaluation rules are presented in Figure 4.6. For brevity, we omit the canonical rules for error propagation.

The following lemma relates error reductions in the $\lambda^{region}_{tt\text{-}m}$-calculus with stuck terms in the λ^{region}_{pi}-calculus.

Lemma 4.4.8 *If* $\mathcal{Q}(S, VE, RE, e, s, e')$ *and* $S, VE, RE \vdash e \Downarrow \mathsf{err}$ *then* s, e' *becomes stuck.*

Proof. By induction on the evaluation of $S, VE, RE \vdash e \Downarrow \mathsf{err}$. All error propagation cases are straightforward applications of the induction hypothesis. The base cases are easily shown using Theorem 4.4.5, we illustrate this with the case for application:

Case $S, VE, RE \vdash e_1 @ e_2 \Downarrow \mathsf{err}$.

So we have that $S, VE, RE \vdash e_1 \Downarrow (r_1, l_1), S_1$. Since $\mathcal{Q}(S, VE, RE, e_1, s, e_1')$ holds and using Theorem 4.4.5 we have either that s, e_1' is stuck, from which we also have that $s, e_1' @ e_2'$ is stuck following reduction rule (4.11).

Alternatively, there exist s_1, v_1' such that $s, e_1' \overset{*}{\rightarrowtail} s_1, v_1'$ where $\mathcal{Q}_v(S_1, (r_1, l_1), s_1, v_1')$. We also have $S_1, VE, RE \vdash e_2 \Downarrow v_2, S_2$ and $\mathcal{Q}(S, VE, RE, e_2, s, e_2')$. Hence, by Lemma 2.3.1, Lemma 4.3.8 and Lemma 4.4.1, we have $\mathcal{Q}(S_1, VE, RE, e_2, s_1, e_2')$. Again, we can apply Theorem 4.4.5 and have either that:

- s_1, e_2' becomes stuck and therefore, by reduction rules (4.11) and (4.12), that $s, e_1' @ e_2'$ also becomes stuck; or
- there exists a configuration s_2, v_2' for which it holds that $s_1, e_2 \overset{*}{\rightarrowtail} s_2, v_2'$ with $\mathcal{Q}_v(S_2, v_2, s_2, v_2')$. Again by Lemma 2.3.1, Lemma 4.3.8 and Lemma 4.4.1 we have that $\mathcal{Q}_v(S_2, (r_1, l_1), s_2, v_1')$.

  Following reduction *(BS-App-Err)*, we have two cases:

  - Case $r_1 \notin Dom(S_2)$. Then $v_1' = (\bullet, l_1)$ and therefore, by the reduction rules (4.5), (4.11) and (4.12), the configuration $s, e_1' @ e_2'$ becomes stuck.
  - Case $S_2(r_1)(l_1) \neq \langle x, e, VE', RE' \rangle$. Then, if $v_1' = (\bullet, l_1)$, see previous case. Otherwise, $S_2(r_1(l_1)) = \langle c \rangle = s_2(r_1(l_1))$ and again by the reduction rules (4.5), (4.11) and (4.12), $s, e_1' @ e_2'$ becomes stuck.

□

Theorem 4.4.9 (Type Soundness of the $\lambda^{region}_{tt\text{-}m}$-calculus) *Suppose a typed start configuration (S, RE, e, μ, φ). Then $S, \{\,\}, RE \vdash e \not\Downarrow \mathsf{err}$.*

Proof. Take s and e' such that $(S, RE, e, \mu, \varphi, s, e')$ is a typed translation. By analogous reasoning as in the proof for Theorem 4.4.7, we know that s, e' does not become stuck. But since $\mathcal{Q}(S, \{\,\}, RE, e, s, e')$ holds by Lemma 4.4.6, we can conclude that $S, \{\,\}, RE \vdash e \not\Downarrow \mathsf{err}$ as a consequence of Lemma 4.4.8. □

4.5 Relating the two small-step Region Calculi

Relating the λ^{region}_{pi}-calculus with the $\lambda^{region}_{tt\text{-}m}$-calculus allows for a type soundness proof of the Tofte-Talpin model by considering the syntactic proof for the imperative region calculus. However, in the rest of the thesis, we will work with variations of the λ^{region}_{s}-calculus since we do not develop a theory for destructive store operators. In addition, the small-step semantics for the λ^{region}_{s}-calculus is extremely concise, which is an appealing

$$
\begin{array}{lcl}
\mathcal{H}(s,(r,l),e',RE_o) & \Leftrightarrow & (e' \equiv \langle c\rangle_{\varrho} \wedge s(r)(l) = \langle c\rangle \wedge RE_o(\varrho) = r) \vee \\
 & & (e' \equiv \langle \lambda\, x.\, e''\rangle_{\varrho} \wedge s(r)(l) = \langle \lambda\, x.\, e\rangle \wedge \\
 & & \mathcal{H}(s,e,e'',RE_o) \wedge RE_o(\varrho) = r) \\
\mathcal{H}(s,(\bullet,l),e',RE_o) & \Leftrightarrow & (e' \equiv \langle c\rangle_{\bullet}) \vee (e' \equiv \langle \lambda\, x.\, e''\rangle_{\bullet}) \\
\mathcal{H}(s,c\ \mathtt{at}\ r,e',RE_o) & \Leftrightarrow & e' \equiv c\ \mathtt{at}\ \varrho \wedge RE_o(\varrho) = r \\
\mathcal{H}(s,c\ \mathtt{at}\ \bullet,e',RE_o) & \Leftrightarrow & e' \equiv c\ \mathtt{at}\ \bullet \\
\mathcal{H}(s,\lambda\, x.\, e\ \mathtt{at}\ r,e',RE_o) & \Leftrightarrow & e' \equiv \lambda\, x.\, e''\ \mathtt{at}\ \varrho \wedge \mathcal{H}(s,e,e'',RE_o) \wedge RE_o(\varrho) = r \\
\mathcal{H}(s,\lambda\, x.\, e\ \mathtt{at}\ \bullet,e',RE_o) & \Leftrightarrow & e' \equiv \lambda\, x.\, e''\ \mathtt{at}\ \bullet \\
\mathcal{H}(s,x,e',RE_o) & \Leftrightarrow & e' \equiv x \\
\mathcal{H}(s,e_1 \mathbin{@} e_2,e',RE_o) & \Leftrightarrow & e' \equiv e'_1 \mathbin{@} e'_2 \wedge \mathcal{H}(s,e_1,e'_1,RE_o) \wedge \mathcal{H}(s,e_2,e'_2,RE_o) \\
\mathcal{H}(s,\mathtt{new}\ \varrho.\, e,e',RE_o) & \Leftrightarrow & e' \equiv \mathtt{new}\ \varrho.\, e'' \wedge r \notin \mathit{Dom}\,(s) \wedge \\
 & & \mathcal{H}(s + \{r \mapsto \{\,\}\}, e[\varrho \mapsto r], e'', RE_o + \{\varrho \mapsto r\}) \\
\mathcal{H}(s,\mathtt{region}\ r\ \mathtt{in}\ e,e',RE_o) & \Leftrightarrow & e' \equiv \mathtt{new}\ \varrho.\, e'' \wedge \mathcal{H}(s,e,e'',RE_o + \{\varrho \mapsto r\})
\end{array}
$$

Figure 4.7: A relation between λ^{region}_{pi}-Terms and $\lambda^{region}_{s\text{-}m}$-Terms

property for using it as a basis to develop an equational theory (see Chapter 5) and a specialization model (see Chapter 6).

To be able to fall back on the λ^{region}_{tt}-calculus, for example to reuse efficient region reconstruction algorithms [11, 146], we relate the λ^{region}_{pi}-calculus to the $\lambda^{region}_{s\text{-}m}$-calculus. The latter is a monomorphic subset of the λ^{region}_{s}-calculus, excluding the `copy`-operator and `let`-bindings.

Fortunately, proving an equivalence result between the λ^{region}_{pi}-calculus and the $\lambda^{region}_{s\text{-}m}$-calculus is considerably simpler because we are dealing with two small-step semantics. The proof boils down to a simple induction on the two transition relations.

The most important difference between these two calculi is the handling of region constants. Region variables play a dual role in the $\lambda^{region}_{s\text{-}m}$-calculus. On the one hand, they are α-convertible to avoid conflicting uses, but on the other hand, they substitute for region names and thus turn up in addresses (see Section 3.2.2 for more explanation). In contrast, the region names in the λ^{region}_{pi}-calculus are bound by the store which guarantees that a region name is not in use at the point where a new region is created.

We cater for this apparent mismatch with an explicit region environment that maps region variables (on the $\lambda^{region}_{s\text{-}m}$-calculus side) to region names (on the λ^{region}_{pi}-calculus side): The region environment $RE_o \in \mathsf{RegionVar} \hookrightarrow \mathsf{RegionName}$ relates allocated region variables with actual regions.

We relate λ^{region}_{pi}-configurations and $\lambda^{region}_{s\text{-}m}$-Terms via the relation

$$\mathcal{H} \in \mathcal{P}(\lambda^{region}_{pi}\text{-}\mathsf{Store} \times \lambda^{region}_{pi}\text{-}\mathsf{Term} \times \lambda^{region}_{s\text{-}m}\text{-}\mathsf{Term} \times (\mathsf{RegionVar} \hookrightarrow \mathsf{RegionName}))$$

which is defined by the equivalences in Figure 4.7. In all the uses of $\mathcal{H}(s,e,e',RE_o)$, we implicitly assume that RE_o is injective and $\mathit{frv}\,(e') \subseteq \mathit{Dom}\,(RE_o)$.

In contrast to the relation $\mathcal{Q}$ of Section 4.4.1, we can define $\mathcal{H}$ as the least fixpoint of the associated functional. In fact, it is easy to see that $\mathcal{H}$ is well-founded and that cycles are a non-issue since it is not possible to create them in the $\lambda^{region}_{s\text{-}m}$-calculus.

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

Proposition 4.5.1 *Suppose e is a λ_{pi}^{region}-*Term *and the function RE_o an injective region environment for which $frv(e) \subseteq$ and $s = \{r \mapsto \{\,\} | r \in Ran(RE_o)\}$.*
Then $\mathcal{H}(RE_o(s), RE_o(e), e, RE_o)$.

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

Lemma 4.5.2 *Suppose $\mathcal{H}(s, e, e', RE_o + \{\varrho \mapsto r\})$.*
Then $\mathcal{H}(s[r \mapsto \bullet] - r, e[r \mapsto \bullet], e'[\varrho \mapsto \bullet], RE_o)$.

Lemma 4.5.3 *Suppose $\mathcal{H}(s, e, e', RE_o)$, $\varrho \notin Dom(RE_o)$ and $r \notin Ran(RE_o)$.*
Then $\mathcal{H}(s, e, e', RE_o + \{\varrho \mapsto r\})$.

Lemma 4.5.4 *Suppose $\mathcal{H}(s, e_1, e_1', RE_o)$ and $\mathcal{H}(s, e_2, e_2', RE_o)$.*
Then $\mathcal{H}(s, e_1[x \mapsto e_2], e_1'[x \mapsto e_2'], RE_o)$.

Let us write $\rightarrowtail^{1,2}$ whenever we make one or two reduction steps in the relation $\rightarrowtail$. The equivalence between the $\lambda_{s\text{-}m}^{region}$-calculus and the λ_{pi}^{region}-calculus is a result of the following theorem:

Theorem 4.5.5 *Suppose $\mathcal{H}(s_1, e_1, e_1', RE_o)$.*

1. *If $s_1, e_1 \rightarrowtail s_2, e_2$ then there exist e_2' and RE_o' such that $\mathcal{H}(s_2, e_2, e_2', RE_o')$ and either $e_1' \rightarrow e_2'$ or $e_1' = e_2'$.*
2. *If $e_1' \rightarrow e_2'$ then there exist a configuration s_2, e_2 and region environment RE_o' such that $s_1, e_1 \rightarrowtail^{1,2} s_2, e_2$ and $\mathcal{H}(s_2, e_2, e_2', RE_o')$.*

Proof. The proof is a case analysis on the definition of $\rightarrowtail$ and $\rightarrow$, making use of Lemma 4.5.2, Lemma 4.5.3 and Lemma 4.5.4. □

4.6 Related Work

The original paper by Tofte and Talpin [150] presents a big-step store-based operational semantics (see also Section 2.3.2), however it does not treat operations on references. Instead, the authors informally discuss extensions of the λ_{tt}^{region}-calculus to cater for full Standard ML. Destructive store operations are introduced as primitives in the environment. Although no dynamic semantics is presented, the informal operational description (Section 11.1 in [150]) seems to coincide with our operational semantics for destructive store operations.

The λ_{i}^{region}-calculus is a closer match to an actual implementation than the λ_{s}^{region}-calculus from Chapter 3 and has some similarities with the Tofte-Talpin λ_{tt}^{region}-calculus. However, the different formal approaches make it surprisingly technical to relate them as witnessed by the elaboration in Section 4.4. The definition of the relation is inspired by

the Calcagno's work [17], who specifies a "high-level" (big-step) operational semantics for the Tofte-Talpin region calculus. In contrast to our treatment, his formal relation is defined as the least fixpoint of a well-founded functional.

Several type soundness proofs have been developed for region calculi, usually indirect via another formalism and only for a monomorphic side-effect free subset. They are listed in Section 3.4 of Chapter 3.

4.7 Chapter Summary

We extended the concise transition-based storeless λ_s^{region}-calculus of Chapter 3 with an explicit store and side-effecting operations. This extension, called the λ_i^{region}-calculus, is given a type system and small-step semantics in Section 4.2. Very similar to the syntactic type soundness proof for the λ_s^{region}-calculus, Section 4.3 elaborates a soundness proof for the imperative λ_i^{region}-calculus. This is achieved in the usual way by formulating a type preservation and progress property.

Section 4.4 formally relates a monomorphic and pure instance of the λ_i^{region}-calculus to a monomorphic subset of the λ_{tt}^{region}-calculus for Section 2.3. Assuming well-typedness, we were able to formalize a type soundness result for the λ_{tt}^{region}-calculus indirectly via its relation to the λ_i^{region}-calculus. Finally, Section 4.5 also related the pure imperative λ_{pi}^{region}-calculus to a monomorphic subset of the λ_s^{region}-calculus.

5 AN EQUATIONAL THEORY FOR A REGION CALCULUS

5.1 Introduction

An equational theory is an axiomatic proof system entirely based on syntactic equations and derivation rules. It provides an elegant method to prove two expressions *semantically* equivalent and is therefore very well suited for proving the correctness of program transformations.

In this chapter, we develop a typed equational theory to reason about region annotated programs. We formalize the theory for a variant of the λ_s^{region}-calculus, which generalizes `letrec`-bound polymorphic functions to first-class recursive function objects. We refrain from investigating an untyped equational theory since our region calculus is inherently type-based: region annotation is guided by a type and effect analysis.

Using bisimulation, a notion originating from work on CCS [101], we prove that the equational theory is sound with respect to Morris-style contextual equivalence [108, 121]. Following methods by Gordon, Abramsky and others [3, 29, 48–50, 53], we specify a notion of applicative bisimilarity for the region calculus. With a proof technique due to Howe [76, 77], re-casted to typed functional programming languages by Gordon [48], we show that bisimilarity is a congruence for our region calculus. We use this result to prove that bisimilarity equals contextual equivalence and includes the equational theory.

We closely follow Gordon's results on contextual equivalence for a polymorphically typed object calculus [49]. However, our syntactic theory is based on a transition semantics. We found it therefore more natural to use a labeled transition system [29, 50] to define bisimulation. The use of a transition system over an alternative formulation is probably a matter of taste.

The results of this chapter are submitted for publication [65].

5.2 The λ_f^{region}-calculus

The λ_f^{region}-calculus is a slight variation of the λ_s^{region}-calculus (see Section 3.2), where `letrec`-bound recursive functions are treated as first-class objects. This generalization allows more elegant and manageable equational rules as will become clear later in this chapter. However, it is possible to recast the theory to the λ_s^{region}-calculus. We will discuss a relation between the λ_f^{region}-calculus and the λ_s^{region}-calculus in Section 5.2.4.

$$\rho \in \mathsf{Placeholder} = \mathsf{RegionVar} \cup \{\bullet\}$$

$$\mathsf{Term} \ni e ::= x \mid e \mathbin{@} e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{new}\ \varrho.\, e \mid o\ \mathtt{at}\ \rho \mid v \mid e\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho$$

$$\mathsf{Object} \ni o ::= \lambda\, x.\, e \mid \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \qquad \mathsf{Value} \ni v ::= \langle o \rangle_\rho$$

Figure 5.1: Syntax of the λ_f^{region}-calculus

5.2.1 Syntax of the λ_f^{region}-calculus

The syntax is reminiscent of the λ_s^{region}-calculus and presented in Figure 5.1. Variables, function application and region abstractions are standard. The `let`-construct is a Hindley-Milner polymorphic generalization of the same construct in the λ_s^{region}-calculus. Object allocations, $o\ \mathtt{at}\ \rho$, and object pointers $\langle o \rangle_\rho$ work as usual, however, we do not consider constants and add recursive functions of the form $\mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.e$. These are lambda abstractions, that additionally abstract over region variables and bind themselves recursively to f within e. We require $n \geq 1$ to distinguish lambda objects from recursive function objects. This restriction is technical and it is an easy exercise to extend the theory of this chapter to lambda objects, which are recursive, but not region polymorphic.

Together with the type-polymorphic `let`-construct, recursive functions subsume the `letrec`-bound functions of the λ_s^{region}-calculus. Due to first-class region polymorphic function objects, we have a more general region application construct $e\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho$. It evaluates the expression e to a recursive function object, applies the regions $\rho_1, \ldots, \rho_n$ and puts the resulting lambda object in region ρ.

The definition of a region placeholder is identical to that of the λ_s^{region}-calculus. This means that we have a distinct *dead* region variable $\bullet$, which serves as a sink region for deallocated objects.

5.2.2 The Dynamic Semantics

The dynamic semantics of the λ_f^{region}-calculus is specified in the usual small-step reduction style of Chapter 3 and Chapter 4. As with the λ_s^{region}-calculus, it requires no store because it does not model destructive updates. Region allocation is implicit and only deallocation is visible in the reduction rules.

In contrast to the presentation in the previous chapters, we separate the beta reduction rules from context reduction, so we can reuse the beta rules in the definition of the equational theory. The reduction relation $\overset{n}{\dashrightarrow} \in \mathcal{P}(\mathsf{Term} \times \mathbb{N} \times \mathsf{Term})$ specifies computational reduction for the λ_f^{region}-calculus and is defined in Figure 5.2.

Rules (5.1) and (5.2) allocate a lambda and a recursive function value respectively. Rule (5.3) deallocates a region after the enclosed expression is reduced to a value. As before, region deallocation is modeled by renaming the region variable ϱ to the dead

$$\lambda\, x.\, e \;\texttt{at}\; \varrho \xrightarrow{1} \langle \lambda\, x.\, e \rangle_\varrho \tag{5.1}$$

$$\texttt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \;\texttt{at}\; \varrho \xrightarrow{2} \langle \texttt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho \tag{5.2}$$

$$\texttt{new}\, \varrho.\, v \xrightarrow{3} v[\varrho \mapsto \bullet] \tag{5.3}$$

$$\langle \lambda\, x.\, e \rangle_\varrho \,@\, v \xrightarrow{4} e[x \mapsto v] \tag{5.4}$$

$$\texttt{let}\; x = v \;\texttt{in}\; e \xrightarrow{5} e[x \mapsto v] \tag{5.5}$$

$$\begin{array}{c} \langle \texttt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho\, [\rho'_1, \ldots, \rho'_n] \;\texttt{at}\; \varrho' \xrightarrow{6} \\ \langle \lambda\, x.\, e[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n][f \mapsto \langle \texttt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho] \rangle_{\varrho'} \end{array} \tag{5.6}$$

Figure 5.2: Dynamic semantics of the λ_f^{region}-calculus - part I

region $\bullet$. All objects previously in ϱ, are inaccessible now and the variable ϱ can be reused. Beta-value reduction is specified by the rules (5.4) and (5.5) for lambdas and the `let`-construct respectively. Rule (5.6) implements region application, where the recursive function is bound to f *simultaneously* with the region substitutions.

A small-step semantics for the λ_f^{region}-calculus is now defined by the relation $\rightharpoonup$ and its rules are listed in Figure 5.3. Rule (5.7) lifts the beta rules from Figure 5.2 into the small-step semantics, where $n \in \{1, \ldots, 6\}$. The rules (5.8) through (5.12) are standard for a left-to-right call-by-value language. As usual, we define the relation $\overset{*}{\rightharpoonup}$ as the reflexive and transitive closure of the one-step reduction relation $\rightharpoonup$.

We say that an expression $e \in \lambda_f^{region}\text{-}\mathsf{Term}$ *converges* ($e \downarrow$) iff there exists a value $v \in \lambda_f^{region}\text{-}\mathsf{Value}$ such that $e \overset{*}{\rightharpoonup} v$. If for each $e \overset{*}{\rightharpoonup} e'$, there exists an e'' such that $e' \rightharpoonup e''$, then we say that e *diverges* ($e \uparrow$).

5.2.3 The Static Semantics

The static semantics for the λ_f^{region}-calculus is a simple adaptation of the type system of the λ_s^{region}-calculus from Figure 2.6 on page 27 and Figure 3.3 on page 33.

First, we have to adapt the static semantic objects to accommodate for the first-class recursive function objects of the λ_f^{region}-calculus. The type language is now given by the following grammar:

$$\mathsf{Type} \ni \tau ::= \alpha \mid \pi_1 \xrightarrow{\epsilon.\varphi} \pi_2 \qquad \mathsf{ResultType} \ni \overline{\sigma} ::= \forall\, \vec{\varrho}\,.\tau$$

$$\mathsf{TypeWPlace} \ni \pi ::= (\overline{\sigma}, \rho) \qquad \mathsf{TypeScheme} \ni \sigma ::= \forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\overline{\sigma}$$

A type τ is either a type variable $\alpha \in \mathsf{TypeVar}$ or an arrow type $\pi_1 \xrightarrow{\epsilon.\varphi} \pi_2$. As with the polymorphic $\vdash_t^t$ typing judgment, the arrow effect $\epsilon.\varphi$ describes the effect of the body and is discharged on function application. The effect variable ϵ stands for the arrow effect of the function.

$$\frac{e \xrightarrow{n} e'}{e \rightharpoonup e'} \qquad n \in \{1, \ldots, 6\} \tag{5.7}$$

$$\frac{e \rightharpoonup e'}{e \mathbin{@} e'' \rightharpoonup e' \mathbin{@} e''} \tag{5.8}$$

$$\frac{e \rightharpoonup e'}{\langle \lambda\, x.\, e'' \rangle_\rho \mathbin{@} e \rightharpoonup \langle \lambda\, x.\, e'' \rangle_\rho \mathbin{@} e'} \tag{5.9}$$

$$\frac{e \rightharpoonup e'}{\mathtt{let}\ x = e\ \mathtt{in}\ e'' \rightharpoonup \mathtt{let}\ x = e'\ \mathtt{in}\ e''} \tag{5.10}$$

$$\frac{e \rightharpoonup e'}{e\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \rightharpoonup e'\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho} \tag{5.11}$$

$$\frac{e \rightharpoonup e'}{\mathtt{new}\ \varrho.\, e \rightharpoonup \mathtt{new}\ \varrho.\, e'} \tag{5.12}$$

Figure 5.3: Dynamic semantics of the λ_f^{region}-calculus - part II

In contrast with the $\vdash_t^t$ typing judgment, we define a *result type* $\overline{\sigma}$, which quantifies a type over region variables. Although this introduces rank-n region polymorphism (not type or effect polymorphism), it eases the presentation below. A type with place π is now a pair of a result type $\overline{\sigma}$ and a region ρ. A type scheme is as usual and quantifies a result type over type variables α and effect variables ϵ.

A substitution $S_s = (S_t, S_e, S_r)$ is a triple of a type, effect and region substitution, and almost identical to the definition of substitution in Section 2.3.3. We refine its definition for λ_f^{region}-Types:

$$S_s(\pi \xrightarrow{\epsilon.\varphi} \pi') = S_s(\pi) \xrightarrow{\epsilon'.\varphi'} S_s(\pi') \quad \textit{where } S_e(\epsilon) = \epsilon'.\varphi'' \textit{ and } \varphi' = \varphi'' \cup S_s(\varphi)$$

$$S_s(\alpha) = S_t(\alpha) \quad S_s(\varrho) = S_r(\varrho) \quad S_s(\bullet) = \bullet \quad S_s(\overline{\sigma}, \rho) = (S_s(\overline{\sigma}), S_s(\rho))$$

$$S_s(\forall\, \vec{\varrho}.\tau) = \forall\, \vec{\varrho}.(S_s - \{\vec{\varrho}\})(\tau) \quad S_s(\forall\, \vec{\alpha}.\forall\, \vec{\epsilon}.\overline{\sigma}) = \forall\, \vec{\alpha}.\forall\, \vec{\epsilon}.(S_s - (\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}))(\overline{\sigma})$$

A type τ is an *instance* of a result type $\overline{\sigma} = \forall\, \vec{\varrho}.\tau'$ via region substitution S_r, written $\tau \prec \overline{\sigma}$ *via* S_r, if $\mathit{Supp}\,(S_r) \subseteq \{\vec{\varrho}\}$ and $S_r(\tau') = \tau$. Similarly, we have that a result type $\overline{\sigma}$ is an instance of type scheme $\sigma = \forall\, \vec{\alpha}.\forall\, \vec{\epsilon}.\overline{\sigma}'$ via substitution $S_s = (S_t, S_e, I_r)$, written $\overline{\sigma} \prec \sigma$ *via* S_s, if $\mathit{Supp}\,(S_t) \subseteq \{\vec{\alpha}\}$, $\mathit{Supp}\,(S_e) \subseteq \{\vec{\epsilon}\}$, and $S_s(\overline{\sigma}') = \overline{\sigma}$. Note that the definition of $\mathit{Clean}\,(\cdot)$ is identical to the one of Section 3.2.3, *i.e.* $\mathit{Clean}\,(A)$ is true iff $\mathit{frv}\,(A) \subseteq \mathsf{RegionVar}$ and false otherwise.

A typing judgment $TE \vdash_f e : (\overline{\sigma}, \rho), \varphi$ assigns type with place $(\overline{\sigma}, \rho)$ and effect φ to the λ_f^{region}-Term e given a type environment TE. The individual type rules are fairly similar to those of the λ_s^{region}-calculus and listed in Figure 5.4.

We only discuss some peculiarities specific to the λ_f^{region}-calculus. Because recursive functions are first-class objects, we have separated region polymorphism from type

(F-Var)
$$\frac{TE(x) = (\sigma, \rho) \quad \overline{\sigma} \prec \sigma \ \mathit{via}\ (S_t, S_e, I_r)}{TE \vdash_f x : (\overline{\sigma}, \rho), \emptyset}$$

(F-Lam)
$$\frac{TE + \{x \mapsto \pi_1\} \vdash_f e : \pi_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_f \lambda\, x.\, e\ \mathtt{at}\ \rho : (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2, \rho), \{\rho\}}$$

(F-Lamv)
$$\frac{TE + \{x \mapsto \pi_1\} \vdash_f e : \pi_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_f \langle \lambda\, x.\, e \rangle_\rho : (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2, \rho), \emptyset}$$

(F-Reglam)
$$\frac{TE \vdash_f e : \pi, \varphi \quad \varrho \notin frv\,(TE, \pi)}{TE \vdash_f \mathtt{new}\ \varrho.\, e : \pi, \varphi \setminus \{\varrho\}}$$

(F-App)
$$\frac{TE \vdash_f e_1 : (\pi_1 \xrightarrow{\epsilon.\varphi} \pi_2, \rho), \varphi_1 \quad TE \vdash_f e_2 : \pi_1, \varphi_2}{TE \vdash_f e_1 \mathbin{@} e_2 : \pi_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

(F-Let)
$$\frac{\begin{array}{c} TE \vdash_f e_1 : (\overline{\sigma}', \rho'), \varphi' \quad (\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap fv\,(TE) = \emptyset \\ TE + \{x \mapsto (\forall\, \vec{\alpha}\, . \forall\, \vec{\epsilon}\, . \overline{\sigma}', \rho')\} \vdash_f e_2 : \pi, \varphi \end{array}}{TE \vdash_f \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \pi, \varphi \cup \varphi'}$$

(F-Fix)
$$\frac{\begin{array}{c} (\{\varrho_1, \ldots, \varrho_n\} \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \\ TE + \{f \mapsto (\forall\, \vec{\epsilon}. \forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \rho)\} \vdash_f \lambda\, x.\, e\ \mathtt{at}\ \rho : (\tau, \rho), \{\rho\} \end{array}}{TE \vdash_f \mathtt{fix}\, f. \Lambda\, \varrho_1, \ldots, \varrho_n. \lambda\, x.\, e\ \mathtt{at}\ \rho : (\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \rho), \{\rho\}}$$

(F-Fixv)
$$\frac{\begin{array}{c} (\{\varrho_1, \ldots, \varrho_n\} \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \\ TE + \{f \mapsto (\forall\, \vec{\epsilon}. \forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \rho)\} \vdash_f \langle \lambda\, x.\, e \rangle_\rho : (\tau, \rho), \emptyset \end{array}}{TE \vdash_f \langle \mathtt{fix}\, f. \Lambda\, \varrho_1, \ldots, \varrho_n. \lambda\, x.\, e \rangle_\rho : (\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \rho), \emptyset}$$

(F-Regapp)
$$\frac{\begin{array}{c} TE \vdash_f e : (\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \rho), \varphi \\ S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\} \end{array}}{TE \vdash_f e\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (S_r(\tau), \rho'), \varphi \cup \{\rho, \rho'\}}$$

(F-Effect)
$$\frac{TE \vdash_f e : \pi, \varphi \quad \epsilon \notin fev\,(TE, \pi)}{TE \vdash_f e : \pi, \varphi \setminus \{\epsilon\}}$$

Figure 5.4: Static semantics for the λ_f^{region}-calculus

polymorphism. Rules *(F-Fix)* and *(F-Fixv)* type a recursive function and its value counterpart respectively. The result type of the recursive function generalizes over region variables. The recursive binding additionally generalizes over effect variables, but not over type variables. Type polymorphism is introduced by type rule *(F-Let)*, which specifies standard Hindley-Milner `let`-polymorphism. The *(F-Let)* rule also generalizes over effect variables.

Type rule *(F-Var)* now only instantiates a type scheme to a result type. Further instantiation of a result type into a "normal" type is done upon region application, as formulated in type rule *(F-Regapp)*.

As for the λ_s^{region}-calculus, we have polymorphic recursion for region and effect variables, but not for type variables. The use of result types in type judgments introduces rank-n region polymorphism, which may make region reconstruction more difficult or even undecidable. However, this is irrelevant for the development of the equational theory.

5.2.4 Properties of the λ_f^{region}-calculus

As with the λ_i^{region}- and λ_s^{region}-calculus, the λ_f^{region}-calculus exhibits many properties, which are usually proven by simple inductions similar in spirit to the proofs found in Chapter 3 and Chapter 4. We have a type substitution lemma as usual:

Lemma 5.2.1 *If* $TE \vdash_f e : \pi, \varphi$ *then, for any type substitution* S_s*, we have that* $S_s(TE) \vdash_f S_s(e) : S_s(\pi), S_s(\varphi)$.

We have a variable substitution lemma, subsuming Lemma 3.3.5 and Lemma 3.3.6 from Section 3.3.1:

Lemma 5.2.2 (Substitution Lemma) *Suppose* $TE \vdash_f v : (\overline{\sigma}, \rho), \emptyset$ *and take* $\vec{\alpha}$ *and* $\vec{\epsilon}$ *such that* $(\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap fv\,(TE) = \emptyset$*. Additionally, assume* $TE + \{x \mapsto (\forall\,\vec{\alpha}\,.\forall\,\vec{\epsilon}\,.\overline{\sigma}, \rho)\} \vdash_f e : \pi, \varphi$*. Then we have* $TE \vdash_f e[x \mapsto v] : \pi, \varphi$

The λ_f^{region}-calculus is obviously not strongly normalizing. We can define a canonical well-typed diverging program. For arbitrary ϱ, we define

$$\Omega_\varrho = ((\mathtt{fix}\, f.\Lambda\,\varrho.\lambda\, x.\,(f\ [\varrho]\ \mathtt{at}\ \varrho)\,@\,x\ \mathtt{at}\ \varrho)\ [\varrho]\ \mathtt{at}\ \varrho)\,@\,(\lambda\, x.\, x\ \mathtt{at}\ \varrho)$$

We always have that $\Omega_\varrho \uparrow$ and for any $\overline{\sigma}$ also that $\{\ \} \vdash_f \Omega_\varrho : (\overline{\sigma}, \varrho), \{\varrho\}$.

The reduction relation $\rightharpoonup$ for the λ_f^{region}-calculus obeys a type preservation property:

Proposition 5.2.3 (Type Preservation) *Suppose* $TE \vdash_f e : \pi, \varphi$*. If* $e \rightharpoonup e'$ *then there exists an effect* φ' *such that* $TE \vdash_f e' : \pi, \varphi'$ *with* $\varphi' \subseteq \varphi$.

Proof. The proof is very similar to the proof of Proposition 3.3.7 on page 37. We omit the details. □

Equally, we have a canonical forms lemma for λ_f^{region}-Values:

Lemma 5.2.4 (Canonical Forms Lemma) *Suppose* $TE \vdash_f v : \pi, \emptyset$.

- *If* $\pi = (\pi_1 \xrightarrow{\epsilon.\varphi} \pi_2, \rho)$, *then* v *is of the form* $\langle \lambda\, x.\, e \rangle_\rho$.
- *If* $\pi = (\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \rho)$ *and* $n \geq 1$, *then* v *is of the form* $\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\rho$.

A progress property is also standard:

Proposition 5.2.5 (Progress) *Suppose* $\{\,\} \vdash_f e : \pi, \varphi$ *and* $Clean\,(\varphi)$, *then either* e *is a value* v *or there exists an* e' *such that* $e \rightharpoonup e'$.

Proof. The proof is very similar to the proof of Proposition 3.3.8 on page 39. We omit the details. □

Type soundness for the λ_f^{region}-calculus is now as follows:

Theorem 5.2.6 (Type Soundness of the λ_f^{region}-calculus) *Suppose* $\{\,\} \vdash_f e : \pi, \varphi$ *and* $Clean\,(\varphi)$. *Then,* $e \downarrow$ *or* $e \uparrow$.

Proof. Simple induction on the reduction of e using Proposition 5.2.3 and Proposition 5.2.5. □

The small-step reduction $\rightharpoonup$ is deterministic:

Lemma 5.2.7 *Suppose* $e \rightharpoonup e'$ *and* $e \rightharpoonup e''$, *then* $e' = e''$.

The relation is also invariant under expression substitution:

Lemma 5.2.8 *Suppose* $e \rightharpoonup e'$, *then* $e[x \mapsto v] \rightharpoonup e'[x \mapsto v]$.

Similarly, the reduction relation is invariant under region substitution, provided the substitution does not cause a dereference of dangling pointers.

Lemma 5.2.9 *Suppose* $TE \vdash_f e : \pi, \varphi$ *and for an* S_r *it holds that* $Clean\,(S_r(\varphi))$. *Then* $e \rightharpoonup e'$ *implies* $S_r(e) \rightharpoonup S_r(e')$.

As mentioned before, the use of the λ_f^{region}-calculus instead of the λ_s^{region}-calculus facilitates the formulation of an equational theory. However, the λ_f^{region}-calculus is a strict generalization of the λ_s^{region}-calculus of Section 3.2.

We define a translation function $[\![\cdot]\!] \in \lambda_s^{region}\text{-}\mathsf{Term} \to \lambda_f^{region}\text{-}\mathsf{Term}$ inductively on λ_s^{region}-Terms, where we omit the cases that only propagate the translation to sub-terms:

$$[\![\langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\rho\, [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho']\!] = \begin{cases} \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, [\![e']\!] \rangle_\rho\, [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' \\ \qquad \textit{if } e \equiv \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\rho\ \mathtt{in}\ e' \\ \langle \mathtt{fix}\, f''.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, [\![e]\!] \rangle_\rho\, [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' \\ \qquad \textit{otherwise, where } \ f'' \notin fv\,(e) \end{cases} \tag{5.13}$$

$$[\![\texttt{letrec}\ f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ e_1\ \texttt{at}\ \rho\ \texttt{in}\ e_2]\!] = \qquad (5.14)$$
$$\texttt{let}\ f = \texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e_1]\!]\ \texttt{at}\ \rho\ \texttt{in}\ [\![e_2]\!]$$

$$[\![\texttt{letrec}\ f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ e_1\rangle_\rho\ \texttt{in}\ e_2]\!] =$$
$$\begin{cases} [\![e_1]\!][f \mapsto \langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e_1]\!]\rangle_\rho] & \text{if } e_1 = e_2 \\ \texttt{let}\ f = \langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e_1]\!]\rangle_\rho\ \texttt{in}\ [\![e_2]\!] & \text{if } e_1 \neq e_2 \end{cases} \qquad (5.15)$$

We have the following two properties:

Proposition 5.2.10 *Suppose* $TE \vdash^t_t e : (\tau,\rho),\varphi$, *then* $TE \vdash_f [\![e]\!] : (\tau,\rho),\varphi$.

Proof. A straightforward induction on the derivation. □

Define $\rightharpoonup^{0,1}$ as the relation making one or no $\rightharpoonup$ reduction steps.

Proposition 5.2.11 *Suppose* $e \to e'$, *then* $[\![e]\!] \rightharpoonup^{0,1} [\![e']\!]$ *too.*

Proof. By induction on the reduction relation $\rightharpoonup$. All the cases that propagate the translation are immediate and so is the translation case (5.14).

For translation case (5.13), the second sub-case is trivial too, so, let us consider the first sub-case:

$$\langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ \texttt{letrec}\ f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ e\rangle_\rho\ \texttt{in}\ e\rangle_\varrho\ [\rho'_1,\ldots,\rho'_n]\ \texttt{at}\ \varrho' \to$$
$$\langle\lambda\,x.\ (\texttt{letrec}\ f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ e\rangle_\rho\ \texttt{in}\ e)[\varrho_1 \mapsto \rho'_1,\ldots,\varrho_n \mapsto \rho'_n]\rangle_{\varrho'}$$

The translated reduction is as follows:

$$\langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e]\!]\rangle_\varrho\ [\rho'_1,\ldots,\rho'_n]\ \texttt{at}\ \varrho' \rightharpoonup$$
$$\langle\lambda\,x.\ ([\![e]\!][f \mapsto \langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e]\!]\rangle_\varrho])[\varrho_1 \mapsto \rho'_1,\ldots,\varrho_n \mapsto \rho'_n]\rangle_{\varrho'}$$

It is easy to verify that this satisfies the claim.

For translation case (5.15), we first look at the reduction in the λ^{region}_s-calculus:

$$\texttt{letrec}\ f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ e_1\rangle_\rho\ \texttt{in}\ e_2 \to$$
$$e_2[f \mapsto \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ \texttt{letrec}\ f = \langle\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ e_1\rangle_\rho\ \texttt{in}\ e_1\rangle_\rho]$$

Whenever $e_1 \neq e_2$ the translated reduction is as follows:

$$\texttt{let}\ f = \langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e_1]\!]\rangle_\rho\ \texttt{in}\ [\![e_2]\!] \rightharpoonup$$
$$[\![e_2]\!][f \mapsto \langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e_1]\!]\rangle_\rho]$$

satisfying the claim. If we have that $e_1 \neq e_1$, then both sides of the reduction translate to the expression:

$$[\![e_1]\!][f \mapsto \langle\texttt{fix}\ f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\,x.\ [\![e_1]\!]\rangle_\rho]$$

This proves the proposition. □

5.3 Typed Relations and Open Extensions

Let $\mathsf{TypedRel} = \mathcal{P}(\mathsf{TypeEnv} \times \mathsf{Term} \times \mathsf{Term} \times \mathsf{TypeWPlace} \times \mathsf{Effect})$. A *typed relation* $\mathcal{R} \in \mathsf{TypedRel}$ is a set of quintuples $(TE, e_1, e_2, \pi, \varphi)$. We write $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ to say that $(TE, e_1, e_2, \pi, \varphi) \in \mathcal{R}$.

Definition 5.3.1 *A typed relation $\mathcal{R}$ is* well-formed *if the following two conditions hold simultaneously:*

1. *if $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ then $TE \vdash_f e_1 : \pi, \varphi_1$ and $TE \vdash_f e_2 : \pi, \varphi_2$ such that $\varphi_1 \cup \varphi_2 \subseteq \varphi$ with $Clean\,(\varphi)$ and $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi_1 \cup \varphi_2$.*
2. *if $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ then for all sets of effects φ' with $Clean\,(\varphi')$, also $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi \cup \varphi'$ holds.*

Well-formedness implies a type invariant together with a subsumption property for effect sets. A typed relation $\mathcal{R}$ is reflexive if $TE \vdash_f e : \pi, \varphi'$ and $\varphi' \subseteq \varphi$ with $Clean\,(\varphi)$ implies $TE \rhd e \,\mathcal{R}\, e : \pi, \varphi,$. Note that the empty relation is *not* reflexive. A typed relation $\mathcal{R}$ is transitive if $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$, and $TE \rhd e_2 \,\mathcal{R}\, e_3 : \pi, \varphi$, implies $TE \rhd e_1 \,\mathcal{R}\, e_3 : \pi, \varphi,$. Symmetry for a typed relation makes that $TE \rhd e_2 \,\mathcal{R}\, e_1 : \pi, \varphi$, whenever $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi,$.

We define $\mathcal{R}^{-1} = \left\{(TE, e_2, e_1, \pi, \varphi) \mid TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi\right\}$ as the inverse relation of $\mathcal{R}$. A typed relation $\mathcal{R}$ is *closed* ($\mathcal{R} \in \mathsf{ClosedTypedRel}$) iff $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ implies $TE = \{\,\}$.

It is sometimes necessary to extend a closed typed relation to a typed relation on non-closed terms. This leads to the notion of *open extension*.

First, define an *iterated substitution*, ξ, as a sequence of value substitutions, *i.e.* a finite function from Var to Value: $(x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$. We denote the empty sequence by ε. An iterated substitution works over λ_f^{region}-terms as follows:

$$\varepsilon(e) = e$$
$$(\xi, x \mapsto v)(e) = \xi(e[x \mapsto v])$$

An iterated substitution, ξ, is *a completion* of a type environment, TE, whenever $TE \succ \xi$ holds and $\succ$ is the least relation closed under the rules:

$$\{\,\} \succ \varepsilon \qquad \frac{TE(x) = (\sigma, \rho) \quad \overline{\sigma} \prec \sigma \;\; via \;\; (S_t, S_e, I_r) \quad TE - x \vdash_f v : (\overline{\sigma}, \rho), \emptyset \quad TE - x \succ \xi}{TE \succ (\xi, x \mapsto v)}$$

We have the following straightforward property for completions:

Lemma 5.3.1 *If $TE \vdash_f e : \pi, \varphi$ and $S_r(TE) \succ \xi$ for any region substitution S_r, then $\{\,\} \vdash_f \xi(S_r(e)) : S_r(\pi), S_r(\varphi)$.*

Proof. First, apply Lemma 5.2.1 on the type judgment and then prove by induction on ξ, using Lemma 5.2.2 for the induction step. □

The *open extension* of a well-formed closed typed relation $\mathcal{R}$ is a well-formed typed relation $\mathcal{R}^\circ$ such that

$$TE \rhd e_1 \,\mathcal{R}^\circ\, e_2 : \pi, \varphi \text{ iff } \begin{array}{l}\text{for all } \xi \text{ and for all } S_r \text{ where } S_r(TE) \succ \xi \text{ and } \mathit{Clean}\,(S_r(\varphi)) \\ \quad \text{it holds that } \{\,\} \rhd \xi(S_r(e_1))\,\mathcal{R}\,\xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)\end{array}$$

This definition may be a little surprising because of the quantified region substitution S_r. However, this somewhat stronger condition is related to the presence of region polymorphism in the λ_f^{region}-calculus. The condition implies that $\{\,\} \rhd e_1 \,\mathcal{R}^\circ\, e_2 : \pi, \varphi$ is not necessarily equivalent to $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ because of the substitution S_r. We will come back to this restricted view of open extensions in Section 5.6.2, where we introduce bisimilarity.

An iterated substitution can be formulated as a sequence of nested let-expressions:

$$\begin{array}{l}\mathcal{T}(\varepsilon, e) = e \\ \mathcal{T}((\xi, x \mapsto v), e) = \mathcal{T}(\xi, \mathtt{let}\ x = v\ \mathtt{in}\ e)\end{array}$$

We have a substitution lemma for open extensions of well-formed relations:

Lemma 5.3.2 *Suppose a well-formed closed typed relation $\mathcal{R}$. If $TE \vdash_f v : (\overline{\sigma}, \rho), \emptyset$ and $TE + \{x \mapsto (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}.\overline{\sigma}, \rho)\} \rhd e_1 \,\mathcal{R}^\circ\, e_2 : \pi, \varphi$ where $(\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap \mathit{fv}\,(TE) = \emptyset$, then $TE \rhd e_1[x \mapsto v] \,\mathcal{R}^\circ\, e_2[x \mapsto v] : \pi, \varphi$.*

Proof. First, we note that without loss of generality, we can assume that $x \notin \mathit{Dom}\,(TE)$. By definition of open extension, take ξ and S_r such that $S_r(TE) \succ \xi$ with $\mathit{Clean}\,(S_r(\varphi))$. So, we have to show that $\{\,\} \rhd \xi(S_r(e_1[x \mapsto v]))\,\mathcal{R}\,\xi(S_r(e_2[x \mapsto v])) : S_r(\pi), S_r(\varphi)$, which is by definition of iterated substitution the same as $\{\,\} \rhd (\xi, x \mapsto S_r(v))(S_r(e_1))\,\mathcal{R}\,(\xi, x \mapsto S_r(v))(S_r(e_1)) : S_r(\pi), S_r(\varphi)$.

We have by assumption that $TE \vdash_f v : (\overline{\sigma}, \rho), \emptyset$ and therefore by Lemma 5.2.1 that $S_r(TE) \vdash_f S_r(v) : (S_r(\overline{\sigma}), S_r(\rho)), \emptyset$. Take S_t and S_e such that $\mathit{Supp}\,(S_t) = \vec{\alpha}$, $\mathit{Supp}\,(S_e) = \vec{\epsilon}$ and $S_r(\overline{\sigma}) \prec S_r(\sigma)$ *via* (S_t, S_e, I_r). Since we also have $S_r(TE) \succ \xi$, by definition of completion we have that $S_r(TE + \{x \mapsto (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}.\overline{\sigma}, \rho)\}) \succ (\xi, x \mapsto S_r(v))$.

Therefore, since we also have by assumption that $TE + \{x \mapsto (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}.\overline{\sigma}, \rho)\} \rhd e_1 \,\mathcal{R}^\circ\, e_2 : \pi, \varphi$, the claim $\{\,\} \rhd (\xi, x \mapsto S_r(v))(S_r(e_1))\,\mathcal{R}\,(\xi, x \mapsto S_r(v))(S_r(e_1)) : S_r(\pi), S_r(\varphi)$ follows. □

5.4 Contextual Equivalence

In Section 2.2.4, we introduced the notion of *contextual equivalence* for the λ^{cbv}-calculus. In this section, we define contextual equivalence for the λ_f^{region}-calculus with a technique due to Lassen [87]. It is slightly more elegant and more general than Definition 2.2.2 on page 19.

We start with the introduction of a compatible refinement operator for typed relations. Using this operator, we define λ_f^{region}-contexts by means of *context closures*. This allows for a general and elegant definition of contextual equivalence.

5.4.1 Compatible Refinement

Given a typed relation $\mathcal{R}$, we define its *compatible refinement* $\widehat{\mathcal{R}} \in \mathsf{TypedRel}$ as the least relation closed under the rules of Figure 5.5. The refinement rules are pretty much binary extensions of the type rules in Figure 5.4 without rule *(F-Effect)*. However, to assure that the compatible extension does not change the typing, we must guarantee that superfluous effect variables can be eliminated in the rules *(Comp-Reglam)*, *(Comp-App)*, *(Comp-Let)* and *(Comp-Regapp)*. We say that a typed relation $\mathcal{R}$ is *compatible* iff $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. Note that the empty relation is not compatible due to rule *(Comp-Var)*.

The main peculiarity of the compatible refinement definition is that the effects are always clean. This restriction is not unreasonable: for instance, a region annotated program produced by a compiler will obey this property.

Additionally, we found the requirement necessary to have a usable notion of contextual equivalence in the first place. Let us explain this with an example.

Suppose the type with place $\pi = ((\alpha, \varrho') \xrightarrow{\epsilon.\{\}} (\alpha, \varrho'), \varrho')$ and consider the following assertion, relating two expressions of type π:

$$\{\,\} \rhd (\langle \lambda\, x.\, x \rangle_{\varrho} \mathbin{@} (\lambda\, w.\, w\ \mathtt{at}\ \varrho'))\, \mathcal{R}\, (\lambda\, w.\, w\ \mathtt{at}\ \varrho') : \pi, \varphi \tag{5.16}$$

Assume now that we do not require clean effects for contexts. For a compatible $\mathcal{R}$, we would then be able to derive the following context assertion of (5.16):

$$\begin{aligned} \{\,\} \rhd (\langle \mathtt{fix}\, f.\Lambda\, \varrho.\lambda\, z.\, (\langle \lambda\, x.\, x \rangle_{\varrho} \mathbin{@} (\lambda\, w.\, w\ \mathtt{at}\ \varrho'))\rangle_{\varrho''}\, [\bullet]\ \mathtt{at}\ \varrho'') \mathbin{@} \langle \lambda\, z.\, z \rangle_{\varrho''}\, \mathcal{R} \\ (\langle \mathtt{fix}\, f.\Lambda\, \varrho.\lambda\, z.\, (\underline{\lambda\, w.\, w\ \mathtt{at}\ \varrho'})\rangle_{\varrho''}\, [\bullet]\ \mathtt{at}\ \varrho'') \mathbin{@} \langle \lambda\, z.\, z \rangle_{\varrho''} : \pi, \varphi' \end{aligned} \tag{5.17}$$

Although it is quite natural to expect the expressions in (5.16) to be *equivalent*, for instance, they both evaluate to $\langle \lambda\, w.\, w \rangle_{\varrho'}$, it is easy to see that the two expressions in (5.17) are not. Indeed, the left-hand side expression does not reduce to any value - it tries to dereference $\bullet$, whereas the right-hand side evaluates to $\langle \lambda\, w.\, w \rangle_{\varrho'}$.

The actual reason for our definition of compatibility (by only considering clean effects) is that well typedness alone does not guarantee progress as witnessed by the soundness Theorem 5.2.6. Clean effects *belong* in some sense to well typedness.

A compatible refinement has two useful properties, which are proven by rule induction on its definition.

Lemma 5.4.1 *Consider a typed relation $\mathcal{R}$.*

1. *If $\mathcal{R}$ is compatible, then $\mathcal{R}$ is reflexive.*
2. *If $\mathcal{R}$ is well-formed, then $\widehat{\mathcal{R}}$ is well-formed as well.*

Additionally, we have that the open extension of a compatible $\mathcal{R}$ can be encoded within $\mathcal{R}$ itself:

Lemma 5.4.2 *Suppose a typed relation $\mathcal{R}^\circ$ which is well-formed and compatible. If for all S_r and ξ we have $S_r(TE) \succ \xi$ with $Clean\,(S_r(\varphi))$ and $TE \rhd e_1\, \mathcal{R}^\circ\, e_2 : \pi, \varphi$, then we also have that $\{\,\} \rhd \mathcal{T}(\xi, S_r(e_1))\, \mathcal{R}\, \mathcal{T}(\xi, S_r(e_2)) : S_r(\pi), S_r(\varphi)$.*

Proof. Proven by simple induction on the length of ξ and using rule *(Comp-Let)* and Lemma 5.2.1 in the induction step. □

(Comp-Var)
$$\frac{TE(x) = (\sigma, \rho) \quad \overline{\sigma} \prec \sigma \ \mathit{via}\ (S_t, S_e, I_r) \quad \mathit{Clean}\,(\varphi)}{TE \rhd x \,\widehat{\mathcal{R}}\, x : (\overline{\sigma}, \rho), \varphi}$$

(Comp-Lam)
$$\frac{TE + \{x \mapsto \pi_1\} \rhd e \,\mathcal{R}\, e' : \pi_2, \varphi' \quad \mathit{Clean}\,(\varphi)}{TE \rhd \lambda\, x.\ e \ \mathtt{at}\ \varrho \,\widehat{\mathcal{R}}\, \lambda\, x.\ e' \ \mathtt{at}\ \varrho : (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2, \varrho), \{\varrho\} \cup \varphi}$$

(Comp-Lamv)
$$\frac{TE + \{x \mapsto \pi_1\} \rhd e \,\mathcal{R}\, e' : \pi_2, \varphi' \quad \mathit{Clean}\,(\varphi)}{TE \rhd \langle \lambda\, x.\ e \rangle_\rho \,\widehat{\mathcal{R}}\, \langle \lambda\, x.\ e' \rangle_\rho : (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2, \rho), \varphi}$$

(Comp-Reglam)
$$\frac{TE \rhd e \,\mathcal{R}\, e' : \pi, \varphi' \quad \varrho \notin \mathit{frv}\,(TE, \pi) \quad \mathit{Clean}\,(\varphi) \quad \epsilon \notin \mathit{fev}\,(TE, \pi)}{TE \rhd \mathtt{new}\ \varrho.\, e \,\widehat{\mathcal{R}}\, \mathtt{new}\ \varrho.\, e' : \pi, (\varphi' \setminus \{\varrho, \epsilon\}) \cup \varphi}$$

(Comp-App)
$$\frac{\begin{array}{c} TE \rhd e_1 \,\mathcal{R}\, e_1' : (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2, \varrho), \varphi_1 \\ \mathit{Clean}\,(\varphi) \quad TE \rhd e_2 \,\mathcal{R}\, e_2' : \pi_1, \varphi_2 \quad \epsilon' \notin \mathit{fev}\,(TE, \pi_2) \end{array}}{TE \rhd e_1 \mathbin{@} e_2 \,\widehat{\mathcal{R}}\, e_1' \mathbin{@} e_2' : \pi_2, (\varphi' \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \varrho\} \cup \varphi) \setminus \{\epsilon'\}}$$

(Comp-Let)
$$\frac{\begin{array}{c} TE \rhd e_1 \,\mathcal{R}\, e_1' : (\overline{\sigma}', \rho'), \varphi_1 \\ (\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap \mathit{fv}\,(TE) = \emptyset \quad \mathit{Clean}\,(\varphi) \quad \epsilon' \notin \mathit{fev}\,(TE, \pi) \\ TE + \{x \mapsto (\forall\, \vec{\alpha}\, . \forall\, \vec{\epsilon}\, . \overline{\sigma}', \rho')\} \rhd e_2 \,\mathcal{R}\, e_2' : \pi, \varphi_2 \end{array}}{TE \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \,\widehat{\mathcal{R}}\, \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2' : \pi, (\varphi_1 \cup \varphi_2 \cup \varphi) \setminus \{\epsilon'\}}$$

(Comp-Fix)
$$\frac{\begin{array}{c} (\{\varrho_1, \ldots, \varrho_n\} \cup \{\vec{\epsilon}\}) \cap (\mathit{fv}\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}\, . \forall\, \varrho_1, \ldots, \varrho_n\, . \tau \quad \mathit{Clean}\,(\varphi) \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \rhd \lambda\, x.\ e \ \mathtt{at}\ \rho \,\mathcal{R}\, \lambda\, x.\ e' \ \mathtt{at}\ \rho : (\tau, \rho), \varphi' \end{array}}{\begin{array}{l} TE \rhd \mathtt{fix}\, f. \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ e \ \mathtt{at}\ \rho \,\widehat{\mathcal{R}} \\ \qquad \mathtt{fix}\, f. \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ e' \ \mathtt{at}\ \rho : (\forall\, \varrho_1, \ldots, \varrho_n\, . \tau, \rho), \varphi' \cup \varphi \end{array}}$$

(Comp-Fixv)
$$\frac{\begin{array}{c} (\{\varrho_1, \ldots, \varrho_n\} \cup \{\vec{\epsilon}\}) \cap (\mathit{fv}\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}\, . \forall\, \varrho_1, \ldots, \varrho_n\, . \tau \quad \mathit{Clean}\,(\varphi) \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \rhd \langle \lambda\, x.\ e \rangle_\rho \,\mathcal{R}\, \langle \lambda\, x.\ e' \rangle_\rho : (\tau, \rho), \varphi' \end{array}}{\begin{array}{l} TE \rhd \langle \mathtt{fix}\, f. \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ e \rangle_\rho \,\widehat{\mathcal{R}} \\ \qquad \langle \mathtt{fix}\, f. \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ e' \rangle_\rho : (\forall\, \varrho_1, \ldots, \varrho_n\, . \tau, \rho), \varphi' \cup \varphi \end{array}}$$

(Comp-Regapp)
$$\frac{\begin{array}{c} TE \rhd e \,\mathcal{R}\, e' : (\forall\, \varrho_1, \ldots, \varrho_n\, . \tau, \varrho), \varphi' \\ \mathit{Clean}\,(\varphi) \quad S_r = \{\varrho_1 \mapsto \rho_1', \ldots, \varrho_n \mapsto \rho_n'\} \quad \epsilon \notin \mathit{fev}\,(TE, \tau) \end{array}}{\begin{array}{l} TE \rhd e\ [\rho_1', \ldots, \rho_n']\ \mathtt{at}\ \varrho' \,\widehat{\mathcal{R}} \\ \qquad e'\ [\rho_1', \ldots, \rho_n']\ \mathtt{at}\ \varrho' : (S_r(\tau), \varrho'), (\varphi' \cup \{\varrho, \varrho'\} \cup \varphi) \setminus \{\epsilon\} \end{array}}$$

Figure 5.5: Compatible refinement of typed relations

5.4.2 Context Closure

The compatible refinement operator builds arbitrary non-empty contexts. However, since we also need empty contexts, we define the *context closure* $\mathcal{R}^c$ of a typed relation $\mathcal{R}$ as the least relation, closed under the rules

$$\textit{(Closure-Empty)}\quad \frac{TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi}{TE \rhd e_1 \,\mathcal{R}^c\, e_2 : \pi, \varphi} \qquad \textit{(Closure-Comp)}\quad \frac{TE \rhd e_1 \,\widehat{\mathcal{R}^c}\, e_2 : \pi, \varphi}{TE \rhd e_1 \,\mathcal{R}^c\, e_2 : \pi, \varphi}$$

Intuitively, a context closure of a typed relation includes all well-typed contexts of the related λ_f^{region}-expressions. Therefore, the context closure of the one-element relation $\{(TE, e_1, e_2, \pi, \varphi)\}$ relates all typed contexts of the expressions e_1 and e_2. The following properties trivially hold:

Lemma 5.4.3 *Suppose a typed relation $\mathcal{R}$.*

1. *If $\mathcal{R}$ is well-formed, then so is $\mathcal{R}^c$.*
2. *The context closure $\mathcal{R}^c$ is compatible, and therefore, also reflexive.*
3. *Compatible refinement and context closure are monotone operators in the complete lattice $(\mathsf{TypedRel}, \subseteq)$.*
4. *The relation $\mathcal{R}$ is compatible iff $\mathcal{R}^c \subseteq \mathcal{R}$.*

One-element context closures have the following useful properties, all shown by induction on the derivations:

Lemma 5.4.4
1. *$TE' \rhd e' \left\{(TE, e, e, \pi, \varphi)\right\}^c e'' : \pi', \varphi'$ implies $e' = e''$.*
2. *$TE' \rhd e'_1 \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e'_2 : \pi', \varphi'$ implies $TE' \rhd e'_2 \left\{(TE, e_2, e_1, \pi, \varphi)\right\}^c e'_1 : \pi', \varphi'$.*
3. *$TE' \rhd e'_1 \left\{(TE, e_1, e_3, \pi, \varphi)\right\}^c e'_3 : \pi', \varphi'$ implies that for all e_2, there exist e'_2 with $TE' \rhd e'_1 \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e'_2 : \pi', \varphi'$ and $TE' \rhd e'_2 \left\{(TE, e_2, e_3, \pi, \varphi)\right\}^c e'_3 : \pi', \varphi'$.*
4. *$TE' \rhd e'_1 \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e'_2 : \pi', \varphi'$ and $TE'' \rhd e''_1 \left\{(TE', e'_1, e'_2, \pi', \varphi')\right\}^c e''_2 : \pi'', \varphi''$ imply that $TE'' \rhd e''_1 \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e''_2 : \pi'', \varphi''$.*

We can use the context closure of a typed relation to give an alternative characterization of compatibility provided the relation is a pre-order and well-formed:

Proposition 5.4.5 *Suppose a typed relation $\mathcal{R}$ which is reflexive, transitive and well-formed. Then, $\mathcal{R}$ is compatible iff $TE' \rhd e'_1 \,\mathcal{R}\, e'_2 : \pi', \varphi'$ whenever $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ and $TE' \rhd e'_1 \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e'_2 : \pi', \varphi'$.*

Proof.

Case $\Rightarrow$. Take $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$, so $\{(TE, e_1, e_2, \pi, \varphi)\} \subseteq \mathcal{R}$. By Lemma 5.4.3.3, we have that $\{(TE, e_1, e_2, \pi, \varphi)\}^c \subseteq \mathcal{R}^c$. And since $\mathcal{R}$ is compatible, we also have $\mathcal{R}^c \subseteq \mathcal{R}$ by Lemma 5.4.3.4. This implies $\{(TE, e_1, e_2, \pi, \varphi)\}^c \subseteq \mathcal{R}$ proving the case.

Case $\Leftarrow$. For the other direction, we have to prove that $TE \rhd e_1 \,\widehat{\mathcal{R}}\, e_2 : \pi, \varphi$ implies $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$, which is shown by case analysis:

Case *(Comp-Var)*. Immediate since rule *(F-Var)* implies $TE \vdash_f x : (\hat{\sigma}, \rho), \emptyset$ and we are given that $\mathcal{R}$ is reflexive.

Case *(Comp-Let)*. Suppose $TE \rhd \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 \,\widehat{\mathcal{R}}\, \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2' : \pi, \varphi$. Then $TE \rhd e_1 \,\mathcal{R}\, e_1' : (\overline{\sigma}', \rho'), \varphi_1$ and $TE + \{x \mapsto (\forall\, \vec{\alpha}\, . \forall\, \vec{\epsilon}\, . \overline{\sigma}', \rho')\} \rhd e_2 \,\mathcal{R}\, e_2' : \pi, \varphi_2$ with $(\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap fv\,(TE) = \emptyset$ and $\varphi_1 \cup \varphi_2 \subseteq \varphi$.

- Take $\mathcal{R}' = \{(TE, e_1, e_1', (\overline{\sigma}', \rho'), \varphi_1)\}$. By *(Closure-Empty)* we have that $TE \rhd e_1 \,\mathcal{R}'^c\, e_1' : (\overline{\sigma}', \rho'), \varphi_1$. Since $\mathcal{R}$ is well-formed and by Lemma 5.4.3.2, we have $TE + \{x \mapsto (\forall\, \vec{\alpha}\, . \forall\, \vec{\epsilon}\, . \overline{\sigma}', \rho')\} \rhd e_2 \,\mathcal{R}'^c\, e_2 : \pi, \varphi_2$. Hence, using rule *(Comp-Let)* we have $TE \rhd \mathtt{let} x = e_1 \mathtt{in} e_2 \,\widehat{\mathcal{R}'^c}\, \mathtt{let} x = e_1' \mathtt{in} e_2 : \pi, \varphi$. Rule *(Closure-Comp)* and the assumption give $TE \rhd \mathtt{let} x = e_1 \mathtt{in} e_2 \,\mathcal{R}\, \mathtt{let} x = e_1' \,\mathtt{in}\, e_2 : \pi, \varphi$.
- By analogous reasoning, we also derive $TE \rhd \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2 \,\mathcal{R}\, \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2' : \pi, \varphi$.

By transitivity of $\mathcal{R}$, we conclude that $TE \rhd \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 \,\mathcal{R}\, \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2' : \pi, \varphi$, which proves the case.

All other cases are proven by similar reasoning.

□

5.4.3 Contextual Equivalence

We now have enough material to introduce a general notion of contextual equivalence for the λ_f^{region}-calculus.

We say that a typed relation $\mathcal{R}$ is *adequate* iff for all e_1, e_2, π and φ and $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ it holds that $e_1 \downarrow\!\Leftrightarrow e_2 \downarrow$.

Requiring that the context closure of a one-element relation is adequate yields the following definition of contextual equivalence:

Definition 5.4.1 (Contextual Equivalence for the λ_f^{region}-calculus)
Let contextual equivalence, $\simeq \in \mathsf{TypedRel}$, be a well-formed typed relation on λ_f^{region}-$\mathsf{Term}s$ such that $TE \rhd e_1 \simeq e_2 : \pi, \varphi$ iff $\{(TE, e_1, e_2, \pi, \varphi)\}^c$ is adequate.

This definition of contextual equivalence is based on the reduction semantics of the λ_f^{region}-calculus (see Section 5.2.2). We note that this choice is not fundamental, since the λ_f^{region}-calculus semantics is reminiscent of the λ_s^{region}-calculus semantics, which we have

proven equivalent in some sense to the big-step evaluation style formulation of Tofte and Talpin (see Section 4.4 and Section 4.5).

Lemma 5.4.6 *Contextual equivalence is well-defined,* i.e. *an equivalence relation.*

Proof. Reflexivity follows from Lemma 5.4.4.1, symmetry from Lemma 5.4.4.2. For transitivity, suppose $TE \rhd e_1 \simeq e_2 : \pi, \varphi$ and $TE \rhd e_2 \simeq e_3 : \pi, \varphi$. By showing that for $\{\,\} \rhd e_1' \left\{(TE, e_1, e_3, \pi, \varphi)\right\}^c e_3' : \pi', \varphi'$ it holds that $e_1' \downarrow\!\!\Leftrightarrow e_3' \downarrow$, we have that $TE \rhd e_1 \simeq e_3 : \pi, \varphi$, *i.e.* transitivity.

Suppose now that $e_1' \downarrow$. Using Lemma 5.4.4.3, we have for e_2, an e_2' such that $\{\,\} \rhd e_1' \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e_2' : \pi', \varphi'$ and $\{\,\} \rhd e_2' \left\{(TE, e_2, e_3, \pi, \varphi)\right\}^c e_3' : \pi', \varphi'$. Because $e_1' \downarrow$ and $TE \rhd e_1 \simeq e_2 : \pi, \varphi$, it must be that $e_2' \downarrow$. This, together with $TE \rhd e_2 \simeq e_3 : \pi, \varphi$ implies $e_3' \downarrow$. Symmetrically, we show that $e_3' \downarrow\!\!\Rightarrow e_1' \downarrow$, proving the case. □

Proposition 5.4.7 *Contextual equivalence is the greatest typed relation that is well-formed, compatible and adequate.*

Proof.

1. By rule *(Closure-Empty)*, we immediately have that contextual equivalence is adequate and by definition well-formed.

   Using Proposition 5.4.5 and Lemma 5.4.6, we show that contextual equivalence is compatible by proving that $TE' \rhd e_1' \simeq e_2' : \pi', \varphi'$ whenever $TE \rhd e_1 \simeq e_2 : \pi, \varphi$ and $TE' \rhd e_1' \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e_2' : \pi', \varphi'$.

   Since $TE \rhd e_1 \simeq e_2 : \pi, \varphi$, we obviously have that $\left\{(TE, e_1, e_2, \pi, \varphi)\right\}$ is well-formed and, by Lemma 5.4.3.1, $\left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c$ is also well-formed. For adequacy of $\left\{(TE', e_1', e_2', \pi', \varphi')\right\}^c$, assume $\{\,\} \rhd e_1'' \left\{(TE', e_1', e_2', \pi', \varphi')\right\}^c e_2'' : \pi'', \varphi''$. By Lemma 5.4.4.4 and the assumptions, $\{\,\} \rhd e_1'' \left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c e_2'' : \pi'', \varphi''$ holds. Also by assumption, we have $TE \rhd e_1 \simeq e_2 : \pi, \varphi$ and therefore $e_1'' \downarrow\!\!\Leftrightarrow e_2'' \downarrow$, showing that $\left\{(TE', e_1', e_2', \pi', \varphi')\right\}^c$ is adequate.

2. To show that $\simeq$ is the greatest well-formed, compatible and adequate typed relation, assume a typed relation $\mathcal{R}$, which is also well-formed, compatible and adequate. Take $TE \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$.

   Since $\mathcal{R}$ is compatible, we have from Lemma 5.4.3.3 and Lemma 5.4.3.4 that $\left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c \subseteq \mathcal{R}$. Because $\mathcal{R}$ is also adequate, we derive that the relation $\left\{(TE, e_1, e_2, \pi, \varphi)\right\}^c$ is adequate, from which we can conclude that $TE \rhd e_1 \simeq e_2 : \pi, \varphi$.

□

5.5 A Syntactic Equational Theory

Although contextual equivalence is a very powerful notion for term-equality, it is not very practical to use. In this section, we present a purely syntactic theory built on a set of typed equational rules. The rest of the chapter shows that the theory is sound with respect to contextual equivalence. Define the relation $\triangleq$ as the least typed relation closed under the rules in Figure 5.6.

Rule *(Eq-Comp)* guarantees that the $\triangleq$ is compatible. As a result of Lemma 5.4.1.1 it follows that the relation $\triangleq$ is reflexive as well. The rules *(Eq-Symm)* and *(Eq-Trans)* make the relation an equivalence.

The beta-reduction rules (5.1), (5.2), (5.5) and (5.6) become typed and bidirectional in rule *(Eq-Beta-n)*. The index n refers to the corresponding reduction rule in Figure 5.2.

Rule *(Eq-App-Cbv)* equates call-by-value function application with a monomorphic `let`-expression. The rules *(Eq-App-Let)*, *(Eq-Let-Let)* and *(Eq-Drop)* express useful transformations that we exploit in Chapter 6 to prove the soundness of a specializer. Rule *(Eq-App-Let)* is a `let`-insertion rule for function application, which makes the left-to-right evaluation of the λ_f^{region}-calculus more explicit. Rule *(Eq-Let-Let)* is standard `let`-flattening [106] and allows for the un-nesting of `let`-expressions. The rule *(Eq-Drop)* specifies that the allocation of a region in a safe place (guaranteed by the condition $\varrho \notin frv\,(TE, \pi)$) does not change the meaning of an expression. Of course, it is hoped that region inference inserts such allocations as much and as early as possible. In fact, rule *(Eq-Drop)* optimizes memory use when used from left to right, and deteriorates it when used from right to left. Finally, rule *(Eq-Dealloc)* expresses that deallocation, when it is safe to do so, *does not matter*.

Note that we do not have the equational rules *(Eq-Beta-3)* and *(Eq-Beta-4)*. The former is covered by the rules *(Eq-Drop)* and *(Eq-Dealloc)* and the latter by the rules *(Eq-App-Cbv)* and *(Eq-Beta-5)*.

In comparison, the typed equational theory for a region calculus by Dal Zilio and Gordon [160] is monomorphic and is subtly different due their presentation of the type system, which records allocated region variables. As a consequence, the rule *(Eq-Drop)* from Figure 5.6 would not be well-formed in their system. In contrast, in our system, the following rules from their theory are derivable:

$$\textit{(Eq-Swap)}\quad \frac{TE \vdash_f e : \pi, \varphi' \quad \{\varrho, \varrho'\} \cap frv\,(TE, \pi) = \emptyset \quad \varphi' \setminus \{\varrho, \varrho'\} \subseteq \varphi \quad Clean\,(\varphi)}{TE \rhd \texttt{new}\,\varrho.\,\texttt{new}\,\varrho'.\,e \triangleq \texttt{new}\,\varrho'.\,\texttt{new}\,\varrho.\,e : \pi, \varphi}$$

$$\textit{(Eq-Rlet)}\quad \frac{\begin{array}{c} TE \vdash_f e_1 : (\overline{\sigma}', \rho'), \varphi_1 \quad (\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap fv\,(TE) = \emptyset \\ (\varphi_1 \cup \varphi_2) \setminus \{\varrho\} \subseteq \varphi \quad Clean\,(\varphi) \quad \varrho \notin frv\,(TE, (\overline{\sigma}', \rho'), \pi) \\ TE + \{x \mapsto (\forall\,\vec{\alpha}\,.\forall\,\vec{\epsilon}.\overline{\sigma}', \rho')\} \vdash_f e_2 : \pi, \varphi_2 \end{array}}{TE \rhd \texttt{new}\,\varrho.\,(\texttt{let}\,x = e_1\,\texttt{in}\,e_2) \triangleq \texttt{let}\,x = (\texttt{new}\,\varrho.\,e_1)\,\texttt{in}\,(\texttt{new}\,\varrho.\,e_2) : \pi, \varphi}$$

$$\textit{(Eq-Rlam)}\quad \frac{\begin{array}{c} TE + \{x \mapsto \pi_1\} \vdash_f e : \pi_2, \varphi' \quad \varphi' \subseteq \varphi'' \\ \varrho \notin frv\,(TE, (\pi_1 \xrightarrow{\epsilon.\varphi''} \pi_2, \varrho')) \quad \varrho' \in \varphi \quad Clean\,(\varphi) \end{array}}{TE \rhd \texttt{new}\,\varrho.\,(\lambda\,x.\,e\ \texttt{at}\,\varrho') \triangleq \lambda\,x.\,(\texttt{new}\,\varrho.\,e)\ \texttt{at}\,\varrho' : (\pi_1 \xrightarrow{\epsilon.\varphi''} \pi_2, \varrho'), \varphi}$$

$$\textit{(Eq-Comp)} \quad \frac{TE \rhd e \mathrel{\widehat{\triangleq}} e' : \pi, \varphi}{TE \rhd e \triangleq e' : \pi, \varphi}$$

$$\textit{(Eq-Symm)} \quad \frac{TE \rhd e' \triangleq e : \pi, \varphi}{TE \rhd e \triangleq e' : \pi, \varphi}$$

$$\textit{(Eq-Trans)} \quad \frac{TE \rhd e \triangleq e' : \pi, \varphi \quad TE \rhd e' \triangleq e'' : \pi, \varphi}{TE \rhd e \triangleq e'' : \pi, \varphi}$$

$$\textit{(Eq-Beta-n)} \quad \frac{TE \vdash_f e : \pi, \varphi' \quad e \xrightarrow{n} e' \quad \varphi' \subseteq \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd e \triangleq e' : \pi, \varphi} \quad n \in \{1, 2, 5, 6\}$$

$$\textit{(Eq-App-Cbv)} \quad \frac{\begin{array}{c} TE \vdash_f \langle \lambda\, x.\, e_1 \rangle_\varrho : (\pi_1 \xrightarrow{\epsilon.\varphi''} \pi, \varrho), \emptyset \\ TE \vdash_f e_2 : \pi_1, \varphi' \quad \varphi' \cup \varphi'' \cup \{\epsilon, \varrho\} \subseteq \varphi \quad \mathit{Clean}\,(\varphi) \end{array}}{TE \rhd \langle \lambda\, x.\, e_1 \rangle_\varrho \mathbin{@} e_2 \triangleq \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 : \pi, \varphi}$$

$$\textit{(Eq-App-Let)} \quad \frac{\begin{array}{c} TE \vdash_f e_1 : (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi, \varrho), \varphi_1 \quad f \notin \mathit{fv}\,(e_2) \\ TE \vdash_f e_2 : \pi_1, \varphi_2 \quad \varphi_1 \cup \varphi_2 \cup \varphi' \cup \{\epsilon, \varrho\} \subseteq \varphi \quad \mathit{Clean}\,(\varphi) \end{array}}{TE \rhd e_1 \mathbin{@} e_2 \triangleq \mathtt{let}\ f = e_1\ \mathtt{in}\ f \mathbin{@} e_2 : \pi, \varphi}$$

$$\textit{(Eq-Let-Let)} \quad \frac{\begin{array}{c} TE \vdash_f e_1 : (\overline{\sigma}_1, \rho_1), \varphi_1 \quad (\{\vec{\alpha}^1\} \cup \{\vec{\epsilon}^1\}) \cap \mathit{fv}\,(TE) = \emptyset \quad \varphi_1 \cup \varphi_2 \cup \varphi_3 \subseteq \varphi \\ TE + \{x \mapsto (\forall\, \vec{\alpha}^1\, .\forall\, \vec{\epsilon}^1\, .\overline{\sigma}_1, \rho_1)\} \vdash_f e_2 : (\overline{\sigma}_2, \rho_2), \varphi_2 \quad x \notin \mathit{frv}\,(e_3) \quad \mathit{Clean}\,(\varphi) \\ (\{\vec{\alpha}^2\} \cup \{\vec{\epsilon}^2\}) \cap \mathit{fv}\,(TE) = \emptyset \quad TE + \{y \mapsto (\forall\, \vec{\alpha}^2\, .\forall\, \vec{\epsilon}^2\, .\overline{\sigma}_2, \rho_2)\} \vdash_f e_3 : \pi, \varphi_3 \end{array}}{\begin{array}{l} TE \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ (\mathtt{let}\ y = e_2\ \mathtt{in}\ e_3) \triangleq \\ \qquad \mathtt{let}\ y = (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)\ \mathtt{in}\ e_3 : \pi, \varphi \end{array}}$$

$$\textit{(Eq-Drop)} \quad \frac{TE \vdash_f e : \pi, \varphi' \quad \varrho \notin \mathit{frv}\,(TE, \pi) \quad \varphi' \subseteq \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd e \triangleq \mathtt{new}\ \varrho.\, e : \pi, \varphi}$$

$$\textit{(Eq-Dealloc)} \quad \frac{TE \vdash_f e : \pi, \varphi' \quad \varrho \notin \mathit{frv}\,(\pi) \cup \varphi' \quad \varphi' \subseteq \varphi \quad \mathit{Clean}\,(\varphi)}{TE \rhd e \triangleq e[\varrho \mapsto \bullet] : \pi, \varphi}$$

Figure 5.6: An equational theory for the λ_f^{region}-calculus

Dal Zilio and Gordon also specify an equational rule (Eq Drop). However, it is weaker than *(Eq-Drop)* and hence, they have to specify variants of the rules *(Eq-Swap)* and *(Eq-Rlet)* as axioms in their equational theory.

Lemma 5.5.1 *The relation* $\triangleq$ *is well-formed.*

Proof. By induction on the derivation, using Lemma 5.4.1.2, Proposition 5.2.3 and inspecting the premises. □

We also have that the encoding of a completion can be equated to the application of the completion to an expression:

Lemma 5.5.2 *If* $TE \vdash_f e : \pi, \varphi$ *and* $S_r(TE) \succ \xi$ *with* $Clean\,(S_r(\varphi))$ *then* $\{\,\} \rhd \xi(S_r(e)) \triangleq \mathcal{T}(\xi, S_r(e)) : S_r(\pi), S_r(\varphi)$.

Proof. A simple inductive argument on the length of ξ using equational rule *(Eq-Beta-5)* with Lemma 5.2.1 and Lemma 5.2.9. □

The equational theory as specified above is formulated for the λ_f^{region}-calculus and includes intermediate computational terms only required for the small-step transition semantics of Figure 5.2 and Figure 5.3. In practice, it is often desirable to prove program transformations on non-computational terms, *i.e.* terms which are used by a programmer or compiler. It is possible to formulate an equational theory directly for such terms, however, the rules would be clumsier and technically difficult to prove sound. Alternatively, such rules can be *derived* from the rules in Figure 5.6. We omit the details.

While the rest of the chapter deals with the soundness of the theory with respect to contextual equivalence, we note that the equational rules are not complete: they are not sufficient to show semantic equality of arbitrary programs. The latter would require a co-inductively defined equational system. However, the rules are sufficient to prove the soundness of a specializer (as we shall see in Chapter 6).

5.6 Bisimilarity for the λ_f^{region}-calculus

The abstract property that two programs are contextual equivalent if no context can distinguish them, is intuitive, but does not yield a practical proof principle. We specify a form of *applicative bisimilarity* to prove program equivalences in the λ_f^{region}-calculus and use it to prove soundness of $\triangleq$.

5.6.1 A Labeled Transition System

Following Crole and Gordon [29, 50], we use a labeled transition system to describe the observations one can make of a program. Observations are defined by the following grammar:

$$\mathsf{Observation} \ni \gamma ::= @\,v \mid [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \bullet$$

$$\textit{(Obs-Red)}\quad \frac{\{\,\}\vdash_f e:\pi,\varphi'' \quad \varphi''\subseteq\varphi \qquad e \rightharpoonup e'' \quad (e'':\pi,\varphi)\xrightarrow{\gamma}(e':\pi',\varphi')}{(e:\pi,\varphi)\xrightarrow{\gamma}(e':\pi',\varphi')}$$

$$\textit{(Obs-Lam)}\quad \frac{\{\,\}\vdash_f \langle\lambda\, x.\, e\rangle_\varrho : (\pi''\xrightarrow{\epsilon.\varphi'}\pi',\varrho),\emptyset \quad \textit{Clean}\,(\varphi') \quad \{\,\}\vdash_f v:\pi'',\emptyset}{(\langle\lambda\, x.\, e\rangle_\varrho : (\pi''\xrightarrow{\epsilon.\varphi'}\pi',\varrho),\varphi_0)\xrightarrow{@\,v}(\langle\lambda\, x.\, e\rangle_\varrho\,@\,v:\pi',\varphi'\cup\{\epsilon,\varrho\})}$$

$$\textit{(Obs-Fix)}\quad \frac{S_r=\{\varrho_1\mapsto\rho'_1,\ldots,\varrho_n\mapsto\rho'_n\} \quad \{\,\}\vdash_f \langle\texttt{fix}\, f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho : (\forall\,\varrho_1,\ldots,\varrho_n\,.\tau,\varrho),\emptyset}{(\langle\texttt{fix}\, f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho : (\forall\,\varrho_1,\ldots,\varrho_n\,.\tau,\varrho),\varphi_0)\xrightarrow{[\rho'_1,\ldots,\rho'_n]\ \texttt{at}\ \varrho'} (\langle\texttt{fix}\, f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho\,[\rho'_1,\ldots,\rho'_n]\ \texttt{at}\ \varrho' : (S_r(\tau),\varrho'),\{\varrho,\varrho'\})}$$

$$\textit{(Obs-Dealloc)}\quad \frac{\{\,\}\vdash_f \langle o\rangle_\bullet:\pi,\emptyset}{(\langle o\rangle_\bullet:\pi,\varphi_0)\xrightarrow{\bullet}(\Omega_{\varrho_\star}:(\alpha,\varrho_\star),\{\varrho_\star\})}$$

$$\textit{(Obs-Dang)}\quad \frac{\{\,\}\vdash_f \langle\lambda\, x.\, e\rangle_\varrho : (\pi\xrightarrow{\epsilon.\varphi}\pi',\varrho),\emptyset \quad \textit{Dirty}\,(\varphi)}{(\langle\lambda\, x.\, e\rangle_\varrho : (\pi\xrightarrow{\epsilon.\varphi}\pi',\varrho),\varphi_0)\xrightarrow{\bullet}(\Omega_{\varrho_\star}:(\alpha,\varrho_\star),\{\varrho_\star\})}$$

$$\textit{(Obs-Fixdead)}\quad \frac{\{\,\}\vdash_f \langle\texttt{fix}\, f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho : (\overline{\sigma},\varrho),\emptyset}{(\langle\texttt{fix}\, f.\Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\, e\rangle_\varrho : (\overline{\sigma},\varrho),\varphi_0)\xrightarrow{[\rho'_1,\ldots,\rho'_n]\ \texttt{at}\ \bullet}(\Omega_{\varrho_\star}:(\alpha,\varrho_\star),\{\varrho_\star\})}$$

Figure 5.7: Labeled transitions for the λ_f^{region}-calculus

Except for $\bullet$, observations in the λ_f^{region}-calculus are applicative. The first kind of observation is a value application, the second applies regions. The last observation caters for deallocated code as we shall explain below. Note that the juxtaposition of an expression e with an observation γ, which is not $\bullet$, yields an new expression in the λ_f^{region}-calculus, *i.e.* $e\gamma \in \lambda_f^{region}\text{-}\mathsf{Term}$. However, the observation $[\rho_1,\ldots,\rho_n]\ \texttt{at}\ \bullet$ is not very useful either since it always leads to an expressions that has a dirty effect. Therefore, we define *deadly* observations as follows: *Deadly* (γ) is true iff $\gamma=\bullet$ or $\gamma=[\rho_1,\ldots,\rho_n]\ \texttt{at}\ \bullet$. The predicate *Safe* (γ) holds iff *Deadly* (γ) is false, *i.e.* observations are *safe*.

Define a labeled transition system

$$\longrightarrow \in \mathcal{P}(\mathsf{Term}\times\mathsf{TypeWPlace}\times\mathsf{Effect}\times\mathsf{Observation}\times\mathsf{Term}\times\mathsf{TypeWPlace}\times\mathsf{Effect})$$

inductively by the rules in Figure 5.7.

In a call-by-value language like the λ_f^{region}-calculus, an expression is essentially characterized by its value. Hence, rule *(Obs-Red)* says that the observations of a closed $\lambda_f^{region}\text{-}\mathsf{Term}$ are the observations one can make of its value, provided it exists.

The rules *(Obs-Lam)* and *(Obs-Fix)* observe function application and region application respectively, provided the objects are not deallocated. In *(Obs-Lam)*, we apply a value, not an expression. This decision is not of great importance since expressions are characterized by their values, but it facilitates proofs. Also, we require the arrow effect to be clean as we want to avoid that the resulting effect of an observation is dirty.

Since we want to make an observation of *every* well-typed value, we have the rules *(Obs-Dealloc)* and *(Obs-Dang)*, which cater for values not covered by *(Obs-Lam)* and *(Obs-Fix)*. The transition rule *(Obs-Fixdead)* allows the observation $[\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \bullet$.

Transition rules with deadly observations are designed to catch values that have a dirty effect when applicatively observed. Therefore, these transitions go to the non-terminating program $\Omega_{\varrho_\star}$ which disallows further observations. The region variable $\varrho_\star$ is arbitrary and not dead, but fixed throughout the rest of the chapter to guarantee a deterministic transition system. The effect φ_0 in the rules which observe values is arbitrary since values do not have an effect.

Note that the two observation rules *(Obs-Lam)* and *(Obs-Fix)* apply the context rules (5.9) and (5.11) from Figure 5.3 respectively - except for the value in *(Obs-Lam)*. Continuing along these lines, one could include labeled transitions which correspond to the other evaluation contexts (5.8), (5.10), and (5.12). However, these observations do not allow us to observe *more*. In particular, the region deallocation rule - context rule (5.12) - is not a distinguishing context since region allocations and deallocations do not influence the termination semantics of the region calculus. A notion of bisimilarity based on all evaluation contexts is called experimental equivalence by Gordon [49, 50] and would provide an alternative formulation of Milner's Context Lemma [99].

The transition system has some very useful properties. The following lemma expresses that a well-typed converging expression always allows a transition:

Lemma 5.6.1 *Suppose* $\{\ \} \vdash_f e : \pi, \varphi''$ *where* $\varphi'' \subseteq \varphi$ *and* $e \downarrow$. *Then there exists an observation* γ *such that the transition* $(e : \pi, \varphi) \xrightarrow{\gamma} (e' : \pi', \varphi')$ *exists. Moreover, if* $\mathit{Safe}\,(\gamma)$, *we have for* $e \xrightarrow{*} v$ *that* $e' = v\gamma$.

Proof. By induction on the number of reductions in $e \xrightarrow{*} v$. If e is the value v, then one of the rules *(Obs-Lam)*, *(Obs-Fix)*, *(Obs-Dealloc)* or *(Obs-Dang)* applies, depending on the actual form of π. Whenever $\mathit{Safe}\,(\gamma)$, we also have $e' = v\gamma$ for $e \xrightarrow{*} v$ by the rules *(Obs-Lam)* and *(Obs-Fix)*. Alternatively, we have $e \rightharpoonup e''$ where $e'' \xrightarrow{*} v$. By Proposition 5.2.3, we also have $\{\ \} \vdash_f e'' : \pi, \varphi'''$ with $\varphi''' \subseteq \varphi'' \subseteq \varphi$. By induction, this yields the transition $(e'' : \pi, \varphi) \xrightarrow{\gamma} (e' : \pi', \varphi')$. Hence, rule *(Obs-Red)* yields $(e : \pi, \varphi) \xrightarrow{\gamma} (e' : \pi', \varphi')$. □

The opposite property also holds:

Lemma 5.6.2 *Suppose a transition* $(e : \pi, \varphi) \xrightarrow{\gamma} (e' : \pi', \varphi')$. *Then* $\{\ \} \vdash_f e : \pi, \varphi''$ *and* $\varphi'' \subseteq \varphi$ *with* $e \downarrow$. *Whenever* $\mathit{Safe}\,(\gamma)$, *it also holds that* $e' = v\gamma$ *for* $e \xrightarrow{*} v$.

Proof. First, see by inspection of the rules in Figure 5.7 that always $\{\ \} \vdash_f e : \pi, \varphi''$ with $\varphi'' \subseteq \varphi$.

Whenever the transition is due to one of the rules *(Obs-Lam)*, *(Obs-Fix)*, *(Obs-Dealloc)*, *(Obs-Dang)* or *(Obs-Fixdead)*, we immediately have $e \downarrow$. If *Safe* (γ), it also holds that $e' = v\gamma$ for $e \xrightarrow{*} v$ by the rules *(Obs-Lam)* and *(Obs-Fix)*. In case of *(Obs-Red)*, we have $e \rightharpoonup e''$ and $(e'' : \pi, \varphi) \xrightarrow{\gamma} (e' : \pi', \varphi')$. By induction we have $e'' \downarrow$ and hence also $e \downarrow$. □

A transition preserves typing and guarantees a clean result effect:

Lemma 5.6.3 *Suppose a transition* $(e : \pi, \varphi) \xrightarrow{\gamma} (e' : \pi', \varphi')$. *Then* $\{\,\} \vdash_f e' : \pi', \varphi'$ *with Clean* (φ').

Proof. Simple proof by rule induction on the transition system and inspection of the premises, using type preservation for *(Obs-Red)*. □

The transition system is deterministic:

Lemma 5.6.4 $(e_1 : \pi_1, \varphi_1) \xrightarrow{\gamma} (e_2 : \pi_2, \varphi_2)$ *and* $(e_1 : \pi_1, \varphi_1) \xrightarrow{\gamma} (e_2' : \pi_2, \varphi_2)$ *imply* $e_2 = e_2'$.

Proof. By induction on the derivations and use of Lemma 5.2.7. □

From Lemma 5.6.1, we already have that convergence implies a transition. We can also choose it to coincide with the observation of another expression of the same type.

Lemma 5.6.5 *Suppose* $(e_1 : \pi, \varphi) \xrightarrow{\gamma} (e_1' : \pi', \varphi')$ *and for* $\{\,\} \vdash_f e_2 : \pi, \varphi''$ *with* $\varphi'' \subseteq \varphi$ *it holds that* $e_2 \downarrow$. *Then there exists a transition* $(e_2 : \pi, \varphi) \xrightarrow{\gamma} (e_2' : \pi', \varphi')$. *Moreover, if Safe* (γ), *then* $e_2' = v_2\gamma$ *for* $e_2 \xrightarrow{*} v_2$.

Proof. Straightforward by rule induction on the labeled transition system, where we note that the choice of observation is guided by the form of the type π. □

5.6.2 Simulation and Bisimulation

Recall from Section 2.2.5 that the labeled transition system produces a derivation tree with terms in its nodes and observations on the edges.

We introduce bisimulation for the λ_f^{region}-calculus by relating derivation trees with the same structure:

Definition 5.6.1 *The functions* $\lceil . \rceil, \lfloor . \rfloor \in \mathsf{ClosedTypedRel} \to \mathsf{ClosedTypedRel}$ *are defined*

as follows:

$$\begin{aligned}\lceil\mathcal{R}\rceil = \{(\{\,\}, e_1, e_2, \pi, \varphi) \mid\ & \textit{whenever for all } S_r \textit{ with } \textit{Clean}\,(S_r(\varphi)) \textit{ the transition}\\ & (S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi') \textit{ implies}\\ & \textit{the existence of an expression } e_2' \textit{ such that}\\ & (S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi') \textit{ and}\\ & \{\,\} \rhd e_1' \,\mathcal{R}\, e_2' : \pi', \varphi' \cup \varphi'' \textit{ with } \textit{Clean}\,(\varphi'')\}\end{aligned}$$

$$\begin{aligned}[\mathcal{R}] &= \lceil\mathcal{R}\rceil \cap \lceil\mathcal{R}^{-1}\rceil^{-1}\\ &= \lceil\mathcal{R}\rceil \cap \{(\{\,\}, e_1, e_2, \pi, \varphi) \mid\ \textit{whenever for all } S_r \textit{ with } \textit{Clean}\,(S_r(\varphi)) \textit{ the transition}\\ &\qquad (S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi') \textit{ implies}\\ &\qquad \textit{the existence of an expression } e_1' \textit{ such that}\\ &\qquad (S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi') \textit{ and}\\ &\qquad \{\,\} \rhd e_2' \,\mathcal{R}\, e_1' : \pi', \varphi' \cup \varphi'' \textit{ with } \textit{Clean}\,(\varphi'')\}\end{aligned}$$

The operators $\lceil.\rceil$ and $[.]$ are monotone in the complete lattice of closed typed relations $(\mathsf{ClosedTypedRel}, \subseteq)$.

By the Knaster-Tarski Theorem 2.2.1 on page 14, we have the following notions of similarity and bisimilarity for the λ_f^{region}-calculus:

Definition 5.6.2 *1. A well-formed closed typed relation $\mathcal{R} \in \mathsf{ClosedTypedRel}$, which is $\lceil.\rceil$-dense,* i.e. *$\mathcal{R} \subseteq \lceil\mathcal{R}\rceil$, is a* simulation. Similarity, *$\lesssim$, is defined as the union of all simulations.*

2. A well-formed closed typed relation $\mathcal{R} \in \mathsf{ClosedTypedRel}$, which is $[.]$-dense, i.e. *$\mathcal{R} \subseteq [\mathcal{R}]$, is a* bisimulation. *The union of all bisimulations is the* Bisimilarity *relation, $\approx$.*

By Lemma 5.6.3, (bi)similarity is well-defined. The similarity and bisimilarity relations are non-empty since they are reflexive as we shall see below. Expanding the definition of simulation yields the following more manageable formulation: For a well-formed relation $\mathcal{R}$ we have that $\mathcal{R}$ is a simulation iff $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ implies that if for any S_r such that $\mathit{Clean}\,(S_r(\varphi))$ the transition $(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi')$ exists, then $(S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi')$ must exist such that $\{\,\} \rhd e_1' \,\mathcal{R}\, e_2' : \pi', \varphi' \cup \varphi''$ with $\mathit{Clean}\,(\varphi'')$.

In contrast with the more usual definition of bisimilarity [50], the operators $\lceil.\rceil$ and $[.]$ quantify over a region substitution S_r before considering a transition. This is due to the fact that the λ_f^{region}-calculus is region-polymorphic and therefore allows any free region variable ϱ to be abstracted by a context.

We explain this with an example. Consider the type $\pi = ((\pi_1 \xrightarrow{\epsilon.\emptyset} \pi_2, \varrho) \xrightarrow{\epsilon.\emptyset} (\alpha, \varrho), \varrho)$ such that

$$\pi_1 = ((\alpha, \varrho) \xrightarrow{\epsilon.\emptyset} ((\alpha, \varrho) \xrightarrow{\epsilon.\emptyset} (\alpha, \varrho), \varrho_3), \varrho_1)$$

and

$$\pi_2 = ((\alpha, \varrho) \xrightarrow{\epsilon.\emptyset} ((\alpha, \varrho) \xrightarrow{\epsilon.\emptyset} (\alpha, \varrho), \varrho_3), \varrho_2)$$

In a monomorphic subset of the λ_f^{region}-calculus, the expression

$$\lambda\, f.\, (f \mathbin{@} \langle \lambda\, w.\, \langle \lambda\, z.\, w \rangle_{\varrho_3} \rangle_{\varrho_1})\ \mathtt{at}\ \varrho$$

of type π is contextually equivalent to the expression

$$\lambda\, f.\, (f \mathbin{@} \langle \lambda\, w.\, \langle \lambda\, z.\, z \rangle_{\varrho_3} \rangle_{\varrho_1})\ \mathtt{at}\ \varrho$$

of type π. Intuitively, this is because the only values of type $(\pi_1 \xrightarrow{\epsilon.\emptyset} \pi_2, \varrho)$ have to be functions that forget their arguments. Indeed, it is not possible to move a value from one region (in this case ϱ_1) to another region (in this case ϱ_2) without effect (the latent effect is empty in our example).

However, in the polymorphic λ_f^{region}-calculus, it is possible that a context abstracts over the variables ϱ_1 and ϱ_2. At region application time, these variables may be mapped onto the same region variable, say ϱ'. But this would make it possible to apply the above expressions to the identity function $\langle \lambda\, x.\, x \rangle_{\varrho'}$, which yields different results. Therefore, a notion of bisimilarity which does not include a quantification over region substitutions renders the above expressions bisimilar, even though they are not contextual equivalent in a polymorphic region calculus.

Finally, we also note that in the definitions $\lceil . \rceil$ and $[.]$, we require $\mathit{Clean}\,(S_r(\varphi))$ for the effect of the transition. We need this condition to make sure that whenever a transition if possible at all, it is possible for all S_r too. Without $\mathit{Clean}\,(S_r(\varphi))$ we cannot guarantee that rule *(Obs-Red)* remains applicable for all S_r.

Knaster-Tarski now says that similarity is the greatest simulation and bisimilarity the greatest bisimulation, leading to the following co-inductive proof principles (see also Section 2.2.1), where we assume that $\mathcal{R}$ is well-formed:

Co-induction	$\mathcal{R} \subseteq [\mathcal{R}]$ *implies* $\mathcal{R} \subseteq \approx$
Strong Co-induction	$\mathcal{R} \subseteq [\mathcal{R} \cup \approx] \cup \approx$ *implies* $\mathcal{R} \subseteq \approx$

Additionally, we have that similarity and bisimilarity are fixpoints for $\lceil . \rceil$ and $[.]$ respectively, *i.e.* $\lceil \lesssim \rceil = \lesssim$ and $[\approx] = \approx$.

Sometimes, it is helpful to use the following refined notion of bisimilarity, which is due to Milner [50, 101]. The closed typed relation $\mathcal{R}$ is a bisimulation *up to* $\approx$, provided $\mathcal{R} \subseteq [\approx\, \mathcal{R}\, \approx]$, where we write $\mathcal{R}\mathcal{R}'$ for the composition of two typed relations $\mathcal{R}$ and $\mathcal{R}'$.

5.6.3 Properties of Simulations

A bisimulation up to $\approx$ is also a bisimulation:

Proposition 5.6.6 *Whenever $\mathcal{R}$ is a bisimulation up to $\approx$, then $\approx \mathcal{R} \approx \subseteq [\approx\, \mathcal{R}\, \approx]$*

Proof. See Milner [101] □

Recall from Section 5.3 that we do not in general have $\{\ \} \rhd e_1\, \mathcal{R}\, e_2 : \pi, \varphi$ iff $\{\ \} \rhd e_1\, \mathcal{R}^\circ\, e_2 : \pi, \varphi$. However, with the above notion of bisimilarity, this property does hold for similarity and bisimilarity. To prove this, consider the following substitution property:

Lemma 5.6.7 *Suppose* $\{\ \} \rhd e_1 \lesssim e_2 : \pi, \varphi$ *and* $\mathit{Clean}\,(S_r(\varphi))$. *Then* $\{\ \} \rhd S_r(e_1) \lesssim S_r(e_2) : S_r(\pi), S_r(\varphi)$.

Proof. We prove by co-induction, so take the relation

$$\mathcal{R} = \big\{(\{\ \}, S_r(e_1), S_r(e_2), S_r(\pi), S_r(\varphi)) \mid \{\ \} \rhd e_1 \lesssim e_2 : \pi, \varphi \textit{ with } \mathit{Clean}\,(S_r(\varphi))\big\}$$

By Lemma 5.2.1, we obviously have that $\mathcal{R}$ is well-formed, so it is sufficient to prove that $\mathcal{R} \subseteq \lceil\lesssim\rceil$.

So, we assume $\{\ \} \rhd S_r(e_1)\,\mathcal{R}\,S_r(e_2) : S_r(\pi), S_r(\varphi)$ and consider the transition $((S'_r \circ S_r)(e_1) : (S'_r \circ S_r)(\pi), (S'_r \circ S_r)(\varphi)) \xrightarrow{\gamma'} (e'_1 : \pi', \varphi')$ where $\mathit{Clean}\,(S'_r(\varphi))$. Now, we obviously have $\mathit{Clean}\,((S'_r \circ S_r)(\varphi))$ as well, and since $\{\ \} \rhd e_1 \lesssim e_2 : \pi, \varphi$ holds, we have the transition $((S'_r \circ S_r)(e_2) : (S'_r \circ S_r)(\pi), (S'_r \circ S_r)(\varphi)) \xrightarrow{\gamma'} (e'_2 : \pi', \varphi')$ such that $\{\ \} \rhd e'_1 \lesssim e'_2 : \pi', \varphi'$ by the definition of simulation. □

With this lemma, we also have the following property for (bi)similarity:

Proposition 5.6.8 $\{\ \} \rhd e_1 \lesssim^\circ e_2 : \pi, \varphi$ *iff* $\{\ \} \rhd e_1 \lesssim e_2 : \pi, \varphi$

Proof. From left-to-right, we can take the region substitution I_r in the definition of open extension. From right-to-left, we apply Lemma 5.6.7 for any S_r with $\mathit{Clean}\,(S_r(\varphi))$. We conclude by the definition of open extension. □

We will often use this result implicitly in the rest of this chapter. The substitution lemma also holds for open extensions.

Proposition 5.6.9 *Suppose* $TE \rhd e_1 \lesssim^\circ e_2 : \pi, \varphi$, *then* $S_r(TE) \rhd S_r(e_1) \lesssim^\circ S_r(e_2) : S_r(\pi), S_r(\varphi)$.

Proof. By definition of open extension, we have to prove that for all S'_r and ξ such that $S'_r(S_r(TE)) \succ \xi$ where $\mathit{Clean}\,(S'_r(S_r(\varphi)))$ it holds that $\{\ \} \rhd \xi((S'_r \circ S_r)(e_1)) \lesssim \xi((S'_r \circ S_r)(e_2)) : (S'_r \circ S_r)(\pi), (S'_r \circ S_r)(\varphi)$. The property now holds immediately since we have by assumption that $TE \rhd e_1 \lesssim^\circ e_2 : \pi, \varphi$ and can therefore apply the definition of open extension with region substitution $S'_r \circ S_r$. □

Proposition 5.6.10 *The open extension* $\approx^\circ$ *of bisimilarity is a well-formed equivalence relation:*

1. $\approx^\circ$ *is well-formed*
2. $\lesssim^\circ$ *is a pre-order,* i.e. *reflexive and transitive*
3. $\approx^\circ$ *is an equivalence,* i.e. *reflexive, symmetric and transitive*

Proof.

1. Follows immediately by definition of bisimilarity and open extension.

2. To show reflexivity, it is easy to see by definition of simulation that the relation

$$\big\{(\{\,\}, e, e, \pi, \varphi) \mid \{\,\} \vdash_f e : \pi, \varphi' \textit{ with } \varphi' \subseteq \varphi \textit{ and } \mathit{Clean}\,(\varphi)\big\}$$

is well-formed and $\lceil . \rceil$-dense, using Lemma 5.6.3 and Lemma 5.2.1. Hence, by co-induction, $\lesssim$ is reflexive. The property holds now trivially for $\lesssim^\circ$ as well.

For transitivity, assume

$$\mathcal{R} = \{(\{\,\}, e_1, e_3, \pi, \varphi) \mid \quad \textit{there exists } e_2 \textit{ such that } \{\,\} \rhd e_1 \lesssim e_2 : \pi, \varphi \textit{ and } \{\,\} \rhd e_2 \lesssim e_3 : \pi, \varphi\}$$

The relation $\lesssim$ is transitive iff we have that $\mathcal{R} \subseteq \lesssim$. By co-induction, it is sufficient to show that $\mathcal{R} \subseteq \lceil \mathcal{R} \rceil$. So, suppose $\{\,\} \rhd e_1 \,\mathcal{R}\, e_3 : \pi, \varphi$. Then $\{\,\} \rhd e_1 \lesssim e_2 : \pi, \varphi$ and $\{\,\} \rhd e_2 \lesssim e_3 : \pi, \varphi$ for some e_2. By simulation, this means that $(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi')$ for $\mathit{Clean}\,(S_r(\varphi))$ implies the transition $(S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi')$ such that $\{\,\} \rhd e_1' \lesssim e_2' : \pi', \varphi' \cup \varphi''$ with $\mathit{Clean}\,(\varphi'')$. Therefore, the transition $(S_r(e_3) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_3' : \pi', \varphi')$ holds and $\{\,\} \rhd e_2' \lesssim e_3' : \pi', \varphi' \cup \varphi''$. Hence, $\{\,\} \rhd e_1' \,\mathcal{R}\, e_3' : \pi', \varphi' \cup \varphi''$, which proves transitivity of $\lesssim$. Since type environments remain invariant under transitivity, the property holds trivially for $\lesssim^\circ$.

3. Reflexivity and transitivity follow by Item 2. Symmetry is obtained by a similar co-induction argument using the definition of bisimilarity.

□

Determinism of the transition system makes bisimilarity equivalent to mutual similarity:

Proposition 5.6.11 *$TE \rhd e_1 \approx^\circ e_2 : \pi, \varphi$ iff $TE \rhd e_1 \lesssim^\circ e_2 : \pi, \varphi$ and $TE \rhd e_2 \lesssim^\circ e_1 : \pi, \varphi$ ($\approx^\circ = \lesssim^\circ \cap \lesssim^{\circ -1}$).*

Proof. It is immediate that when the property holds for similarity (respectively bisimilarity), it also holds for their open extensions. Hence, we prove the proposition for similarity and bisimilarity.

We observe that $\approx \,=\, [\approx] \,=\, \lceil \approx \rceil \cap \lceil \approx^{-1} \rceil^{-1}$. Hence, we have $\approx \subseteq \lceil \approx \rceil$ and by co-induction that $\approx \subseteq \lesssim$. So, by Proposition 5.6.10.3, also $\approx \subseteq \lesssim^{-1}$, implying $\approx \subseteq \lesssim \cap \lesssim^{-1}$.

For the other direction, take the well-formed typed relation

$$\mathcal{R} = \lesssim \cap \lesssim^{-1} = \big\{(\{\,\}, e_1, e_2, \pi, \varphi) \mid \{\,\} \rhd e_1 \lesssim e_2 : \pi, \varphi \textit{ and } \{\,\} \rhd e_2 \lesssim e_1 : \pi, \varphi\big\}$$

By co-induction, it suffices to show that $\mathcal{R} \subseteq [\mathcal{R}]$. Since $\mathcal{R}$ is symmetric and $[\mathcal{R}] = \lceil \mathcal{R} \rceil \cap \lceil \mathcal{R}^{-1} \rceil^{-1}$, we have to prove $\mathcal{R} \subseteq \lceil \mathcal{R} \rceil$. So, take $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ and suppose the transition $(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi')$ where $\mathit{Clean}\,(S_r(\varphi))$. By definition of $\mathcal{R}$, there exists an e_2' such that $(S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi')$ and

$\{\,\} \rhd e_1' \lesssim e_2' : \pi', \varphi' \cup \varphi''$ with *Clean* (φ''). Also, we have that $\{\,\} \rhd e_2 \lesssim e_1 : \pi, \varphi$, so there exists an e_1'' such that $(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1'' : \pi', \varphi')$ and $\{\,\} \rhd e_2' \lesssim e_1'' : \pi', \varphi' \cup \varphi''$. By Lemma 5.6.4, it must be that $e_1' = e_1''$, so $\{\,\} \rhd e_2' \lesssim e_1' : \pi', \varphi' \cup \varphi''$, which implies $\{\,\} \rhd e_1' \,\mathcal{R}\, e_2' : \pi, \varphi' \cup \varphi''$, showing that $\mathcal{R}$ is a simulation. □

Mutual similarity allows the following practical formulation of bisimilarity:

$$\{\,\} \rhd e_1 \approx e_2 : \pi, \varphi$$
iff
for all S_r *such that Clean* $(S_r(\varphi))$ *it holds that*
$$(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi') \Leftrightarrow (S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi')$$
where $\{\,\} \rhd e_1' \approx e_2' : \pi', \varphi' \cup \varphi''$ *with Clean* (φ'')

Henceforth, we will assume that application of Proposition 5.6.11 is implicit.

If a converging expression is similar to an expression, then that expression must also converge:

Lemma 5.6.12 *Suppose* $TE \rhd e \lesssim^\circ e' : \pi, \varphi$. *If* $e \downarrow$ *then* $e' \downarrow$.

Proof. By open extension, we take ξ and S_r such that $S_r(TE) \succ \xi$ with *Clean* $(S_r(\varphi))$ and have $\{\,\} \rhd \xi(S_r(e)) \lesssim \xi(S_r(e')) : S_r(\pi), S_r(\varphi)$. By Lemma 5.2.8 and Lemma 5.2.9 we now have that $\xi(S_r(e)) \downarrow$, so using Lemma 5.3.1 and Lemma 5.6.1, we have a transition from $\xi(S_r(e))$. By the definition of similarity therefore also from $\xi(S_r(e'))$. Lemma 5.3.1 and Lemma 5.6.2 prove the claim. □

A well-typed clean diverging expression is immediately similar to any other well-typed clean expression.

Lemma 5.6.13 *Suppose* $\{\,\} \vdash_f e_1 : \pi, \varphi_1$ *and* $\{\,\} \vdash_f e_2 : \pi, \varphi_2$ *where* $\varphi_1 \cup \varphi_2 \subseteq \varphi$ *and Clean* (φ). *If* $e_1 \uparrow$ *then* $\{\,\} \rhd e_1 \lesssim e_2 : \pi, \varphi$.

Proof. Simple consequence of the definition of similarity and Lemma 5.6.2. □

The left and right-hand side of a λ_f^{region}-reduction are bisimilar:

Lemma 5.6.14 *Suppose* $TE \vdash_f e : \pi, \varphi'$ *and* $e \rightharpoonup e'$ *where* $\varphi' \subseteq \varphi$ *and Clean* (φ). *Then* $TE \rhd e \approx^\circ e' : \pi, \varphi$.

Proof. We prove the lemma for $\approx$ since it trivially holds for $\approx^\circ$ by Lemma 5.2.9. So define

$$\mathcal{R} = \big\{(\{\,\}, e, e', \pi, \varphi) \mid \{\,\} \vdash_f e : \pi, \varphi' \text{ and } e \rightharpoonup e' \text{ where } \varphi' \subseteq \varphi \text{ with } \mathit{Clean}\,(\varphi)\big\}$$

By Proposition 5.2.3, we also have $\{\,\} \vdash_f e' : \pi, \varphi''$ with $\varphi'' \subseteq \varphi'$. Hence $\mathcal{R}$ is well-formed and we have to show that $\mathcal{R} \subseteq \approx$, where $\approx = [\approx] = \lceil \approx \rceil \cap \lceil \approx^{-1} \rceil^{-1}$. So, suppose $\{\,\} \rhd e \,\mathcal{R}\, e' : \pi, \varphi$.

First, assume that $(S_r(e) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_0 : \pi', \varphi')$ such that *Clean* $(S_r(\varphi))$. Since $S_r(e)$ cannot be a value, the transition must be due to rule *(Obs-Red)* and therefore by Lemma 5.2.7 and Lemma 5.2.9 we have $(S_r(e') : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_0 : \pi', \varphi')$. By Lemma 5.6.3, we have that $\{\,\} \vdash_f e_0 : \pi', \varphi'$ with *Clean* (φ') and hence by reflexivity of $\approx$, also that $\{\,\} \rhd e_0 \approx e_0 : \pi', \varphi' \cup \varphi''$ with *Clean* (φ''). Hence, $\mathcal{R} \subseteq \lceil \approx \rceil$.

The other way around, assume transition $(S_r(e') : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e'_0 : \pi', \varphi')$ where *Clean* $(S_r(\varphi))$. Then, immediately by rule *(Obs-Red)* and Lemma 5.2.9, we have that $(S_r(e) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e'_0 : \pi', \varphi')$. Because of Lemma 5.6.3, we also have $\{\,\} \vdash_f e'_0 : \pi', \varphi'$ with *Clean* (φ') and therefore by reflexivity that $\{\,\} \rhd e'_0 \approx e'_0 : \pi', \varphi' \cup \varphi''$ with *Clean* (φ''), implying $\mathcal{R} \subseteq \lceil \approx^{-1} \rceil^{-1}$.

□

5.7 Bisimilarity is Compatible

In this section, we prove that bisimilarity obeys a congruence property, more specifically, we show that the bisimilarity relation is compatible. We use a technique due to Howe [76, 77], which has been re-casted to typed functional languages by Gordon [48–50]. The key idea is to define the notion of *compatible extension*, a compatible relation which we prove to coincide with the open extension of similarity.

The compatible extension $\lesssim^*$ of $\lesssim$ is inductively defined as the least well-formed typed relation closed under the following rule:

$$\textit{(Def-Comp-Ext)} \quad \frac{TE \rhd e_1 \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi \quad TE \rhd e_3 \lesssim^\circ e_2 : \pi, \varphi}{TE \rhd e_1 \lesssim^* e_2 : \pi, \varphi}$$

We have the following auxiliary result:

Lemma 5.7.1 *If* $TE \rhd e_1 \lesssim^* e_2 : \pi, \varphi$ *and* $TE \rhd e_2 \lesssim^\circ e_3 : \pi, \varphi$*, then* $TE \rhd e_1 \lesssim^* e_3 : \pi, \varphi$

Proof. By rule *(Def-Comp-Ext)* there exists an e'_2 such that $TE \rhd e_1 \mathrel{\widehat{\lesssim^*}} e'_2 : \pi, \varphi$ and $TE \rhd e'_2 \lesssim^\circ e_2 : \pi, \varphi$. By transitivity of $\lesssim^\circ$, we have that $TE \rhd e'_2 \lesssim^\circ e_3 : \pi, \varphi$. Hence, by *(Def-Comp-Ext)*, it follows that $TE \rhd e_1 \lesssim^* e_3 : \pi, \varphi$. □

The compatible extension $\lesssim^*$ has the following basic properties:

Proposition 5.7.2 *1.* $\lesssim^*$ *is compatible*

2. $\lesssim^*$ *is reflexive*

3. $\lesssim^\circ \subseteq \lesssim^*$

Proof.

1. Take $TE \rhd e_1 \widehat{\lesssim^*} e_2 : \pi, \varphi$. Since $\lesssim^*$ is well-formed by definition, we have by Lemma 5.4.1.2 that $\widehat{\lesssim^*}$ is well-formed and hence, $TE \vdash_f e_2 : \pi, \varphi'$ with $\varphi' \subseteq \varphi$ and $\mathit{Clean}\,(\varphi)$. Since $\lesssim^\circ$ is reflexive, this gives us that $TE \rhd e_2 \lesssim^\circ e_2 : \pi, \varphi$. Therefore, by *(Def-Comp-Ext)*, we conclude that $TE \rhd e_1 \lesssim^* e_2 : \pi, \varphi$.

2. Immediate by Lemma 5.4.1.1 and the previous item.

3. Assume $TE \rhd e_1 \lesssim^\circ e_2 : \pi, \varphi$. Since $\lesssim^\circ$ is well-formed, we have $TE \vdash_f e_2 : \pi, \varphi'$ with $\varphi' \subseteq \varphi$ and $\mathit{Clean}\,(\varphi)$. So by Item 2 this implies that $TE \rhd e_2 \lesssim^* e_2 : \pi, \varphi$. Using Lemma 5.7.1 this yields $TE \rhd e_1 \lesssim^* e_2 : \pi, \varphi$, proving the claim.

□

The compatible extension $\lesssim^*$ obeys a variable substitution lemma:

Lemma 5.7.3 *If* $TE \rhd v_1 \lesssim^* v_2 : (\overline{\sigma}, \rho), \emptyset$ *and* $TE + \{y \mapsto (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\overline{\sigma}, \rho)\} \rhd e_1 \lesssim^* e_2 : \pi, \varphi$ *with* $(\{\vec{\alpha}\} \cup \{\vec{\epsilon}\}) \cap \mathit{fv}\,(TE) = \emptyset$, *then* $TE \rhd e_1[y \mapsto v_1] \lesssim^* e_2[y \mapsto v_2] : \pi, \varphi$.

Proof. So we have $TE + \{y \mapsto (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\overline{\sigma}, \rho)\} \rhd e_1 \widehat{\lesssim^*} e_3 : \pi, \varphi$ and $TE + \{y \mapsto (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\overline{\sigma}, \rho)\} \rhd e_3 \lesssim^\circ e_2 : \pi, \varphi$. By Lemma 5.3.2 we already have $TE \rhd e_3[y \mapsto v_2] \lesssim^\circ e_2[y \mapsto v_2] : \pi, \varphi$.

Using an inductive case analysis on the compatible refinement we prove that $TE \rhd e_1[y \mapsto v_1] \lesssim^* e_3[y \mapsto v_2] : \pi, \varphi$ and conclude by applying Lemma 5.7.1. The only interesting case is *(Comp-Var)* where $x = y$ and we have to prove that $TE \rhd v_1 \lesssim^* v_2 : \pi, \varphi$. This, however, is given by assumption. All other cases are shown by application of the induction hypothesis and use of Proposition 5.7.2.1. □

Additionally, the relation $\lesssim^*$ obeys a region substitution lemma (similar to Proposition 5.6.9).

Lemma 5.7.4 *Suppose* $TE \rhd e_1 \lesssim^* e_2 : \pi, \varphi$. *Then* $S_r(TE) \rhd S_r(e_1) \lesssim^* S_r(e_2) : S_r(\pi), S_r(\varphi)$.

Proof. So, we have by *(Def-Comp-Ext)* that

$$TE \rhd e_1 \widehat{\lesssim^*} e_3 : \pi, \varphi \tag{5.18}$$

and $TE \rhd e_3 \lesssim^\circ e_2 : \pi, \varphi$. Applying Proposition 5.6.9 on the latter yields $S_r(TE) \rhd S_r(e_3) \lesssim^\circ S_r(e_2) : S_r(\pi), S_r(\varphi)$. It remains to show that $S_r(TE) \rhd S_r(e_1) \lesssim^* S_r(e_3) : S_r(\pi), S_r(\varphi)$, since we can then use Lemma 5.7.1 to obtain the desired result. The claim is proven by a straightforward induction on the compatible refinement, using Proposition 5.7.2.1. □

If a converging expression is $\lesssim^*$-related to another expression, then that expression converges, too.

Lemma 5.7.5 *Suppose* $TE \rhd e \lesssim^* e' : \pi, \varphi$. *If* $e \downarrow$ *then* $e' \downarrow$.

Proof. By *(Def-Comp-Ext)* we have $TE \rhd e \mathrel{\widehat{\lesssim^*}} e'' : \pi, \varphi$ and $TE \rhd e'' \lesssim^\circ e' : \pi, \varphi$. For the cases *(Comp-Lam)*, *(Comp-Lamv)*, *(Comp-Fix)* and *(Comp-Fixv)*, it is immediate that $e'' \downarrow$. For the other cases we apply the induction hypothesis to obtain $e'' \downarrow$ too. The lemma is now proven by Lemma 5.6.12. □

To show that the open extension of similarity is compatible, we need an important auxiliary lemma:

Lemma 5.7.6 *Suppose* $\{\,\} \rhd e_1 \lesssim^* e_2 : \pi, \varphi$ *and* $e_1 \rightharpoonup e_1'$. *Then* $\{\,\} \rhd e_1' \lesssim^* e_2 : \pi, \varphi$.

Proof. By rule induction on $e_1 \rightharpoonup e_1'$, where we always use *(Def-Comp-Ext)* on $\{\,\} \rhd e_1 \lesssim^* e_2 : \pi, \varphi$.

Case $\lambda x.\, e\ \mathtt{at}\ \varrho \rightharpoonup \langle \lambda x.\, e \rangle_\varrho$**.** So we have that $\{\,\} \rhd \lambda x.\, e\ \mathtt{at}\ \varrho \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$ and

$$\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi \tag{5.19}$$

By rule *(Comp-Lam)* we have that $\pi = \pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2$ and $e_3 = \lambda x.\, e'\ \mathtt{at}\ \varrho$. Additionally we have $\varrho \in \varphi$ and $\{x \mapsto \pi_1\} \rhd e \lesssim^* e' : \pi_2, \varphi'$. By Lemma 5.6.14 the similarity $\{\,\} \rhd \langle \lambda x.\, e' \rangle_\varrho \lesssim \lambda x.\, e'\ \mathtt{at}\ \varrho : \pi, \varphi$ holds, and hence, by (5.19) and transitivity of $\lesssim$ also that

$$\{\,\} \rhd \langle \lambda x.\, e' \rangle_\varrho \lesssim e_2 : \pi, \varphi \tag{5.20}$$

Since the premises of *(Comp-Lam)* and *(Comp-Lamv)* are identical, we also have $\{\,\} \rhd \langle \lambda x.\, e \rangle_\varrho \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$, which combined with (5.20) and *(Def-Comp-Ext)* makes $\{\,\} \rhd \langle \lambda x.\, e \rangle_\varrho \lesssim^* e_2 : \pi, \varphi$.

Case $\mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e\ \mathtt{at}\ \varrho \rightharpoonup \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e \rangle_\varrho$**.**
Analogous to the previous case.

Case $\mathtt{new}\, \varrho.\, v \rightharpoonup v[\varrho \mapsto \bullet]$**.** Hence, $\{\,\} \rhd \mathtt{new}\, \varrho.\, v \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$ and $\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi$. By rule *(Comp-Reglam)*, it must be that $e_3 = \mathtt{new}\, \varrho.\, e_3''$ where $\{\,\} \rhd v \lesssim^* e_3'' : \pi, \varphi$ and $\varrho \notin \mathit{frv}\,(\pi)$. Using Lemma 5.7.5, we have that $e_3'' \overset{*}{\rightharpoonup} v_3$ and by reduction rule (5.3) also that $\mathtt{new}\, \varrho.\, e_3'' \overset{*}{\rightharpoonup} v_3[\varrho \mapsto \bullet]$. Hence, by Lemma 5.6.14, we have $\{\,\} \rhd e_3'' \lesssim v_3 : \pi, \varphi$ and $\{\,\} \rhd v_3[\varrho \mapsto \bullet] \lesssim e_3 : \pi, \varphi$. Transitivity of similarity on the one hand and Lemma 5.7.1 on the other make

$$\{\,\} \rhd v_3[\varrho \mapsto \bullet] \lesssim e_2 : \pi, \varphi \tag{5.21}$$

and $\{\,\} \rhd v \lesssim^* v_3 : \pi, \emptyset$. Applying Lemma 5.7.4 on the latter gives $\{\,\} \rhd v[\varrho \mapsto \bullet] \lesssim^* v_3[\varrho \mapsto \bullet] : \pi, \emptyset$. By well-formedness we also have $\{\,\} \rhd v[\varrho \mapsto \bullet] \lesssim^* v_3[\varrho \mapsto \bullet] : \pi, \varphi$ and hence, we can combine with (5.21) using Lemma 5.7.1 to obtain $\{\,\} \rhd v[\varrho \mapsto \bullet] \lesssim^* e_2 : \pi, \emptyset$.

Case $\langle \lambda\, x.\, e \rangle_\varrho \mathbin{@} v \rightharpoonup e[x \mapsto v]$**.** We are given that $\{\,\} \rhd \langle \lambda\, x.\, e \rangle_\varrho \mathbin{@} v \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$ and $\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi$. By *(Comp-App)* we must have that $e_3 = \langle \lambda\, x.\, e'' \rangle_\varrho \mathbin{@} e'$ where $\{\,\} \rhd v \lesssim^* e' : \pi', \varphi_1$ and

$$\{x \mapsto \pi'\} \rhd e \lesssim^* e'' : \pi, \varphi_2 \tag{5.22}$$

Using Lemma 5.7.5 it also holds that $e' \xrightarrow{*} v'$ and by Lemma 5.6.14 we have $\{\,\} \rhd e' \lesssim v' : \pi', \varphi_1$. From Lemma 5.7.1 this implies $\{\,\} \rhd v \lesssim^* v' : \pi', \varphi_1$. The latter, (5.22) and Lemma 5.7.3 imply

$$\{\,\} \rhd e[x \mapsto v] \lesssim^* e''[x \mapsto v'] : \pi, \varphi_2 \tag{5.23}$$

Since we have that $\langle \lambda\, x.\, e'' \rangle_\varrho \mathbin{@} e' \xrightarrow{*} e''[x \mapsto v']$, it also holds that $\{\,\} \rhd e''[x \mapsto v'] \lesssim e_3 : \pi, \varphi$ and hence, by transitivity of $\lesssim$ also $\{\,\} \rhd e''[x \mapsto v'] \lesssim e_2 : \pi, \varphi$. This, in combination with (5.23) and Lemma 5.7.1 proves the case.

Case $\mathtt{let}\ x = v\ \mathtt{in}\ e \rightharpoonup e[x \mapsto v]$**.** Analogous to the previous case.

Case
$$\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' \rightharpoonup \langle \lambda\, x.\, e[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n][f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho] \rangle_{\varrho'}\ .$$

Now, we have that $\{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$ and $\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi$ hold. Hence, by *(Comp-Regapp)*, the only possible form for e_3 is $e'_3\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho'$ where $\pi = (S_r(\tau), \varrho')$; the substitution $S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}$ and $\overline{\sigma} = \forall\, \varrho_1, \ldots, \varrho_n\, .\tau$. Additionally, we also have $\{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho \lesssim^* e'_3 : (\overline{\sigma}, \varrho), \varphi'$ with $\varphi' \cup \{\varrho, \varrho'\} \subseteq \varphi$.

Using Lemma 5.7.5, we have that $e'_3 \xrightarrow{*} v_3$ and by the canonical forms Lemma 5.2.4 also that v_3 has the form $\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\varrho$. Using Lemma 5.6.14 and Lemma 5.7.1 we therefore have

$$\{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho \lesssim^* \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\varrho : (\overline{\sigma}, \varrho), \varphi' \tag{5.24}$$

Since $e_3 \xrightarrow{*} \langle \lambda\, x.\, S_r(e')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\varrho] \rangle_{\varrho'}$ too, we also obtain with Lemma 5.6.14 that

$$\{\,\} \rhd \langle \lambda\, x.\, S_r(e')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\varrho] \rangle_{\varrho'} \lesssim e_3 : \pi, \varphi$$

Since $\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi$, by transitivity of $\lesssim$ this yields

$$\{\,\} \rhd \langle \lambda\, x.\, S_r(e')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\varrho] \rangle_{\varrho'} \lesssim e_2 : \pi, \varphi \tag{5.25}$$

By *(Def-Comp-Ext)*, we have from (5.24) that

$$\{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e \rangle_\varrho \mathrel{\widehat{\lesssim^*}} e_4 : (\overline{\sigma}, \varrho), \varphi' \tag{5.26}$$

and

$$\{\,\} \rhd e_4 \lesssim \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e' \rangle_\varrho : (\overline{\sigma}, \varrho), \varphi' \tag{5.27}$$

From *(Comp-Fixv)*, the congruence (5.26) implies that e_4 must be a recursive function object of the form $\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e'' \rangle_\varrho$ with $\varrho \notin \{\varrho_1, \ldots, \varrho_n\}$ and also $\{f \mapsto (\forall\, \vec{\epsilon}.\overline{\sigma}, \varrho)\} \rhd \langle \lambda\, x.\ e \rangle_\varrho \lesssim^* \langle \lambda\, x.\ e'' \rangle_\varrho : (\tau, \varrho), \emptyset$. The latter on its turn yields with *(Def-Comp-Ext)* and *(Comp-Lamv)* that $\{f \mapsto (\forall\, \vec{\epsilon}.\overline{\sigma}, \varrho)\} \rhd \langle \lambda\, x.\ e \rangle_\varrho \mathrel{\widehat{\lesssim^*}} \langle \lambda\, x.\ e''' \rangle_\varrho : (\tau, \varrho), \emptyset$ and $\{f \mapsto (\forall\, \vec{\epsilon}.\overline{\sigma}, \varrho)\} \rhd \langle \lambda\, x.\ e''' \rangle_\varrho \lesssim^\circ \langle \lambda\, x.\ e'' \rangle_\varrho : (\tau, \varrho), \emptyset$. The compatibility rule *(Comp-Lamv)*, the definition of open extension, the transition rule *(Obs-Lam)* and the type rule *(F-Lamv)* make that $\{f \mapsto (\forall\, \vec{\epsilon}.\overline{\sigma}, \varrho)\} \rhd \langle \lambda\, x.\ e \rangle_{\varrho'} \mathrel{\widehat{\lesssim^*}} \langle \lambda\, x.\ e''' \rangle_{\varrho'} : (\tau, \varrho'), \emptyset$ and $\{f \mapsto (\forall\, \vec{\epsilon}.\overline{\sigma}, \varrho)\} \rhd \langle \lambda\, x.\ e''' \rangle_{\varrho'} \lesssim^\circ \langle \lambda\, x.\ e'' \rangle_{\varrho'} : (\tau, \varrho'), \emptyset$ also holds. Recombining with *(Def-Comp-Ext)* yields $\{f \mapsto (\forall\, \vec{\epsilon}.\overline{\sigma}, \varrho)\} \rhd \langle \lambda\, x.\ e \rangle_{\varrho'} \lesssim^* \langle \lambda\, x.\ e'' \rangle_{\varrho'} : (\tau, \varrho'), \emptyset$. Using Lemma 5.7.4 with S_r and Lemma 5.7.3 with (5.26) on this assertion, we obtain

$$\begin{aligned} \{\,\} \rhd \langle \lambda\, x.\ S_r(e)[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e \rangle_\varrho] \rangle_{\varrho'} \lesssim^* \\ \langle \lambda\, x.\ S_r(e'')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e'' \rangle_\varrho] \rangle_{\varrho'} : \pi, \emptyset \end{aligned} \tag{5.28}$$

But we also have by (5.27) and the definition of similarity (using *(Obs-Fix)*) that

$$\begin{aligned} \{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e'' \rangle_\varrho\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \varrho' \lesssim \\ \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e' \rangle_\varrho\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \varrho' : (\tau, \varrho'), \emptyset \end{aligned}$$

Using Lemma 5.6.14 and transitivity of $\lesssim$, we obtain

$$\begin{aligned} \{\,\} \rhd \langle \lambda\, x.\ S_r(e'')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e'' \rangle_\varrho] \rangle_{\varrho'} \lesssim \\ \langle \lambda\, x.\ S_r(e')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e' \rangle_\varrho] \rangle_{\varrho'} : (\tau, \varrho'), \emptyset \end{aligned}$$

In combination with (5.28) via Lemma 5.7.1 this yields

$$\begin{aligned} \{\,\} \rhd \langle \lambda\, x.\ S_r(e)[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e \rangle_\varrho] \rangle_{\varrho'} \lesssim^* \\ \langle \lambda\, x.\ S_r(e')[f \mapsto \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e' \rangle_\varrho] \rangle_{\varrho'} : \pi, \varphi \end{aligned}$$

By combining with (5.25) using Lemma 5.7.1, we prove the case.

Case $e \mathbin{@} e'' \rightharpoonup e' \mathbin{@} e''$ **where** $e \rightharpoonup e'$**.** We have $\{\,\} \rhd e \mathbin{@} e'' \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$ and

$$\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi \tag{5.29}$$

By *(Comp-App)*, e_3 must be an application of the form $e_0 \mathbin{@} e_0''$ where $\{\,\} \rhd e \lesssim^* e_0 : (\pi' \xrightarrow{\epsilon.\varphi} \pi, \rho), \varphi_1$ and $\{\,\} \rhd e'' \lesssim^* e_0'' : \pi', \varphi_2$. Because of the induction hypothesis, we also have $\{\,\} \rhd e' \lesssim^* e_0 : (\pi' \xrightarrow{\epsilon.\varphi} \pi, \rho), \varphi_1$. Again, by rule *(Comp-App)*, we have that $\{\,\} \rhd e' \mathbin{@} e'' \mathrel{\widehat{\lesssim^*}} e_0 \mathbin{@} e_0'' : \pi, \varphi$ and combining this with (5.29) and *(Def-Comp-Ext)* the case follows.

The remaining context cases are all analogous to the case for $e \mathbin{@} e'' \rightharpoonup e' \mathbin{@} e''$. □

With this lemma, the following inclusion result can be proven:

Lemma 5.7.7 *If* $\{\,\} \rhd e_1 \lesssim^* e_2 : \pi, \varphi$*, then* $\{\,\} \rhd e_1 \lesssim e_2 : \pi, \varphi$.

Proof. We prove by co-induction, hence, define the relation

$$\mathcal{R} = \left\{(\{\,\}, e_1, e_2, \pi, \varphi) \mid \{\,\} \rhd e_1 \lesssim^* e_2 : \pi, \varphi\right\}$$

Obviously, $\mathcal{R}$ is well-formed since $\lesssim^*$ is, so it is sufficient to prove that $\mathcal{R} \subseteq \lceil \mathcal{R} \cup \lesssim \rceil$. Take $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ and a transition $(S'_r(e_1) : S'_r(\pi), S'_r(\varphi)) \xrightarrow{\gamma} (e'_1 : \pi', \varphi')$ such that *Clean* $(S'_r(\varphi))$.

We proceed by an inductive case analysis on the labeled transition system:

Case *(Obs-Red)*. So we have that $\{\,\} \rhd e_1 \lesssim^* e_2 : \pi, \varphi$ and since $e_1 \rightharpoonup e''_1$ holds we can use Lemma 5.7.6 to obtain $\{\,\} \rhd e''_1 \lesssim^* e_2 : \pi, \varphi$.

Hence, $\{\,\} \rhd e''_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ and so, by induction and Lemma 5.2.9, there is a labeled transition $(S'_r(e_2) : S'_r(\pi), S'_r(\varphi)) \xrightarrow{\gamma} (e'_2 : \pi', \varphi')$ such that $(\{\,\}, e'_1, e'_2, \pi', \varphi' \cup \varphi'') \in \mathcal{R} \cup \lesssim$ with *Clean* (φ''), proving the case.

Case *(Obs-Lam)*. So we have that $e_1 = \langle \lambda\, x.\, e''_1 \rangle_\varrho$, the type $\pi = (\pi_1 \xrightarrow{\epsilon.\varphi'} \pi_2, \varrho)$ with *Clean* (φ'), the observation $\gamma = @\, v$, and $e'_1 = S'_r(\langle \lambda\, x.\, e''_1 \rangle_\varrho) \,@\, v$.

But we also have $\{\,\} \rhd \langle \lambda\, x.\, e''_1 \rangle_\varrho \lesssim^* e_2 : \pi, \varphi$ and therefore, by *(Def-Comp-Ext)*, this implies $\{\,\} \rhd \langle \lambda\, x.\, e''_1 \rangle_\varrho \mathrel{\widehat{\lesssim^*}} \langle \lambda\, x.\, e''_3 \rangle_\varrho : \pi, \varphi$ and $\{\,\} \rhd \langle \lambda\, x.\, e''_3 \rangle_\varrho \lesssim e_2 : \pi, \varphi$. Hence, by Proposition 5.7.2.1, we also have $\{\,\} \rhd \langle \lambda\, x.\, e''_1 \rangle_\varrho \lesssim^* \langle \lambda\, x.\, e''_3 \rangle_\varrho : \pi, \varphi$ and therefore by Lemma 5.7.4 that $\{\,\} \rhd S'_r(\langle \lambda\, x.\, e''_1 \rangle_\varrho) \lesssim^* S'_r(\langle \lambda\, x.\, e''_3 \rangle_\varrho) : S'_r(\pi), S'_r(\varphi)$ too. Also, by Proposition 5.7.2.2, we have $\{\,\} \rhd v \lesssim^* v : S'_r(\pi_1), \emptyset$. Combining this with *(Comp-App)* we obtain that

$$\{\,\} \rhd S'_r(\langle \lambda\, x.\, e''_1 \rangle_\varrho) \,@\, v \mathrel{\widehat{\lesssim^*}} S'_r(\langle \lambda\, x.\, e''_3 \rangle_\varrho) \,@\, v : S'_r(\pi'_2), S'_r(\varphi'') \tag{5.30}$$

where $S'_r(\varphi') \cup \left\{\epsilon, S'_r(\varrho)\right\} \subseteq S'_r(\varphi'')$ and *Clean* $(S'_r(\varphi''))$. From Lemma 5.6.5, if follows that transition $(S'_r(\langle \lambda\, x.\, e''_3 \rangle_\varrho) : S'_r(\pi), S'_r(\varphi)) \xrightarrow{@\, v} (\langle \lambda\, x.\, e''_3 \rangle_\varrho \,@\, v : \pi'_2, \varphi'')$ exists. Since $\{\,\} \rhd \langle \lambda\, x.\, e''_3 \rangle_\varrho \lesssim e_2 : \pi, \varphi$ holds and by the definition of $\lesssim$, we also have $\{\,\} \rhd S'_r(\langle \lambda\, x.\, e''_3 \rangle_\varrho) \,@\, v \lesssim S'_r(e_2) \,@\, v : S'_r(\pi'_2), S'_r(\varphi'')$. This, together with (5.30) and *(Def-Comp-Ext)* proves the case, where we note that $\lesssim^*$ is already well-formed.

Case *(Obs-Fix)*. We have that $e_1 = \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_1 \rangle_\varrho$ and the type with place π is $(\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \varrho)$. So, the observation γ is $[\rho'_1, \ldots, \rho'_n]$ $\mathtt{at}\ \varrho'$ and the expression e'_1 is $S'_r(e_1)$ $[\rho'_1, \ldots, \rho'_n]$ $\mathtt{at}\ \varrho'$. The type π' is $(S_r(S'_r(\tau)), \varrho')$ where the region substitution $S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}$.

However, since we have $\{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_1 \rangle_\varrho \lesssim^* e_2 : \pi, \varphi$, by *(Def-Comp-Ext)*, we also have $\{\,\} \rhd \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_1 \rangle_\varrho \mathrel{\widehat{\lesssim^*}} e_3 : \pi, \varphi$ and $\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi$ where $e_3 = \langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_3 \rangle_\varrho$. Due to Proposition 5.7.2.1, Lemma 5.7.4 and *(Comp-Regapp)* we have that

$$\begin{aligned} \{\,\} \rhd\ & S'_r(\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_1 \rangle_\varrho)\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' \lesssim^* \\ & S'_r(\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_3 \rangle_\varrho)\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' : (\pi_0, \varrho'), \varphi'' \end{aligned} \tag{5.31}$$

where $\pi_0 = S_r(S'_r(\tau))$, $\{S'_r(\varrho), \varrho'\} \subseteq \varphi''$ and $\mathit{Clean}\,(\varphi'')$. Since the transition

$$(S'_r(\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_3\rangle_\varrho) : S'_r(\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \varrho), S'_r(\varphi)) \xrightarrow{\gamma} \\ (S'_r(\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, e''_3\rangle_\varrho)\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' : (\pi_0, \varrho'), \varphi'')$$

is possible by Lemma 5.6.5, and $\{\,\} \rhd e_3 \lesssim e_2 : \pi, \varphi$ holds too, we have by definition of $\lesssim$ that

$$(S'_r(e_2) : S'_r(\forall\, \varrho_1, \ldots, \varrho_n\, .\tau, \varrho), S'_r(\varphi)) \xrightarrow{\gamma} (e_2\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' : (\pi_0, \varrho'), \varphi'')$$

and $\{\,\} \rhd S'_r(e_3)\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho' \lesssim e'_2 : (\pi_0, \varrho'), \varphi''$ where we take e'_2 to be $S'_r(e_2)\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho'$. Combining this with (5.31) and Lemma 5.7.1 proves that $\{\,\} \rhd e'_1 \lesssim^* e'_2 : \pi', \varphi''$ and hence the case.

Cases *(Obs-Dealloc)*, *(Obs-Dang)* and *(Obs-Fixdead)*. Because of Lemma 5.6.2 we have that $e_1 \downarrow$ and hence by Lemma 5.7.5 also $e_2 \downarrow$. The transition from e_1 goes to the expression $\Omega_{\varrho_\star}$ and by Lemma 5.6.5, we therefore have a transition from e_2 to $\Omega_{\varrho_\star}$ as well. Reflexivity of $\lesssim$ proves the case.

□

The actual theorem now follows easily:

Theorem 5.7.8 (Compatibility of Bisimilarity) $\approx^\circ$ *is compatible.*

Proof. We already have by Proposition 5.7.2.3 that $\lesssim^\circ \subseteq \lesssim^*$. For the other direction, assume $TE \rhd e_1 \lesssim^* e_2 : \pi, \varphi$ and take a ξ and S_r for which $S_r(TE) \succ \xi$ with $\mathit{Clean}\,(S_r(\varphi))$. Now, by Proposition 5.7.2.2 and Lemma 5.7.3, it is easy to prove that $\{\,\} \rhd \xi(S_r(e_1)) \lesssim^* \xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)$. So, from Lemma 5.7.7 we obtain $\{\,\} \rhd \xi(S_r(e_1)) \lesssim \xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)$, which by definition of completion implies $TE \rhd e_1 \lesssim^\circ e_2 : \pi, \varphi$.

Compatibility of $\approx^\circ$ follows immediately by Proposition 5.6.11. □

5.8 The Equational Theory is a Bisimulation

Bisimulation provides a strong proof principle, which we now use to prove that the equational theory of Section 5.5 is included in bisimilarity. We need two lemmas, which show that the region abstraction $\mathtt{new}\,\varrho.\, e$ is bisimilar to the expression e provided the well-typedness conditions hold. Here we use the refined notion of bisimulation *up to* $\approx$ (see page 93).

We first show the result for bisimilarity:

Lemma 5.8.1 *Suppose* $\{\,\} \vdash_f e : \pi, \varphi'$ *where* $\varrho \notin \mathit{frv}\,(\pi)$ *and* $\varphi' \subseteq \varphi$ *with* $\mathit{Clean}\,(\varphi)$, *then* $\{\,\} \rhd e \approx \mathtt{new}\,\varrho.\, e : \pi, \varphi$.

Proof. As usual, define the relation

$$\mathcal{R} = \{(\{\,\}, e, \mathtt{new}\,\varrho.\,e, \pi, \varphi) \mid \begin{array}{l} \{\,\} \vdash_f e : \pi, \varphi' \textit{ such that } \varphi' \subseteq \varphi \textit{ with} \\ \textit{Clean}\,(\varphi) \textit{ and } \varrho \notin \textit{frv}\,(\pi)\} \end{array}$$

It is immediate that $\mathcal{R}$ is well-formed, hence, by co-induction we have to show that $\mathcal{R} \subseteq [(\approx \mathcal{R} \approx) \cup \approx]$ or $\mathcal{R} \subseteq \lceil(\approx \mathcal{R} \approx) \cup \approx\rceil \cap \lceil(\approx \mathcal{R}^{-1} \approx) \cup \approx\rceil^{-1}$. We show that $\mathcal{R} \subseteq \lceil(\approx \mathcal{R} \approx) \cup \approx\rceil$, the other case is by similar reasoning. Take $\{\,\} \rhd e\,\mathcal{R}\,\mathtt{new}\,\varrho.\,e : \pi, \varphi$ and a transition $(S_r(e) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_0 : \pi_0, \varphi_0)$ such that $\textit{Clean}\,(S_r(\varphi))$. First, we can assume without loss of generalization that, by α-renaming, $\varrho \notin \textit{frv}\,(\textit{Img}\,(S_r)) \cup \textit{frv}\,(\gamma)$. From Lemma 5.6.2, we have that $e \downarrow$ and hence by Lemma 5.2.9, also that $S_r(e) \downarrow$. By reduction rule (5.5), this implies that $S_r(\mathtt{new}\,\varrho.\,e) \downarrow$ as well. Hence, from Lemma 5.6.1 we have the transition $(S_r(\mathtt{new}\,\varrho.\,e) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_0' : \pi_0, \varphi_0)$.

Whenever $\textit{Deadly}\,(\gamma)$, then trivially $\{\,\} \rhd e_0 \lesssim e_0' : \pi_0, \varphi_0 \cup \varphi''$ for $\textit{Clean}\,(\varphi'')$ by reflexivity of $\approx^\circ$. So, assume $\textit{Safe}\,(\gamma)$. Then we have $S_r(e) \xrightarrow{*} v$ and $S_r(\mathtt{new}\,\varrho.\,e) \xrightarrow{*} v[\varrho \mapsto \bullet]$ where $e_0 = v\gamma$ and $e_0' = (v[\varrho \mapsto \bullet])\gamma$.

First, observe that by either rule *(Obs-Lam)* or *(Obs-Fix)* we have that $\varrho \notin \textit{frv}\,(\pi_0) \cup \textit{frv}\,(\varphi_0)$. Also, since $\varrho \notin \textit{frv}\,(\gamma)$, we have that $e_0' = (v\gamma)[\varrho \mapsto \bullet]$. Since $\textit{Clean}\,(\varphi_0)$ and $\{\,\} \vdash_f v\gamma : \pi_0, \varphi_0$, we have that $v\gamma \downarrow$ or $v\gamma \uparrow$. Moreover, if $v\gamma \uparrow$ then also $(v\gamma)[\varrho \mapsto \bullet] \uparrow$ and the case is proven by Lemma 5.6.13. Similarly, if we have $v\gamma \downarrow$, then $(v\gamma)[\varrho \mapsto \bullet] \downarrow$, so assume $v\gamma \xrightarrow{*} v'$ and so by Lemma 5.2.9 also $(v\gamma)[\varrho \mapsto \bullet] \xrightarrow{*} v'[\varrho \mapsto \bullet]$. Using reduction rule (5.5), transitivity of $\approx$ and Lemma 5.6.14, we therefore have $\{\,\} \rhd e_0 \approx v' : \pi_0, \varphi_0$ and $\{\,\} \rhd \mathtt{new}\,\varrho.\,v' \approx e_0' : \pi_0, \varphi_0$. Therefore we have that $\{\,\} \rhd e_0 \approx \mathcal{R} \approx e_0' : \pi_0, \varphi_0$, proving the lemma.

□

Now, we extend the result to open extensions:

Lemma 5.8.2 *Suppose* $TE \vdash_f e : \pi, \varphi'$ *where* $\varrho \notin \textit{frv}\,(\pi, TE)$ *and* $\varphi' \subseteq \varphi$ *with* $\textit{Clean}\,(\varphi)$, *then* $TE \rhd e \approx^\circ \mathtt{new}\,\varrho.\,e : \pi, \varphi$.

Proof. To prove the result for open extensions, we take a ξ and a S_r such that $S_r(TE) \succ \xi$ and $\textit{Clean}\,(S_r(\varphi))$. By definition of open extension, it is sufficient to prove that $\{\,\} \rhd \xi(S_r(e)) \approx \xi(S_r(\mathtt{new}\,\varrho.\,e)) : S_r(\pi), S_r(\varphi)$. By α-renaming, we can assure that $\varrho \notin \textit{frv}\,(\xi) \cup \textit{frv}\,(\textit{Img}\,(S_r))$, hence the assertion is equivalent to $\{\,\} \rhd \xi(S_r(e)) \approx \mathtt{new}\,\varrho.\,\xi(S_r(e)) : S_r(\pi), S_r(\varphi)$.

We have $\textit{Clean}\,(S_r(\varphi))$ and because $\varrho \notin \textit{frv}\,(\pi)$, we also have $\varrho \notin \textit{frv}\,(S_r(\pi))$. Moreover, since $TE \vdash_f e : \pi, \varphi'$ and by Lemma 5.3.1, we can apply Lemma 5.8.1 to prove the result. □

Now, we can prove the main theorem:

Theorem 5.8.3 $\triangleq \subseteq \approx^\circ$.

Proof. We have to show that $\approx^\circ$ satisfies the rules in Figure 5.6. Then, because $\triangleq$ is the least relation satisfying these rules, we automatically have $\triangleq \subseteq \approx^\circ$. Observe that by Lemma 5.5.1, we already have that $\triangleq$ is well-formed.

(Eq-Comp). Immediate by Theorem 5.7.8.

(Eq-Symm) and *(Eq-Trans).* Immediate from Proposition 5.6.10.3.

(Eq-Beta-n) with $n \in \{1,2,5,6\}$. Suppose $TE \vdash_f e : \pi, \varphi'$ where $\varphi' \subseteq \varphi$ with $\mathit{Clean}\,(\varphi)$ and $e \xrightarrow{n} e'$. For arbitrary ξ and S_r where $S_r(TE) \succ \xi$ with $\mathit{Clean}\,(S_r(\varphi))$, we have by Lemma 5.3.1 that $\{\,\} \vdash_f \xi(S_r(e)) : S_r(\pi), S_r(\varphi')$. Also, by Lemma 5.2.8 and Lemma 5.2.9, we have that $\xi(S_r(e)) \xrightarrow{n} \xi(S_r(e'))$. So, by Lemma 5.6.14 (and via reduction rule (5.7)), it holds that $\{\,\} \rhd \xi(S_r(e)) \approx \xi(S_r(e')) : S_r(\pi), S_r(\varphi)$ and this for arbitrary ξ and S_r with $S_r(TE) \succ \xi$. Hence, by definition of open extension, we obtain that $TE \rhd e \approx^\circ e' : \pi, \varphi$.

(Eq-App-Cbv). Given that $TE \vdash_f \langle \lambda\, x.\, e_1 \rangle_\varrho : (\pi_1 \xrightarrow{\epsilon.\varphi''} \pi, \varrho), \emptyset$ and $TE \vdash_f e_2 : \pi_1, \varphi'$ with $\varphi' \cup \varphi'' \cup \{\epsilon, \varrho\} \subseteq \varphi$ and $\mathit{Clean}\,(\varphi)$, we have to prove that $TE \rhd \langle \lambda\, x.\, e_1 \rangle_\varrho \mathbin{@} e_2 \approx^\circ \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 : \pi, \varphi$. Without loss of generality, by α-renaming, let us assume that $x \notin \mathit{Dom}\,(TE)$. By definition of completion, this implies that x does not occur free in ξ either, where we take ξ and S_r such that $S_r(TE) \succ \xi$ with $\mathit{Clean}\,(S_r(\varphi))$. We have to show $\{\,\} \rhd \langle \lambda\, x.\, \xi(S_r(e_1)) \rangle_{S_r(\varrho)} \mathbin{@} \xi(S_r(e_2)) \approx \mathtt{let}\ x = \xi(S_r(e_2))\ \mathtt{in}\ \xi(S_r(e_1)) : S_r(\pi), S_r(\varphi)$.

First, we can easily see that if $\langle \lambda\, x.\, \xi(S_r(e_1)) \rangle_{S_r(\varrho)} \mathbin{@} \xi(S_r(e_2)) \downarrow$, then $(\mathtt{let}\ x = \xi(S_r(e_2))\ \mathtt{in}\ \xi(S_r(e_1))) \downarrow$ and the other way around. So, by Lemma 5.6.13, we have that the two terms are bisimilar.

Suppose now that $\langle \lambda\, x.\, \xi(S_r(e_1)) \rangle_{S_r(\varrho)} \mathbin{@} \xi(S_r(e_2)) \xrightarrow{*} \langle \lambda\, x.\, \xi(S_r(e_1)) \rangle_{S_r(\varrho)} \mathbin{@} v_2 \xrightarrow{*} \xi(S_r(e_1))[x \mapsto v_2]$ where we assume $\xi(S_r(e_2)) \xrightarrow{*} v_2$. Then, by Lemma 5.6.14, we obtain $\{\,\} \rhd \xi(S_r(e_1)) \mathbin{@} \xi(S_r(e_2)) \approx \xi(S_r(e_1))[x \mapsto v_2] : S_r(\pi), S_r(\varphi)$. We also have $\mathtt{let}\, x = \xi(S_r(e_2))\, \mathtt{in} \xi(S_r(e_1)) \xrightarrow{*} \mathtt{let}\, x = v_2\, \mathtt{in} \xi(S_r(e_1)) \xrightarrow{*} \xi(S_r(e_1))[x \mapsto v_2]$, so again by Lemma 5.6.14 we obtain $\{\,\} \rhd \mathtt{let}\ x = \xi(S_r(e_2))\ \mathtt{in}\ \xi(S_r(e_1)) \approx \xi(S_r(e_1))[x \mapsto v_2] : S_r(\pi), S_r(\varphi)$. Bisimilarity of the two terms now follows by transitivity and symmetry of the relation.

The fact that rule *(Eq-App-Let)* is included in bisimilarity, is a consequence of a result for the weaker notion of *Kleene-equivalence* [49,117]: Given $\{\,\} \vdash_f e_1 : \pi, \varphi_1$ and $\{\,\} \vdash_f e_2 : \pi, \varphi_2$ with $\mathit{Clean}\,(\varphi_1 \cup \varphi_2)$, then e_1 is Kleene-equivalent to e_2 iff there exists a v such that $e_1 \xrightarrow{*} v$ and $e_2 \xrightarrow{*} v$. By similar reasoning as above, it is easy to prove that Kleene-equivalent terms are always bisimilar.

(Eq-App-Let) and *(Eq-Let-Let).* It is easy to see that the two sides both the rules *(Eq-App-Let)* and *(Eq-Let-Let)* are Kleene-equivalent after performing an arbitrary completion. So, analogous to the previous case, we can show that they are bisimilar.

(Eq-Drop). Immediate by Lemma 5.8.2.

(Eq-Dealloc). Since we have $\mathit{Clean}\,(\varphi)$, by Type Soundness (Proposition 5.2.6) either $e \downarrow$ (and since $\varrho \notin \varphi$ also $e[\varrho \mapsto \bullet] \downarrow$) or $e \uparrow$ (and therefore $e[\varrho \mapsto \bullet] \uparrow$).

Assume $e \downarrow$ first. Then bisimilarity follows immediately by Lemma 5.6.13. So, suppose we have $e \downarrow$ and $e[\varrho \mapsto \bullet] \downarrow$, so we can assume that $e \xrightarrow{*} v$ and $e[\varrho \mapsto \bullet] \xrightarrow{*} v[\varrho \mapsto \bullet]$. Therefore, we obtain by Lemma 5.6.14 that $\{\,\} \rhd e \approx v : \pi, \varphi$ and $\{\,\} \rhd v[\varrho \mapsto \bullet] \approx e[\varrho \mapsto \bullet] : \pi, \varphi$.

Also, by reduction (5.3) and the same Lemma we have $\{\,\} \rhd \mathtt{new}\, \varrho.\, v \approx v[\varrho \mapsto \bullet] : \pi, \varphi$ too. By transitivity of $\approx$ and Lemma 5.8.2, we prove the case.

□

5.9 Bisimilarity equals Contextual Equivalence

Now that we have shown that the equational theory $\triangleq$ is included in the bisimilarity relation $\approx$, we prove that the latter is equivalent to the contextual equivalence relation $\simeq$. As a corollary, we derive that the equational theory is sound with respect to contextual equivalence.

Proposition 5.9.1 *$\approx^\circ$ is well-formed, compatible and adequate.*

Proof. From Proposition 5.6.10.1 we have that $\approx^\circ$ is well-formed and by Theorem 5.7.8, we also know it is compatible. Adequacy follows from Lemma 5.6.12. □

We can now prove a Context Lemma, but instead of taking context rules [99], we only consider a sequence of safe (and well-typed) observations:

Lemma 5.9.2 *$\{\,\} \rhd e_1 \lesssim e_2 : \pi, \varphi$ iff $\{\,\} \vdash_f e_1 : \pi, \varphi_1$ and $\{\,\} \vdash_f e_2 : \pi, \varphi_2$ where $\varphi_1 \cup \varphi_2 \subseteq \varphi$ and for all S_r with Clean $(S_r(\varphi))$ and for all observations $\gamma_1, \ldots, \gamma_n$ such that for all $i \in \{1, \ldots, n\}$ where Safe (γ_i) and $\{\,\} \vdash_f S_r(e_1)\gamma_1 \ldots \gamma_i : \pi^i, \varphi_1^i$ and $\{\,\} \vdash_f S_r(e_2)\gamma_1 \ldots \gamma_i : \pi^i, \varphi_2^i$, it holds that $S_r(e_1)\gamma_1 \ldots \gamma_n \downarrow \Rightarrow S_r(e_2)\gamma_1 \ldots \gamma_n \downarrow$.*

Proof.

Left-to-right Take a sequence of observations $\gamma_1, \ldots, \gamma_n$ such that for all $i \in \{1, \ldots, n\}$ we have that *Safe* (γ_i) where $\{\,\} \vdash_f S_r(e_1)\gamma_1 \ldots \gamma_i : \pi^i, \varphi_1^i$ and $\{\,\} \vdash_f S_r(e_2)\gamma_1 \ldots \gamma_i : \pi^i, \varphi_2^i$. Since we have that $S_r(e_1)\gamma_1 \ldots \gamma_n \downarrow$, by reduction rules (5.8) and (5.11) the following reductions hold: $S_r(e_1) \xrightarrow{*} v_1^1$; $v_1^1\gamma_1 \xrightarrow{*} v_1^2$; $v_1^2\gamma_2 \xrightarrow{*} v_1^3$; ... and $v_1^n\gamma_n \downarrow$. By Lemma 5.6.1 and Proposition 5.2.3, this implies the existence of a transition $(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma_1} (v_1^1\gamma_1 : \pi^1, \varphi^1)$ and for $i \in \{1, \ldots, n-1\}$ also the transitions $(v_1^i\gamma_i : \pi^i, \varphi^i) \xrightarrow{\gamma_{i+1}} (v_1^{i+1}\gamma_{i+1} : \pi^{i+1}, \varphi^{i+1})$.

Hence, by the similarity, we also have $(S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma_1} (v_2^1\gamma_1 : \pi^1, \varphi^1)$ and for $i \in \{1, \ldots, n-1\}$ the transitions $(v_2^i\gamma_i : \pi^i, \varphi^i) \xrightarrow{\gamma_{i+1}} (v_2^{i+1}\gamma_{i+1} : \pi^{i+1}, \varphi^{i+1})$ where $\{\,\} \rhd v_1^i\gamma_i \lesssim v_2^i\gamma_i : \pi^i, \varphi^i$ and also $\{\,\} \rhd v_1^n\gamma_n \lesssim v_2^n\gamma_n : \pi^n, \varphi^n$. From the latter and since $v_1^n\gamma_n \downarrow$, we have by Lemma 5.6.1, that $v_2^n\gamma_n \downarrow$ too. Applying

Lemma 5.6.2 subsequently on the transitions from $v_2^i\gamma_i$, makes that $S_r(e_2) \xrightarrow{*} v_2^1$; $v_2^1\gamma_1 \xrightarrow{*} v_2^2$; $v_2^2\gamma_2 \xrightarrow{*} v_2^3$; ... too. Therefore we have $S_r(e_2)\gamma_1 \dots \gamma_n \downarrow$, which proves the case.

Right-to-left We prove by co-induction, so define the set

$$\begin{aligned}\mathcal{R} = \{(\{\,\}, e_1, e_2, \pi, \varphi) \mid \{\,\} \vdash_f e_1 : \pi, \varphi_1 \textit{ and } \{\,\} \vdash_f e_2 : \pi, \varphi_2 \textit{ where } \varphi_1 \cup \varphi_2 \subseteq \varphi \\ \textit{and for all } S_r \textit{ with } \mathit{Clean}\,(S_r(\varphi)) \textit{ and for all } \gamma_1, \dots, \gamma_n \textit{ such that for all} \\ i \in \{1, \dots, n\} \textit{ we have } \mathit{Safe}\,(\gamma_i) \textit{ and } \{\,\} \vdash_f S_r(e_1)\gamma_1 \dots \gamma_i : \pi^i, \varphi_1^i \textit{ and} \\ \{\,\} \vdash_f S_r(e_2)\gamma_1 \dots \gamma_i : \pi^i, \varphi_2^i \textit{ it holds that} \\ S_r(e_1)\gamma_1 \dots \gamma_n \downarrow \Rightarrow S_r(e_2)\gamma_1 \dots \gamma_n \downarrow\}\end{aligned}$$

Since $\mathcal{R}$ is well-formed, it is sufficient to show that $\mathcal{R} \subseteq \lceil \mathcal{R} \cup \lesssim \rceil$. Take $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ and suppose transition $(S_r(e_1) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_1' : \pi', \varphi')$ such that $\mathit{Clean}\,(S_r(\varphi))$. This implies that $S_r(e_1) \downarrow$ and since $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$ we therefore have $S_r(e_2) \downarrow$ (where we take the empty sequence of observations). By Lemma 5.6.5, $(S_r(e_2) : S_r(\pi), S_r(\varphi)) \xrightarrow{\gamma} (e_2' : \pi', \varphi')$ exists.

Whenever $\mathit{Deadly}\,(S_r(\gamma))$, then the claim is proven by reflexivity of $\lesssim$. Therefore, assume $\mathit{Safe}\,(S_r(\gamma))$. So, we also have that $S_r(e_1) \xrightarrow{*} v_1$ with $e_1' = v_1\gamma$ and $S_r(e_2) \xrightarrow{*} v_2$ with $e_2' = v_2\gamma$. By Lemma 5.6.3, we also have that $\{\,\} \vdash_f e_1' : \pi', \varphi'$ and $\{\,\} \vdash_f e_2' : \pi', \varphi'$.

Now, take an arbitrary sequence of observations $\gamma_1, \dots, \gamma_n$ such that for all $i \in \{1, \dots, n\}$ it holds that $\mathit{Safe}\,(\gamma_i)$ and $\{\,\} \vdash_f S_r(e_1)\gamma_1 \dots \gamma_i : \pi^i, \varphi_1^i$ and $\{\,\} \vdash_f S_r(e_2)\gamma_1 \dots \gamma_i : \pi^i, \varphi_2^i$. Now assume that $v_1 S_r(\gamma)\gamma_1 \dots \gamma_n \downarrow$. Since $S_r(e_1) \xrightarrow{*} v_1$, we have because of the reductions (5.8) and (5.11) that $S_r(e_1\gamma)\gamma_1 \dots \gamma_n \downarrow$. Because $\{\,\} \rhd e_1 \,\mathcal{R}\, e_2 : \pi, \varphi$, this implies $S_r(e_2\gamma)\gamma_1 \dots \gamma_n \downarrow$. But, we also have $S_r(e_2) \xrightarrow{*} v_2$ and again using the reductions (5.8) and (5.11) that $v_2 S_r(\gamma)\gamma_1 \dots \gamma_n \downarrow$. It now follows that $\{\,\} \rhd e_1' \,\mathcal{R}\, e_2' : \pi', \varphi'$ (the typings are trivially checked). By definition of $\mathcal{R}$, we also have $\{\,\} \rhd e_1' \,\mathcal{R}\, e_2' : \pi', \varphi' \cup \varphi''$ with $\mathit{Clean}\,(\varphi'')$, proving the claim.

□

Proposition 5.9.3 *If* $\{\,\} \rhd e_1 \simeq e_2 : \pi, \varphi$, *then* $\{\,\} \rhd e_1 \approx e_2 : \pi, \varphi$.

Proof. We have $\{\,\} \rhd e_1 \simeq e_2 : \pi, \varphi$ and hence, by definition that $\{\,\} \vdash_f e_1 : \pi, \varphi_1$ and $\{\,\} \vdash_f e_2 : \pi, \varphi_2$ with $\varphi_1 \cup \varphi_2 \subseteq \varphi$ and $\mathit{Clean}\,(\varphi)$. Then, take S_r such that $\mathit{Clean}\,(S_r(\varphi))$ and a sequence of observations $\gamma_1, \dots, \gamma_n$ such that for all $i \in \{1, \dots, n\}$ it holds that $\mathit{Safe}\,(\gamma_i)$ and $\{\,\} \vdash_f S_r(e_1)\gamma_1 \dots \gamma_i : \pi^i, \varphi^i$ and $\{\,\} \vdash_f S_r(e_2)\gamma_1 \dots \gamma_i : \pi^i, \varphi^i$. If we can prove that $S_r(e_1)\gamma_1 \dots \gamma_n \downarrow \Leftrightarrow S_r(e_2)\gamma_1 \dots \gamma_n \downarrow$, the Proposition follows directly from Context Lemma 5.9.2.

To show the latter, let us assume that $\mathit{Supp}\,(S_r) = \{\varrho_1, \dots, \varrho_n\}$ and $\mathit{Img}\,(S_r) = \{\rho_1', \dots, \rho_n'\}$. By compatibility of $\simeq$ and the context rules *(Comp-Lamv)*, *(Comp-Fixv)*, and *(Comp-Regapp)* we have that $\{\,\} \rhd e_1' \simeq e_2' : S_r(\pi), \varphi \cup \{\varrho\}$, where $\varrho \notin \mathit{Supp}\,(S_r)$,

the expression $e'_1 = (\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e_1\rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho) @ \langle \lambda\, x.\ x\rangle_\varrho$ and $e'_2 = (\langle \mathtt{fix}\, f.\Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ e_2\rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho) @ \langle \lambda\, x.\ x\rangle_\varrho$, assuming $\{x, f\} \cap (fv\,(e_1) \cup fv\,(e_2)) = \emptyset$.

Additionally, by context rules *(Comp-App)* and *(Comp-Regapp)*, we can derive that $\{\ \} \rhd e'_1\gamma_1 \ldots \gamma_n\ \{(\{\ \}, e'_1, e'_2, S_r(\pi), \varphi \cup \{\varrho\})\}^c\ e'_2\gamma_1 \ldots \gamma_n : \pi^n, \varphi^n$. Since contextual equivalence makes $\{(\{\ \}, e'_1, e'_2, S_r(\pi), \varphi \cup \{\varrho\})\}^c$ adequate, we also have $e'_1\gamma_1 \ldots \gamma_n \downarrow\!\Leftrightarrow e'_2\gamma_1 \ldots \gamma_n \downarrow$. From the definition of e'_1 and e'_2, it is immediate that $S_r(e_1)\gamma_1 \ldots \gamma_n \downarrow\!\Leftrightarrow S_r(e_2)\gamma_1 \ldots \gamma_n \downarrow$ also holds. □

We conclude the chapter by formulating the main equivalence theorem: the notion of applicative bisimilarity for the λ_f^{region}-calculus is an alternative characterization of contextual equivalence.

Theorem 5.9.4 (Bisimilarity equals Contextual Equivalence) $\approx^\circ = \simeq$

Proof. As an immediate consequence of Proposition 5.4.7 and Proposition 5.9.1, we have that $\approx^\circ \subseteq \simeq$. Remains to show that $\simeq \supseteq \approx^\circ$. Suppose $TE \rhd e_1 \simeq e_2 : \pi, \varphi$, an iterated substitution ξ and a region substitution S_r for which $TE \succ \xi$ such that $Clean\,(S_r(\varphi))$. To prove that $TE \rhd e_1 \approx^\circ e_2 : \pi, \varphi$, we have to show that $\{\ \} \rhd \xi(S_r(e_1)) \approx \xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)$.

By Proposition 5.4.7 and Lemma 5.4.2, we have $\{\ \} \rhd \mathcal{T}(\xi, S_r(e_1)) \simeq \mathcal{T}(\xi, S_r(e_2)) : S_r(\pi), S_r(\varphi)$. Also, by Theorem 5.8.3 and Lemma 5.5.2 we have $\{\ \} \rhd \mathcal{T}(\xi, S_r(e_1)) \simeq \xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)$ and $\{\ \} \rhd \xi(S_r(e_1)) \simeq \mathcal{T}(\xi, S_r(e_2)) : S_r(\pi), S_r(\varphi)$. So, by transitivity of $\simeq$, this implies that $\{\ \} \rhd \xi(S_r(e_1)) \simeq \xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)$ and hence, by Proposition 5.9.3, it follows that $\{\ \} \rhd \xi(S_r(e_1)) \approx \xi(S_r(e_2)) : S_r(\pi), S_r(\varphi)$. □

5.10 Related Work

Whereas type soundness properties for region calculi have been extensively examined [9, 18,28,68,150,160] (see also the material in Chapter 3 and Chapter 4), little attention has been paid to the development of a semantic framework to reason about region-annotated programs.

In contrast, operational techniques for proving equivalences within the lambda calculus and its many variations, for example PCF, are old. Morris [108] first introduced the notion of contextual equivalence and Plotkin [121] made it a standard notion for program equivalence. However, since it is not very practical to prove contextual equivalence directly, alternative characterizations for operational program equivalence have been proposed. For instance, Milner [99] introduced his context lemma to reduce the need for Plotkin's full-blown contexts to evaluation contexts.

CCS-style bisimilarity is originally only used in process calculi [101, 125], where a derivation tree is an important semantical notion. Its use as a simpler operational method to reason about equivalences in functional programming languages dates from the early 90s, when Abramsky [2] and Abramsky and Ong [3] define a notion of applicative

bisimilarity for the lazy lambda calculus. Since then, bisimilarity has been extended and fine-tuned by Gordon [48–50] and Crole and Gordon [29] to other programming models. Notably, Gordon [50] proved that a form of bisimilarity for PCF with lazy streams is fully abstract with respect to contextual equivalence.

Applicative bisimilarity is not always complete with respect to contextual equivalence, for example, Gordon and Rees [53] found that a translation of Abramsky's applicative bisimilarity for a first-order object calculus with sub-typing tells more programs apart than contextual equivalence. Experimental equivalence, a co-inductive variant of Milner's context lemma, turns out to be complete for such object calculi [49]. The bisimilarity notion presented in this chapter is applicative and complete with respect to contextual equivalence.

Although all region calculi are designed to make operations on regions explicit, they do not necessarily provide for local state. This is a non-trivial addition with respect to an equational theory. Mason and Talcott define a context lemma particularly tailored towards state operations. This is known as CIU-equivalence [95–97, 135]. Pitts [116] and Pitts and Stark [118] use logical relations and an unwinding theorem to prove equivalences in state-based languages. In this chapter, we did not pursue these alternative methods since bisimilarity seems to work well for state-less polymorphic calculi as witnessed by others [49, 114].

As far as we know, the only other work which directly discusses equational rules for the region calculus is a paper by Dal Zilio and Gordon on the pi-calculus with name groups [160]. They derive their rules from similar rules for their extended pi-calculus and conjecture soundness. However, they do not give proofs, treat only a monomorphic calculus and have a somewhat awkward equational theory due to a particular choice of formulating the type system.

5.11 Chapter Summary

In order to formulate an equational theory for a region calculus, we adapted the λ_s^{region}-calculus of Chapter 3 in Section 5.2 to incorporate first-class recursive function objects, resulting in the λ_f^{region}-calculus. Addressing some preliminary definitions in Section 5.3, we defined a notion of contextual equivalence for the λ_f^{region}-calculus in Section 5.4. It defines an operational approach to program equivalence, to which we want to prove the equational theory of Section 5.5 sound. To do so, Section 5.6 introduced a form of applicative bisimilarity for the λ_f^{region}-calculus. We proved it compatible in Section 5.7, a property we exploited in Section 5.8 to show that the equational equivalence relation is a bisimulation. Finally, in Section 5.9, we showed that our bisimilarity is in fact an alternative formulation for contextual equivalence, a result which implies that the equational theory is contextual sound. We use the equational theory for the λ_f^{region}-calculus in Chapter 6 to prove soundness of a specializer for an ML-polymorphic language.

6 POLYMORPHIC SPECIALIZATION FOR ML

6.1 Introduction

In the previous chapters, we have built a comprehensive operational framework to reason syntactically about expressions and types within the region calculus. The main motivation to develop these theoretical tools is to be able to prove the correctness of an offline partial evaluation method for ML-like languages.

In this chapter, we present a novel technique for the offline partial evaluation of functional languages with an ML-style typing discipline. Our program specialization method comprises a polymorphic binding-time analysis with polymorphic recursion. It is based on the λ_s^{region}-calculus and we specify a binding-time analysis as a constraint analysis on top of region inference. Our insight is regarding binding times as properties of regions. This view allows us to extend work by Dussart, Henglein and Mossin [36, 73] in a natural way to Hindley-Milner polymorphic languages.

As in previous chapters, we specify specialization as a small-step semantics and extend standard evaluation of the λ_s^{region}-calculus. Following the approach of Hatcliff and Danvy [60], we use syntactic type soundness techniques to prove that the constraint analysis is sound with respect to the specializer. In addition, we prove that specialization preserves the call-by-value semantics of the λ_s^{region}-calculus (see Section 3.2) by showing that the reductions of the specializer are contextual equivalences in the λ_s^{region}-calculus. To this end, we actually show that every specialization rule is sound with respect to the equational theory of Figure 5.6 on page 86.

The specialization theory of this chapter is submitted for publication [69].

6.2 A Constraint Analysis

The static semantics of an offline partial evaluator is usually called a *binding-time analysis* in the literature [82]. In this section, we specify it as a *constraint analysis* on top of the region type system of Figure 2.6 on page 27. The constraint analysis formalizes the generation of constraints between regions. Since the λ_{tt}^{region}-calculus employs polymorphic recursion for regions, we generally have to solve constraints at runtime. Therefore, we will use the term constraint analysis instead of binding-time analysis in the rest of this chapter.

$$\rho \in \mathsf{Placeholder} = \mathsf{RegionVar} \qquad C \in \mathsf{ConstraintSet} = \mathcal{P}(\{\rho_1 \leq \rho_2 \mid \rho_1, \rho_2 \in \mathsf{Placeholder}\})$$

$$\begin{aligned}\mathsf{Term} \ni e ::= \; & x \mid c\ \mathtt{at}\ \rho \mid \lambda x.\, e\ \mathtt{at}\ \rho \mid e\,@\,e \mid \mathtt{lift}\,[\rho_1, \rho_2]\,e \mid \mathtt{new}\,\varrho : C\,.e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \\ & f\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \mathtt{letrec}\ f = \Lambda\,\varrho_1, \ldots, \varrho_n.\lambda x.\, e\ \mathtt{at}\ \rho\ \mathtt{in}\ e\end{aligned}$$

Figure 6.1: Syntax of the λ_C^{region}-calculus

6.2.1 The Constraint-annotated λ_C^{region}-calculus

Although we formalize program specialization by means of a small-step reduction semantics, we do not yet consider computational terms for the constraint analysis. Following standard practice [71], the constraint analysis is specified as a type system which extends the $\vdash_t^t$-typing for the λ_{tt}^{region}-calculus. It yields a constraint-annotated λ_C^{region}-Term, which is generated by the grammar in Figure 6.1.

The syntax of the λ_C^{region}-calculus is very similar to that of the λ_{tt}^{region}-calculus as defined in Figure 2.4 on page 22. The main distinction is the added constraint set in the region abstraction construct $\mathtt{new}\,\varrho : C\,.e$. A constraint set C contains ordered pairs of regions and imposes *binding-time constraints* as we shall explain below. In the region abstraction $\mathtt{new}\,\varrho : C\,.e$, the constraint set C must be solved to determine the binding time of the region ϱ. We will make this mechanism more precise when we discuss specialization.

The primitive `lift`-operator is identical to the `copy`-primitive of the λ_{tt}^{region}-calculus. However, we change its name since we will extend its meaning in Section 6.3.2. Also note that the set of λ_C^{region}-Placeholders only includes the region variables RegionVar. As usual, we will extend this set to include other semantic objects when we discuss a transition semantics.

Define the *erasure* as a total function $|\cdot| \in \lambda_C^{region}\text{-Term} \to \lambda_{tt}^{region}\text{-Term}$. It transforms a λ_C^{region}-expression into a λ_{tt}^{region}-expression by stripping the constraint sets C from the region abstractions.

6.2.2 Semantics Objects

Most semantic objects required in the constraint analysis are reminiscent of those for the type system of the λ_{tt}^{region}-calculus (see Section 2.3.3). We briefly summarize the extensions and changes relevant for the static semantics of the λ_C^{region}-calculus.

The type language differs only in its type schemes, which not only quantify type, region and effect variables, but also *qualify* the type scheme with a constraint set C:

$$\mathsf{TypeScheme} \ni \sigma ::= \forall\,\vec{\alpha}\,.\forall\,\vec{\epsilon}\,.\forall\,\vec{\varrho}\,.C \Rightarrow \tau$$

Constraint set qualifiers transport constraints from the function definition site to the function application site.

As with the λ_{tt}^{region}-calculus, arrow types contain an arrow effect, which is an effect variable paired with a latent effect. Effect variables are a technical device for region

reconstruction of the region calculus [147] and play no active role during the constraint analysis. However, it is worth mentioning that an effect variable identifies a group of functions that share at least one application. This corresponds to a simple control-flow analysis [112].

In the λ_{tt}^{region}-calculus, the latent effect of an arrow effect contains the regions affected by the body of the function. In a region inference context, it is usually discharged upon function application as witnessed by type rule *(TT-App)* in Figure 2.6 on page 27. In a partial evaluation setting, however, latent effects also discharge upon function creation to account for specialization-time effects. We will explain this somewhat more conservative approach to the effect analysis when we discuss the actual constraint analysis.

A substitution on semantic objects in the λ_{C}^{region}-calculus extends its definition from Section 2.3.3 to work on ConstraintSets and qualified λ_{C}^{region}-TypeSchemes:

$$S_s(C) = \{S_s(\rho_1) \leq S_s(\rho_2) \mid \rho_1 \leq \rho_2 \in C\}$$

$$S_s(\forall\,\vec{\alpha}\,.\forall\,\vec{\epsilon}\,.\forall\,\vec{\varrho}\,.C \Rightarrow \tau) = \forall\,\vec{\alpha}\,.\forall\,\vec{\epsilon}\,.\forall\,\vec{\varrho}\,.S'_s(C) \Rightarrow S'_s(\tau) \quad \textit{where } S'_s = S_s - \{\vec{\epsilon}, \vec{\varrho}, \vec{\alpha}\}$$

A pair (C, τ) is an *instance* of a qualified type scheme $\sigma = \forall\,\vec{\alpha}\,.\forall\,\vec{\epsilon}\,.\forall\,\vec{\varrho}\,.C' \Rightarrow \tau'$ via substitution S_s, written $(C, \tau) \prec \sigma$ *via* S_s, if $\mathit{Supp}\,(S_t) \subseteq \{\vec{\alpha}\}$, $\mathit{Supp}\,(S_e) \subseteq \{\vec{\epsilon}\}$ and $\mathit{Supp}\,(S_r) \subseteq \{\vec{\varrho}\}$, where $S_s(\tau') = \tau$ and $S_s(C') = C$. The instance relation extends to type schemes by $\sigma \prec \sigma'$ iff $(C, \tau) \prec \sigma$ *via* S_s implies $(C, \tau) \prec \sigma'$ *via* S'_s.

The *erasure* function $|\cdot|$ for λ_{C}^{region}-Terms naturally carries over to type schemes and type environments by stripping the qualifying constraint sets. So, we have $|\sigma| \in \lambda_{tt}^{region}$-TypeScheme and $|TE| \in \lambda_{tt}^{region}$-TypeEnv.

Definition 6.2.1 *Suppose a constraint set C.*

1. *C is* transitive closed *iff $\rho_1 \leq \rho_2 \in C$ and $\rho_2 \leq \rho_3 \in C$ implies $\rho_1 \leq \rho_3 \in C$.*
2. *A constraint set C' is* consistent *with respect to C (written $C \rhd C'$) iff C is transitive closed and $C' \subseteq C$.*

Given some constraint set C, there is always a least set $\widetilde{C}$ such that $\widetilde{C} \rhd C$. It is constructed by closing C under transitivity and we always have $\mathit{frv}\,(C) = \mathit{frv}\,(\widetilde{C})$. Constraint sets have upper and lower projections with respect to a set of regions:

$$C^{\uparrow\{\rho_1,\ldots,\rho_n\}} = \{\rho_0 \leq \rho'_0 \in C \mid \exists\, i \in \{1,\ldots,n\} : \rho_0 = \rho_i \vee \rho'_0 = \rho_i\}$$

$$C_{\downarrow\{\rho_1,\ldots,\rho_n\}} = C \setminus C^{\uparrow\{\rho_1,\ldots,\rho_n\}} \qquad C_{\downarrow\rho} = C_{\downarrow\{\rho\}} \qquad C^{\uparrow\rho} = C^{\uparrow\{\rho\}}$$

A binding time, β, ranges over the set $\{\mathsf{s}, \mathsf{d}\}$, which is ordered by the least partial order $\leq$ such that $\mathsf{s} \leq \mathsf{d}$, where s is called *static* and d *dynamic*. A *region annotation*, $\delta \in \mathsf{BtAnn} = (\mathsf{Placeholder} \hookrightarrow \{\mathsf{s}, \mathsf{d}\})$, is a partial function from regions to binding times.

A region annotation δ is a *solution* for a constraint set C (written $\delta \models C$) if $\mathit{frv}\,(C) \subseteq \mathit{Dom}\,(\delta)$ and for all $\rho_1 \leq \rho_2 \in C$ it holds that $\delta(\rho_1) \leq \delta(\rho_2)$. A constraint set C is *solvable* if it has a solution. The set of all solutions of C is closed under greatest lower bounds: if δ_1 and δ_2 are solutions, then so is $\min(\delta_1, \delta_2)$ (pointwise minimum). This implies that if C is solvable, then there exists a unique least solution, which is the minimum of all solutions.

6.2.3 Constraint Rules

A traditional specializer does not allow static values to be embedded in generated code. This property is enforced by the well-formedness of the binding-time annotations and is specified using constraints on the underlying type system. A type scheme, $\sigma \in \mathsf{TypeScheme}$, is well-formed with respect to constraint set C if the judgment $C \vdash_{wft} \sigma$ can be derived using the following rules:

$$\textit{(Wft-Base)} \quad \frac{C \rhd \emptyset}{C \vdash_{wft} (\mathtt{int}, \rho)} \qquad \textit{(Wft-Tyvar)} \quad \frac{C \rhd \emptyset}{C \vdash_{wft} (\alpha, \rho)}$$

$$\textit{(Wft-Fun)} \quad \frac{\begin{array}{c} C \rhd \{\rho \leq \rho_1, \rho \leq \rho_2\} \quad C \vdash_{wft} \mu_1 \\ C \vdash_{wft} \mu_2 \quad \mu_1 = (\tau_1, \rho_1) \quad \mu_2 = (\tau_2, \rho_2) \end{array}}{C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)}$$

$$\textit{(Wft-Scheme)} \quad \frac{C \vdash_{wft} (\tau, \rho)}{C_{\downarrow\{\vec{\varrho}\}} \vdash_{wft} (\forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}.\forall\, \vec{\varrho}\,.C^{\uparrow\{\vec{\varrho}\}} \Rightarrow \tau, \rho)}$$

This judgment only collects the necessary binding-time constraints for well-formedness, it does not guarantee that the resulting constraint set C is solvable.

For base types and type variables, rules *(Wft-Base)* and *(Wft-Tyvar)* respectively, require a transitive closed C, but no additional constraints. An arrow type is well-formed by rule *(Wft-Fun)*. It requires that the subparts of the arrow are well-formed and includes constraints in C which guarantee that the subparts of the arrow are dynamic whenever the function is dynamic. Rule *(Wft-Scheme)* generalizes the well-formedness property to a pair of a qualified type scheme and region. It does this by collecting all the constraints for the un-quantified type τ. The qualified constraint set only contains the constraints involving quantified regions. The constraint set of the context does not need these.

Using this definition, we can define *a well-formed type environment* TE with respect to a constraint set C (written $C \vdash_{wft} TE$) as follows: for all $x \in Dom\,(TE)$, it holds that $C \vdash_{wft} TE(x)$.

A constraint-type judgment has the form $TE, C \vdash_c e : \mu, \varphi$, which is read "given a type environment TE and a constraint set C, the λ_C^{region}-expression e has type μ and effect φ". The rules in Figure 6.2 specify the constraint analysis.

The rules *(C-Var)*, *(C-Const)*, *(C-App)*, *(C-Let)* and *(C-Effect)* are almost identical to the rules *(TT-Var)*, *(TT-Const)*, *(TT-App)*, *(TT-Let)* and *(TT-Effect)* from Figure 2.6 on page 27 respectively. In contrast with the type rules of the $\vdash_t^t$ typing judgment, constraint rules are designed to guarantee a consistent constraint set in the assumption of a judgment.

Rule *(C-Lift)* deals with copying a value from region ρ_1 to ρ_2. Accordingly, we require that ρ_1 is of earlier binding time than ρ_2. This allows for (first-order) *binding-time coercions* as we shall explain later. Rule *(C-Reglam)* introduces a region ϱ. The body, e, of the region abstraction is type checked with the constraint set C. The region abstraction passes those constraints to the context that do not mention ϱ, *i.e.* $C_{\downarrow\varrho}$. The binding time of ϱ is determined by the constraints that only mention ϱ, *i.e.* $C^{\uparrow\varrho}$. As we

(C-Var)
$$\frac{TE(x) = \mu \quad C \rhd \emptyset}{TE, C \vdash_c x : \mu, \emptyset}$$

(C-Const)
$$\frac{C \vdash_{wft} (\mathtt{int}, \rho)}{TE, C \vdash_c c\ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}}$$

(C-Lam)
$$\frac{\begin{array}{c} TE + \{x \mapsto \mu_1\}, C \vdash_c e : \mu_2, \varphi \\ C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho) \quad \varphi \subseteq \varphi' \end{array}}{TE, C \vdash_c \lambda x.\, e\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\} \cup \varphi}$$

(C-Letrec)
$$\frac{\begin{array}{c} P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\, .C^{\uparrow P} \Rightarrow \tau \quad \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \lambda x.\, e_1\ \mathtt{at}\ \rho : (\tau, \rho), \varphi' \\ \sigma = \forall\, \vec{\alpha}\, .\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c e_2 : \mu, \varphi \end{array}}{TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda x.\, e_1\ \mathtt{at}\ \rho\ \mathtt{in}\ e_2 : \mu, \varphi \cup \varphi'}$$

(C-App)
$$\frac{TE, C \vdash_c e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \quad TE, C \vdash_c e_2 : \mu_1, \varphi_2}{TE, C \vdash_c e_1 \mathbin{@} e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

(C-Let)
$$\frac{TE, C \vdash_c e_1 : \mu', \varphi' \quad TE + \{x \mapsto \mu'\}, C \vdash_c e_2 : \mu, \varphi}{TE, C \vdash_c \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mu, \varphi \cup \varphi'}$$

(C-Regapp)
$$\frac{\begin{array}{c} TE(f) = (\sigma, \rho) \quad \sigma = \forall\, \vec{\alpha}\, .\forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\, .C'' \Rightarrow \tau \\ (C', \tau') \prec \sigma\ \ via\ \ (S_t, S_e, \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}) \\ C \rhd C' \cup \{\rho' \leq \rho, \rho \leq \rho', \rho' \leq \rho'_1, \ldots, \rho' \leq \rho'_n\} \end{array}}{TE, C \vdash_c f\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

(C-Reglam)
$$\frac{TE, C \vdash_c e : \mu, \varphi \quad \varrho \notin frv\,(TE, \mu)}{TE, C_{\downarrow \varrho} \vdash_c \mathtt{new}\ \varrho : C^{\uparrow \varrho}\, .e : \mu, \varphi \setminus \{\varrho\}}$$

(C-Effect)
$$\frac{TE, C \vdash_c e : \mu, \varphi \quad \epsilon \notin fev\,(TE, \mu)}{TE, C \vdash_c e : \mu, \varphi \setminus \{\epsilon\}}$$

(C-Lift)
$$\frac{TE, C \vdash_c e : (\mathtt{int}, \rho_1), \varphi \quad C \rhd \{\rho_1 \leq \rho_2\}}{TE, C \vdash_c \mathtt{lift}\,[\rho_1, \rho_2]\, e : (\mathtt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}}$$

Figure 6.2: Static semantics of the λ_C^{region}-calculus

shall see later, this is due to the fact that during specialization, all other regions occurring in $C^{\uparrow\varrho}$ will have a known binding time. Similar reasoning applies to rule *(C-Letrec)*: the body of f is type checked with a constraint set C, but the type scheme of f only needs the constraint set $C^{\uparrow P}$. The `letrec`-body and the context can be type-checked with the constraints in C *not* mentioning the region variables in P, *i.e.* $C_{\downarrow P}$.

Rule *(C-Regapp)* types a region application. After instantiation, it shifts the constraint set of the type scheme into the constraint set of the context. Moreover, it adds constraints guaranteeing that the application has the same binding time as the defining `letrec`. Additionally, if the defining `letrec`-bound function is dynamic, it cannot pass static arguments.

Rule *(C-Lam)* slightly deviates from type rule *(TT-Lam)* of the λ_{tt}^{region}-calculus in its treatment of the effect. It types a lambda expression and introduces well-formedness constraints, which we need to obtain sound specialization. As usual, the latent effect φ' is a superset of the effect φ of the body. In the $\vdash_t^t$ typing judgment for the λ_{tt}^{region}-calculus, the effect of a lambda expression is only $\{\rho\}$, indicating that a closure is allocated in the region ρ. However, since the constraint analysis is a static semantics for program *specialization*, the effect $\{\rho\}$ is not sufficient. Indeed, if the lambda turns out to be dynamic (where we stress that the actual binding time is only known at specialization time), then the body e is further reduced. Hence, the effect of the lambda's body must be included to ensure soundness of the analysis, otherwise, a dangling pointer may be dereferenced.

The rule *(C-Lam)* from Figure 6.2 avoids this problem at the cost of delayed region deallocation. This approach is conservative because discharging the arrow effect in *(C-Lam)* is not necessary if the lambda is static. Indeed, if the static lambda contains dead code it will not immediately be deallocated, even though it is safe to do so. Instead, deallocation will only take place whenever the static lambda itself is not used anymore. However, if the static lambda does not contain dead code, the body's effect will be discharged by rule *(C-App)*, making the practical difference of the change rather small. Also, late deallocation is not very critical in a program specialization context. The main reason why early deallocation is still desirable is to keep the constraint sets small as can be seen from rule *(C-Reglam)*.

Note that we cannot do away with region abstraction altogether. First, because of polymorphic recursion, regions may be introduced in every recursive function unfolding, which is only decided at runtime. Second, having region abstractions in the λ_C^{region}-calculus keeps a close relationship between the constraint analysis and the $\vdash_t^t$ typing judgment of Section 2.3.3.

The attentive reader might wonder if the change in rule *(C-Lam)* invalidates known region reconstruction algorithms. However, it turns out that only minimal changes are required in the algorithms by Birkedal and Tofte [11, 146].

In this thesis, we base the constraint analysis on a region derivation, as produced by a region inference algorithm. Hence, all λ_C^{region}-Terms e have a fixed region type with place μ indicated by the notation e^{μ}. However, to avoid clutter, we usually leave out the annotation unless it is required by the context.

6.2.4 Properties of the Constraint Analysis

Constraints and the constraint analysis have many useful properties, usually proven by straightforward induction arguments.

Lemma 6.2.1 *Let* $P \subseteq \mathsf{Placeholder}$.

1. $C^{\uparrow P} \cup C_{\downarrow P} = C$ *and* $C^{\uparrow P} \cap C_{\downarrow P} = \emptyset$.
2. *If* C *is transitive closed, then so is* $C_{\downarrow P}$.

Lemma 6.2.2 *Whenever* $P_1, P_2 \subseteq \mathsf{Placeholder}$, *we have* $(C_{\downarrow P_1})_{\downarrow P_2} = (C_{\downarrow P_2})_{\downarrow P_1}$

Lemma 6.2.3 *Suppose* $C \vdash_{wft} \mu$, *then* C *is transitive closed.*

Lemma 6.2.4 *Suppose* $TE, C \vdash_c e : \mu, \varphi$, *then* C *is transitive closed.*

Proof. Simple induction on the type derivation, using Lemma 6.2.1.2 and Lemma 6.2.3. □

It is always safe to add constraints to a derivation after closing them under transitivity:

Lemma 6.2.5 *Suppose* $TE, C \vdash_c e : \mu, \varphi$ *and* $C' \rhd C$ *then* $TE, C' \vdash_c e : \mu, \varphi$.

A constraint type judgment exhibits a type substitution lemma.

Lemma 6.2.6 *If* $TE, C \vdash_c e : \mu, \varphi$ *then, for all substitutions* S_s, *we also have the judgment* $S_s(TE), S_s(C) \vdash_c S_s(e) : S_s(\mu), S_s(\varphi)$.

Proof. By induction on the derivation of $TE, C \vdash_c e : \mu, \varphi$, similar to Lemma 3.3.3 on page 34. □

We also have a weakening lemma:

Lemma 6.2.7 *If* $TE + \{f \mapsto (\sigma, \rho)\}, C \vdash_c e : \mu, \varphi$, *and* $\sigma \prec \sigma'$, *then* $TE + \{f \mapsto (\sigma', \rho)\}, C \vdash_c e : \mu, \varphi$.

Proof. A straightforward induction on $TE + \{f \mapsto (\sigma, \rho)\}, C \vdash_c e : \mu, \varphi$. Simple extension of Lemma 3.3.4 on page 34 to the λ_C^{region}-calculus. □

Lemma 6.2.8 *If* $TE, C \vdash_c e : \mu, \varphi$ *and* $C \vdash_{wft} TE$, *then* $C \vdash_{wft} \mu$.

Proof. The proof follows by straightforward induction on the type derivation. □

Closing a constraint set C does not affect its solution.

Lemma 6.2.9 *If* $\delta \models C$, *then* $\delta \models \widetilde{C}$.

A region annotation solving a transitive closed set can always be extended with one new region variable, without sacrifying solvability:

Lemma 6.2.10 *Suppose C is transitive closed and $\delta \models C_{\downarrow\varrho}$ with $\varrho \notin Dom(\delta)$. Then, there always exists a binding time β such that $\delta + \{\varrho \mapsto \beta\} \models C^{\uparrow\varrho}$.*

Proof. By contradiction: assume there is no such β for which $\delta + \{\varrho \mapsto \beta\} \models C^{\uparrow\varrho}$. Then, since every constraint in $C^{\uparrow\varrho}$ mentions ϱ, there must be a pair of constraints $\{\rho_1 \leq \varrho, \varrho \leq \rho_2\} \subseteq C^{\uparrow\varrho}$ where $\varrho \notin \{\rho_1, \rho_2\}$ such that $\delta(\rho_1) = \mathsf{d}$ and $\delta(\rho_2) = \mathsf{s}$. However, since we have that C is transitive closed, it holds that $\rho_1 \leq \rho_2 \in C$ and hence, $\rho_1 \leq \rho_2 \in C_{\downarrow\varrho}$. By assumption we have that $\delta \models C_{\downarrow\varrho}$, so $\delta(\rho_1) = \mathsf{d} \leq \mathsf{s} = \delta(\rho_2)$, which leads to a contradiction. □

In addition, such an extended solution works for the whole constraint set:

Lemma 6.2.11 *If $\delta \models C_{\downarrow\varrho}$ and $\delta' = \delta + \{\varrho \mapsto \beta\}$ so that $\delta' \models C^{\uparrow\varrho}$, then $\delta' \models C$.*

Proof. By Lemma 6.2.1.1, we have that $C = C^{\uparrow\varrho} \cup C_{\downarrow\varrho}$, so for any $\rho_1 \leq \rho_2 \in C$, we either have $\rho_1 \leq \rho_2 \in C^{\uparrow\varrho}$, which trivially implies $\delta \models C^{\uparrow\varrho}$ by assumption, or $\rho_1 \leq \rho_2 \in C_{\downarrow\varrho}$. Since $\delta \models C_{\downarrow\varrho}$ and $\varrho \notin frv(C_{\downarrow\varrho})$, by definition, we obviously have that $\delta' \models C_{\downarrow\varrho}$. □

The constraint analysis of Figure 6.2 induces the $\vdash_t$ typing judgment for the λ_{tt}^{region}-calculus by *erasing* the constraint judgments. That is, for every judgment $TE, C \vdash_c e : \mu, \varphi$ in the $\vdash_c$ typing judgment, there is a type judgment $|TE| \vdash_t |e| : \mu, \varphi$ in the $\vdash_t$ typing judgment. The rules for the $\vdash_t$ typing judgment are identical to those found in Figure 2.6 on page 27, except for type rule *(TT-Lam)*, which is replaced by the following rule:

$$\textit{(T-Lam)} \qquad \frac{TE + \{x \mapsto \mu_1\} \vdash_t e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_t \lambda\, x.\, e\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\} \cup \varphi}$$

The $\vdash_t$ typing judgment is a conservative approximation of the $\vdash_t^t$ typing judgment:

Proposition 6.2.12 *Suppose $TE \vdash_t e : \mu, \varphi$. Then $TE \vdash_t^t e : \mu, \varphi'$, where $\varphi' \subseteq \varphi$.*

Proof. All cases are straightforward, except for rule *(T-Lam)*. So, suppose $TE \vdash_t \lambda\, x.\, e\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu_2, \rho), \{\rho\} \cup \varphi$ and hence, we have that $TE + \{x \mapsto \mu_1\} \vdash_t e : \mu_2, \varphi$ with $\varphi \subseteq \varphi''$. Using the induction hypothesis, we have that $TE + \{x \mapsto \mu_1\} \vdash_t^t e : \mu_2, \varphi'$ where $\varphi' \subseteq \varphi$. Therefore, we also have $\varphi' \subseteq \varphi''$ and we can apply rule *(TT-Lam)* to obtain $TE \vdash_t^t \lambda\, x.\, e\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi''} \mu_2, \rho), \{\rho\}$. Obviously it holds that $\{\rho\} \subseteq \{\rho\} \cup \varphi$, proving the case. □

6.2.5 A Constraint Completion

An offline partial evaluator requires a *constraint completion*[1] to specialize successfully. Starting from a region derivation $\{\ \} \vdash_t e : \mu, \varphi$, the constraint completion provides a λ_C^{region}-term e whose erasure is identical to the λ_{tt}^{region}-Term e and which is typeable in the constraint analysis of Figure 6.2 with a solvable constraint set. The following definition makes this precise:

Definition 6.2.2 *Suppose* $\{\ \} \vdash_t e' : \mu, \varphi$ *for a* λ_{tt}^{region}-Term e' *with* λ_{tt}^{region}-TypeWPlace μ *and* λ_{tt}^{region}-Effect φ.

A λ_C^{region}-Term e *with initial constraint set* $C \in$ ConstraintSet *and region annotation* δ_c *is a* constraint completion *of* e' *iff* $|e^\mu| = e'$*, the judgment* $\{\ \}, C \vdash_c e^\mu : \mu, \varphi$ *holds,* $\delta_c \models C$ *and for all* $\rho \in \varphi \cup frv\,(C, \mu)$ *it holds that* $\delta_c(\rho) = \mathsf{d}$.

A constraint completion always exists if region inference succeeds.

Theorem 6.2.13 *Suppose* $\{\ \} \vdash_t e' : \mu, \varphi$*. Then there exists a constraint completion* e^μ *with region annotation* δ_c.

Proof. The construction is a follows:

- take the constraint $C = \left\{\varrho \leq \varrho' \mid \text{ for all region variables } \varrho \text{ and } \varrho' \text{ in the program}\right\}$
- translate every $\texttt{new}\ \varrho.\ e$ in the program into $\texttt{new}\ \varrho : C^{\uparrow \varrho}.e'$, where e' is the translation of e.
- from this, it is easy to see that the program is constraint-typeable with the rules from Figure 6.2 because we can enlarge the constraint sets in typing judgments arbitrarily with Lemma 6.2.5.
- finally, we take δ_c such that it maps all region variables in $\varphi \cup frv\,(C, \mu)$ onto d.

□

In partial evaluation, it is important to have an idea on the *quality* of the resulting binding times. In our framework we actually want to know about the quality of the constraints since they determine the binding time during specialization. There are two factors influencing this.

The first factor is the accuracy of the underlying region inference. If only one region is allocated for the entire program, no specialization will take place whatsoever as this region has to be dynamic. Generally, one would like to separate values in as many regions as possible since every region can have a different binding time. Fortunately, region separation coincides with the region inference goals in a memory management setting, where more regions allow more fine-grained garbage collection [150]. However, it is still an open question if every Hindley-Milner typeable expression has a principal (or most general) region type. As a consequence, we cannot guarantee a principal binding-time type, unlike Dussart et al. [36].

[1] In partial evaluation literature, this is usually called a *binding-time completion* [70].

The second binding-time quality factor is the number of constraints in the set C. Unfortunately, the constraint completion as constructed in the proof of Theorem 6.2.13 is not very interesting since it the only annotations that satisfy the constraints map all variables on d, *i.e.* no specialization will take place. A more interesting question would be whether it is possible to construct a unique minimal constraint completion, given an explicitely region-typed program. And if so, whether it can be done in a reasonable time-complexity. Although we conjecture that these two questions have a positive answer, we do not pursue them in this thesis and leave it for future work.

Finally, we note that in our specialization model, constraints have to be solved at runtime. The total number of constraints that must be solved at each region allocation is bounded by the number of regions in the program since the set C in the expression $\mathtt{new}\,\varrho : C\,.e$ only contains constraints that mention ϱ. In other words, we have at most $O(n)$ constraints to solve at each region allocation during specialization. We conjecture that this upper bound is pathological in practice though.

6.3 Specialization

In this section, we present a region-based partial evaluator for the λ_C^{region}-calculus. As always, we formulate it as a small-step reduction semantics and hence, we need to extend the λ_C^{region}-calculus with several computational intermediate constructs. Since the specializer actually calculates binding times, we will extend the λ_C^{region}-calculus to a *two-level* λ_2^{region}-calculus [111] to make intermediate terms explicit in the syntax.

6.3.1 The Two-Level λ_2^{region}-calculus

The λ_2^{region}-calculus is a conservative extension of the λ_C^{region}-calculus and defined by the grammar in Figure 6.3. Two-level λ_2^{region}-Terms introduce a few intermediate constructs, value pointers $V \in \mathsf{Value}$ and *residual terms* $R \in \mathsf{ResidualTerm}$.

Next to region variables, λ_2^{region}-Placeholders may stand for deallocated region variables, $\varrho^\bullet \in \mathsf{DeadRegionVar}$, as well as two unique global binding-time regions, where we assume that DeadRegionVar, BTRegion and RegionVar are mutually disjoint. In contrast with the approach of the previous chapters, a deallocated region is taken from the set DeadRegionVar, not just the dead region $\bullet$. This enhancement is required since we are dealing with constraints on region placeholders. Consistency of a constraint set implies that the set is transitive closed. To maintain this invariant, substituting a region ϱ by the dead region $\bullet$ introduces new constraints that may not be intended since the variable $\bullet$ does not "know" what region it was substituted for. We avoid this problem, by remembering the deallocated region. Note that we could have introduced this extension in earlier chapters. However, in a non-specialization context, this enhancement is superfluous and introduces clutter.

Region placeholders can also be one of the *virtual* binding-time region variables, $\mathtt{s}$ or $\mathtt{d}$. They are a technical device for proving soundness of the constraint-analysis as we will explain below. The $\mathit{Clean}\,(\cdot)$ predicate retains its definition from Section 3.2.3, *i.e.* $\mathit{Clean}\,(A)$ is true iff $\mathit{frv}\,(A) \subseteq \mathsf{RegionVar}$ and false otherwise.

$$
\begin{array}{rl}
\mathsf{Term} \ni E ::= & V \mid R \mid c\ \mathtt{at}\ \rho \mid \lambda x.\, E\ \mathtt{at}\ \rho \mid E\ @\ E \mid \mathtt{lift}\ [\rho_1, \rho_2]\ E \mid \mathtt{new}\ \varrho : C\ .E \mid \\
& f\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, E \rangle_\rho\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \rho \mid \\
& \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, E\ \mathtt{at}\ \rho\ \mathtt{in}\ E \mid \mathtt{let}\ x = E\ \mathtt{in}\ E \mid \\
& \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, E \rangle_\rho\ \mathtt{in}\ E \mid \mathtt{let}^{\mathsf{d}}\ x = R\ \mathtt{in}\ E \mid \\
& \mathtt{letrec}^{\mathsf{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x.\, R\ \mathtt{at}\ \rho\ \mathtt{in}\ E \mid \mathtt{new}^{\beta}\ \varrho . E
\end{array}
$$

$$
\mathsf{Value} \ni V ::= \langle c \rangle_\rho \mid \langle \lambda x.\, E \rangle_\rho
$$

$$
\begin{array}{rl}
\mathsf{ResidualTerm} \ni R ::= & x \mid c\ \underline{\mathtt{at}}\ \varrho \mid \underline{\lambda} x.\, R\ \underline{\mathtt{at}}\ \varrho \mid R\ \underline{@}\ R \mid \underline{\mathtt{lift}}\ [\varrho_1, \varrho_2]\ R \mid \underline{\mathtt{new}}\ \varrho .\, R \mid \\
& \underline{\mathtt{let}}\ x = R\ \underline{\mathtt{in}}\ R \mid f\ [\varrho_1, \ldots, \varrho_n]\ \underline{\mathtt{at}}\ \varrho \mid \\
& \underline{\mathtt{letrec}}\ f = \underline{\Lambda}\, \varrho_1, \ldots, \varrho_n . \underline{\lambda} x.\, R\ \underline{\mathtt{at}}\ \varrho\ \underline{\mathtt{in}}\ R
\end{array}
$$

$$
\mathsf{DeadRegionVar} = \{\varrho^\bullet \mid \varrho \in \mathsf{RegionVar}\} \qquad \mathsf{BTRegion} = \{\mathtt{s}, \mathtt{d}\}
$$
$$
\mathsf{Placeholder} = \mathsf{RegionVar} \cup \mathsf{DeadRegionVar} \cup \mathsf{BTRegion}
$$

Figure 6.3: Syntax of the λ_2^{region}-calculus

As with previous intermediate region constructs, a value $V \in \lambda_2^{region}$-Value is a *pointer* to a constant or a lambda abstraction. For example, the term $\langle c \rangle_\varrho$ denotes a pointer to a constant c allocated in region ϱ, whereas the expression $c\ \mathtt{at}\ \varrho$ performs the allocation of the constant. Similarly, we have a region closure pointer of the form $\langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda x . E \rangle_\rho$. However, unlike a values, a closure pointer cannot occur in isolation within a two-level expression. It is either bound to a variable in a `letrec`-expression or applied to some region arguments.

There are four intermediate constructs: The dynamic $\mathtt{let}^{\mathsf{d}}$ and $\mathtt{letrec}^{\mathsf{d}}$ terms are very much like `let` and `letrec` expressions, except that they require their bound expressions to be residual. The binding-time annotated region abstraction $\mathtt{new}^{\beta}\ \varrho . E$ is the result of reducing a constraint qualified region abstraction $\mathtt{new}\ \varrho : C\ .E$. The binding time β is calculated from the constraint set C and fixed for the rest of the reduction.

The computational two-level calculus also defines residual terms. They are identical to the terms of the λ_{tt}^{region}-calculus, except that the keywords are underlined. Residual terms are the result of specialization of a constraint completion.

The syntactic category λ_2^{region}-Term contains expressions that never occur in the *result* of a specialization. We make the notion of result more accurate by defining two sub-sets of λ_2^{region}-Term, produced by the grammar in Figure 6.4.

A specialization *answer*, Q, cannot be further reduced by the transition semantics. It is either a residual term R or a *static answer*, Q^{s}. The latter is a value V embedded in a dynamic context: the empty context, a dynamic region abstraction, a $\mathtt{let}^{\mathsf{d}}$-expression or a $\mathtt{letrec}^{\mathsf{d}}$-expression. A static answer that is not a value admits some well-known transformations that improve the binding times, as we will explain below.

We overload the erasure function for λ_C^{region}-expressions to λ_2^{region}-expressions, *i.e* $|\cdot| \in \lambda_2^{region}\text{-Term} \rightarrow \lambda_s^{region}\text{-Term}$. It produces a term in the λ_s^{region}-calculus by stripping constraints sets, removing underlines from residual terms as well as the binding-time

$$\begin{aligned} \mathsf{StaticAnswer} \ni \mathrm{Q}^{\mathsf{s}} ::= \quad & V \mid \mathtt{new}^{\mathsf{d}}\, \varrho.\mathrm{Q}^{\mathsf{s}} \mid \mathtt{let}^{\mathsf{d}}\, x = R\, \mathtt{in}\, \mathrm{Q}^{\mathsf{s}} \mid \\ & \mathtt{letrec}^{\mathsf{d}}\, f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, R\, \mathtt{at}\, \rho\, \mathtt{in}\, \mathrm{Q}^{\mathsf{s}} \\ \mathsf{Answer} \ni Q ::= \quad & \mathrm{Q}^{\mathsf{s}} \mid R \end{aligned}$$

Figure 6.4: Specialization Answers

annotations from region binders and the `let` and `letrec` constructs. Finally, it also maps all placeholders from DeadRegionVar and BTRegion onto the dead region, $\bullet \in \lambda_s^{region}$-Placeholder.

Not all region annotations $\delta \in \mathsf{BtAnn}$ are useful for specialization. For example, region deallocation only takes place at specialization time and we expect the static region $\mathtt{s}$ to be mapped to the static binding time s and similarly, the dynamic region $\mathtt{d}$ should have binding time d too. Therefore assume an initial region annotation $\delta_{\text{init}} \in \mathsf{BtAnn}$ such that $\delta_{\text{init}} = \{\varrho^{\bullet} \mapsto \mathsf{s} \mid \varrho \in \mathsf{RegionVar}\} + \{\mathtt{s} \mapsto \mathsf{s}, \mathtt{d} \mapsto \mathsf{d}\}$.

6.3.2 A Specialization Semantics

The small-step transition semantics is defined as a relation $\rightsquigarrow \in \mathcal{P}(\mathsf{BtAnn} \times \mathsf{Term} \times \mathsf{BtAnn} \times \mathsf{Term})$. It defines a reduction step from a region annotation and term into a new region annotation and term. The region annotation δ records the binding times for regions as they are calculated during specialization. Figure 6.5, Figure 6.6 and Figure 6.7 specify the individual transition rules for $\rightsquigarrow$.

Rules (6.1), (6.2), and (6.3) allocate static constants, lambdas, and region-lambda abstractions respectively. Rule (6.4) deallocates a static region after specializing the body. The `lift` in rule (6.5) moves a constant between two static regions and rule (6.6) specifies beta value reduction for static lambdas. Similarly, a `let`-expression is beta-reduced by rule (6.7), where the `let`-bound variable is already reduced to a value. Static region application is defined in rule (6.8), and rule (6.9) unfolds `letrec`-bound functions. Finally, rules (6.10) to (6.14) specify a left-to-right call-by-value semantics for static parts of the $\lambda_{\varrho}^{region}$-calculus

The binding time of a region is calculated by rule (6.15) in Figure 6.6. The rule extends the region annotation, δ, such that the constraint set C is solvable. The extra condition $\varrho \notin \mathit{Dom}\,(\delta)$ can trivially be satisfied by α-renaming ϱ in C and E if required.

Rule (6.16) implements a first-order *binding-time coercion*: it transforms a static, specialization-time, constant into a dynamic constant by moving it from a static region into a dynamic region. Such binding-time coercions are necessary to obtain good results [70, 82]. Embedding every expression of type `int` in a `lift`-expression yields optimal results [70]. This is the additional functionality of `lift` alluded to in Section 6.2.1.

The rules (6.17) through (6.24) generate residual code. This is only allowed if all sub-terms are already residual. Note that in rule (6.21), we do not reduce the dynamic lift. This is because it copies a value from one region to the other, but we can only

$$\delta, c \ \texttt{at}\ \varrho \rightsquigarrow \delta, \langle c \rangle_{\varrho} \quad \textit{if}\ \delta(\varrho) = \texttt{s} \tag{6.1}$$

$$\delta, \lambda\, x.\ E \ \texttt{at}\ \varrho \rightsquigarrow \delta, \langle \lambda\, x.\ E \rangle_{\varrho} \quad \textit{if}\ \delta(\varrho) = \texttt{s} \tag{6.2}$$

$$\begin{array}{c} \delta, \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ E_1 \ \texttt{at}\ \varrho\ \texttt{in}\ E_2 \rightsquigarrow \\ \delta, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ E_1 \rangle_{\varrho}\ \texttt{in}\ E_2 \end{array} \quad \textit{if}\ \delta(\varrho) = \texttt{s} \tag{6.3}$$

$$\delta, \texttt{new}^{\texttt{s}}\ \varrho . Q \rightsquigarrow \delta, Q[\varrho \mapsto \varrho^{\bullet}] \tag{6.4}$$

$$\delta, \texttt{lift}\ [\varrho_1, \varrho_2]\ \langle c \rangle_{\varrho_1} \rightsquigarrow \delta, \langle c \rangle_{\varrho_2} \quad \textit{if}\ \delta(\varrho_2) = \texttt{s} \tag{6.5}$$

$$\delta, \langle \lambda\, x.\ E \rangle_{\varrho}\ @\ Q \rightsquigarrow \delta, \texttt{let}\ x = Q\ \texttt{in}\ E \tag{6.6}$$

$$\delta, \texttt{let}\ x = V\ \texttt{in}\ E \rightsquigarrow \delta, E[x \mapsto V] \tag{6.7}$$

$$\delta, \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ E \rangle_{\varrho}\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \varrho_0 \rightsquigarrow \delta, \langle \lambda\, x.\ E[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n] \rangle_{\varrho_0} \tag{6.8}$$

$$\begin{array}{c} \delta, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ E_1 \rangle_{\rho}\ \texttt{in}\ E_2 \rightsquigarrow \\ \delta, E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\ E_1 \rangle_{\rho}\ \texttt{in}\ E_1 \rangle_{\rho}] \end{array} \tag{6.9}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, E\ @\ E'' \rightsquigarrow \delta', E'\ @\ E''} \tag{6.10}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, Q\ @\ E \rightsquigarrow \delta', Q\ @\ E'} \tag{6.11}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{lift}\ [\rho_1, \rho_2]\ E \rightsquigarrow \delta', \texttt{lift}\ [\rho_1, \rho_2]\ E'} \tag{6.12}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{let}\ x = E\ \texttt{in}\ E'' \rightsquigarrow \delta', \texttt{let}\ x = E'\ \texttt{in}\ E''} \tag{6.13}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{new}^{\texttt{s}}\ \varrho . E \rightsquigarrow \delta', \texttt{new}^{\texttt{s}}\ \varrho . E'} \tag{6.14}$$

Figure 6.5: Dynamic semantics of the $\lambda_{\varrho}^{region}$-calculus - part I

$$\delta, \texttt{new}\ \varrho : C\ .E \rightsquigarrow \delta + \{\varrho \mapsto \beta\}, \texttt{new}^{\beta}\ \varrho.E \quad \textit{if } \varrho \notin \textit{Dom}\,(\delta) \textit{ and } \delta + \{\varrho \mapsto \beta\} \models C \tag{6.15}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\,\langle c \rangle_{\varrho_1} \rightsquigarrow \delta, c\ \texttt{at}\ \varrho_2 \quad \textit{if } \delta(\varrho_2) = \texttt{d} \tag{6.16}$$

$$\delta, c\ \texttt{at}\ \varrho \rightsquigarrow \delta, c\ \underline{\texttt{at}}\ \varrho \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{6.17}$$

$$\delta, \lambda\, x.\ R\ \texttt{at}\ \varrho \rightsquigarrow \delta, \underline{\lambda}\, x.\ R\ \underline{\texttt{at}}\ \varrho \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{6.18}$$

$$\delta, \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R_1\ \texttt{at}\ \varrho\ \texttt{in}\ R_2 \rightsquigarrow \delta, \underline{\texttt{letrec}}\ f = \underline{\Lambda}\, \varrho_1, \ldots, \varrho_n.\underline{\lambda}\, x.\ R_1\ \underline{\texttt{at}}\ \varrho\ \underline{\texttt{in}}\ R_2 \tag{6.19}$$

$$\delta, \texttt{new}^{\texttt{d}}\ \varrho.R \rightsquigarrow \delta, \underline{\texttt{new}}\ \varrho.\ R \tag{6.20}$$

$$\delta, \texttt{lift}\,[\varrho_1, \varrho_2]\ R \rightsquigarrow \delta, \underline{\texttt{lift}}\,[\varrho_1, \varrho_2]\ R \tag{6.21}$$

$$\delta, R_1 \mathbin{@} R_2 \rightsquigarrow \delta, R_1 \mathbin{\underline{@}} R_2 \tag{6.22}$$

$$\delta, \texttt{let}^{\texttt{d}}\ x = R_1\ \texttt{in}\ R_2 \rightsquigarrow \delta, \underline{\texttt{let}}\ x = R_1\ \underline{\texttt{in}}\ R_2 \tag{6.23}$$

$$\delta, f\ [\varrho_1, \ldots, \varrho_n]\ \texttt{at}\ \varrho \rightsquigarrow \delta, f\ [\varrho_1, \ldots, \varrho_n]\ \underline{\texttt{at}}\ \varrho \tag{6.24}$$

$$\delta, \texttt{let}\ x = R\ \texttt{in}\ E \rightsquigarrow \delta, \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E \tag{6.25}$$

$$\delta, \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E \rightsquigarrow \delta, \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{6.26}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \lambda\, x.\ E\ \texttt{at}\ \varrho \rightsquigarrow \delta', \lambda\, x.\ E'\ \texttt{at}\ \varrho} \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{6.27}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E\ \texttt{at}\ \varrho\ \texttt{in}\ E'' \rightsquigarrow \delta', \texttt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ E'\ \texttt{at}\ \varrho\ \texttt{in}\ E''} \quad \textit{if } \delta(\varrho) = \texttt{d} \tag{6.28}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{new}^{\texttt{d}}\ \varrho.E \rightsquigarrow \delta', \texttt{new}^{\texttt{d}}\ \varrho.E'} \tag{6.29}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E \rightsquigarrow \delta', \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E'} \tag{6.30}$$

$$\frac{\delta, E \rightsquigarrow \delta', E'}{\delta, \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E \rightsquigarrow \delta', \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \varrho\ \texttt{in}\ E'} \tag{6.31}$$

Figure 6.6: Dynamic semantics of the λ_2^{region}-calculus - part II

$$\delta, \mathtt{lift}\,[\rho_1,\rho_2]\,(\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{lift}\,[\rho_1,\rho_2]\,\mathrm{Q}^{\mathsf{s}}) \quad \textit{if } \varrho \notin \{\rho_1,\rho_2\} \tag{6.32}$$

$$\delta, \mathtt{let}\; x = (\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}})\;\mathtt{in}\;E^{\mu} \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{let}\; x = \mathrm{Q}^{\mathsf{s}}\;\mathtt{in}\;E^{\mu}) \quad \textit{if } \varrho \notin \mathit{frv}\,(\mu) \tag{6.33}$$

$$\delta, (\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}_1^{\mathsf{s}}) \mathbin{@} Q_2^{\mu} \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathrm{Q}_1^{\mathsf{s}} \mathbin{@} Q_2^{\mu}) \quad \textit{if } \varrho \notin \mathit{frv}\,(\mu) \tag{6.34}$$

$$\delta, \mathtt{lift}\,[\rho_1,\rho_2]\,(\mathtt{let}^{\mathsf{d}}\; x = R\;\mathtt{in}\;\mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{let}^{\mathsf{d}}\; x = R\;\mathtt{in}\;(\mathtt{lift}\,[\rho_1,\rho_2]\,\mathrm{Q}^{\mathsf{s}}) \tag{6.35}$$

$$\delta, \mathtt{let}\; y = (\mathtt{let}^{\mathsf{d}}\; x = R\;\mathtt{in}\;\mathrm{Q}^{\mathsf{s}})\;\mathtt{in}\;E \rightsquigarrow \delta, \mathtt{let}^{\mathsf{d}}\; x = R\;\mathtt{in}\;(\mathtt{let}\; y = \mathrm{Q}^{\mathsf{s}}\;\mathtt{in}\;E) \quad \textit{if } x \notin \mathit{fv}\,(E) \tag{6.36}$$

$$\delta, (\mathtt{let}^{\mathsf{d}}\; x = R\;\mathtt{in}\;\mathrm{Q}_1^{\mathsf{s}}) \mathbin{@} Q_2 \rightsquigarrow \delta, \mathtt{let}^{\mathsf{d}}\; x = R\;\mathtt{in}\;(\mathrm{Q}_1^{\mathsf{s}} \mathbin{@} Q_2) \quad \textit{if } x \notin \mathit{fv}\,(Q_2) \tag{6.37}$$

$$\delta, \mathtt{lift}\,[\rho_1,\rho_2]\,(\mathtt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\; R\;\mathtt{at}\;\varrho\;\mathtt{in}\;\mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\; R\;\mathtt{at}\;\varrho\;\mathtt{in}\;(\mathtt{lift}\,[\rho_1,\rho_2]\,\mathrm{Q}^{\mathsf{s}}) \tag{6.38}$$

$$\delta, \mathtt{let}\; x = (\mathtt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\; R\;\mathtt{at}\;\varrho\;\mathtt{in}\;\mathrm{Q}^{\mathsf{s}})\;\mathtt{in}\;E \rightsquigarrow \delta, \mathtt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\; R\;\mathtt{at}\;\varrho\;\mathtt{in}\;(\mathtt{let}\; x = \mathrm{Q}^{\mathsf{s}}\;\mathtt{in}\;E) \quad \textit{if } f \notin \mathit{fv}\,(E) \tag{6.39}$$

$$\delta, (\mathtt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\; R\;\mathtt{at}\;\varrho\;\mathtt{in}\;\mathrm{Q}_1^{\mathsf{s}}) \mathbin{@} Q_2 \rightsquigarrow \delta, \mathtt{letrec}^{\mathsf{d}}\; f = \Lambda\,\varrho_1,\ldots,\varrho_n.\lambda\, x.\; R\;\mathtt{at}\;\varrho\;\mathtt{in}\;(\mathrm{Q}_1^{\mathsf{s}} \mathbin{@} Q_2) \quad \textit{if } f \notin \mathit{fv}\,(Q_2) \tag{6.40}$$

Figure 6.7: Dynamic semantics of the λ_2^{region}-calculus - part III

do that if R was a value. The rules (6.27) and (6.28) are context rules that specialize under dynamic lambdas. These rules are the main deviation from a standard semantics and increase the potential for non-termination. Mechanisms to avoid this undesirable behavior (e.g. program point specialization or memoization [138]) are not considered in this thesis and left for future work. Rules (6.25) and (6.26) create the $\mathtt{let}^{\mathsf{d}}$ and $\mathtt{letrec}^{\mathsf{d}}$ intermediate expressions and the context rules (6.29), (6.30) and (6.31) specialize inside static answers.

Recall that static answers Q^{s} are static values embedded in a dynamic context. If a static answer occurs in a static context, a specializer can bring the static context closer to the embedded static value, allowing further specialization. These are well-known program transformations in partial evaluation for call-by-value calculi [60, 90, 91] and are implemented by the specialization rules in Figure 6.7.

In the λ_2^{region}-calculus there are five such static contexts, which can be read from the context rules (6.10) through (6.14). However, as can be seen from reduction rule (6.6), we recast the static context induced by rule (6.11) to that of (6.13). Moreover, the static context from rule (6.14) need not be transported since rule (6.4) deallocates a static region for any answer Q. Hence, the following three static contexts need to be transported to the static value: a `lift`-expression, which is handled by the rules (6.32), (6.35) and (6.38); a `let`-expression, handled by the rules (6.33), (6.36) and (6.39); and function application, covered by the rules (6.34), (6.37) and (6.40).

The side constraints on rules (6.32), (6.33) and (6.34) are necessary to make the specialization step sound: they guarantee the condition $\varrho \notin \mathit{frv}\,(TE, \mu)$ in type rule for region abstraction. These constraints can always be satisfied by α-renaming ϱ in the static answer $\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}}$. On an expression-variable level, α-renaming x and f in the rules

(6.36) and (6.37), and (6.39) and (6.40) respectively, can always satisfy the conditions. The reflexive transitive closure of the specialization relation $\leadsto$ is denoted by $\overset{*}{\leadsto}$.

To specialize a program $e' \in \lambda_{tt}^{region}$-Term, we need a constraint completion $e \in \lambda_{C}^{region}$-Term (this notion is defined in Section 6.2.5) with associated region annotation $\delta_c = \{\varrho_1 \mapsto \mathsf{d}, \ldots, \varrho_n \mapsto \mathsf{d}\}$. The actual input to the specializer is the region annotation $\delta_{\text{init}} + \delta_c$ with the λ_{C}^{region}-program e. As we shall prove in Section 6.4, our operational specialization semantics either does not terminate or produces a residual program R for such a completion, *i.e.* $\delta_{\text{init}} + \delta_c, e \overset{*}{\leadsto} \delta', R$. In Section 6.5, we also show that $|R| \in \lambda_{tt}^{region}$-Term is contextually equivalent to $e' \in \lambda_{tt}^{region}$-Term.

6.3.3 Examples

We will illustrate specialization by means of two small examples in an ML-typeable applied lambda calculus, extended with an `if`-construct. The semantics of `if` is standard, *i.e.* we evaluate the `else`-branch when its condition evaluates to 0 and to the `then`-branch otherwise.

Example 1

The first example illustrates a typical specialization process. Consider the following expression, where d is some dynamic input. Additionally, we have inserted `lift`-expressions around some expressions of type `int`, where we assume that the `lift`-construct in our applied lambda calculus coincides with the identity function:

$$\begin{array}{l}\lambda\, d.\, \texttt{letrec}\; apply = \lambda\, f.\, \lambda\, x.\, f \,@\, x\; \texttt{in} \\ \qquad \texttt{let}\;\; y = (apply \,@\, (\lambda\, n.\, 5)) \,@\, (\texttt{lift}\, 3) \\ \qquad \texttt{in let}\;\; z = (apply \,@\, (\lambda\, m.\, m)) \,@\, (\texttt{lift}\, d) \\ \qquad\quad \texttt{in if}\; z\; \texttt{then}\; (\texttt{lift}\, y)\; \texttt{else}\; (\texttt{lift}\, 0)\end{array}$$

Recall that, to obtain good specialization results [70], a preprocessor has to insert `lift` operations around *all* expressions of type `int`. However, to maintain readability, we insert just a few `lift`s for exemplification as only two `lift`-expressions turn out to be actual coercions. As a first step, we do region inference:

$$\begin{array}{l} e = \lambda\, d.\, (\texttt{new}\; \varrho_4, \varrho_5, \varrho_{18}. \\ \qquad \texttt{letrec}\; apply = \texttt{new}\; \varrho_6, \varrho_7, \varrho_8, \varrho_9.\, \lambda\, f.\, (\lambda\, x.\, (f \,@\, x)\; \texttt{at}\; \varrho_6)\; \texttt{at}\; \varrho_5\; \texttt{in} \\ \qquad\quad \texttt{let}\;\; y = \texttt{new}\; \varrho_{10}, \varrho_{11}, \varrho_{12}.\, (\texttt{new}\; \varrho_{13}.\, (apply\, [\varrho_{10}, \varrho_{11}, \varrho_4, \varrho_{12}]\; \texttt{at}\; \varrho_{13}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad @\, (\lambda\, n.\, (5\; \texttt{at}\; \varrho_4)\; \texttt{at}\; \varrho_{11})) \\ \qquad\qquad\qquad\qquad\qquad @\, (\texttt{new}\; \varrho_{14}.\, \texttt{lift}\, [\varrho_{14}, \varrho_{12}]\, (3\; \texttt{at}\; \varrho_{14})) \\ \qquad\quad \texttt{in let}\;\; z = \texttt{new}\; \varrho_{15}, \varrho_{16}.\, (\texttt{new}\; \varrho_{17}.\, (apply\, [\varrho_{15}, \varrho_{16}, \varrho_{18}, \varrho_{18}]\; \texttt{at}\; \varrho_{17}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad @\, (\lambda\, m.\, m\; \texttt{at}\; \varrho_{16})) \\ \qquad\qquad\qquad\qquad\qquad @\, (\texttt{lift}\, [\varrho_2, \varrho_{18}]\, d) \\ \qquad\qquad \texttt{in if}\; z\; \texttt{then}\; \texttt{lift}\, [\varrho_4, \varrho_3]\, y \\ \qquad\qquad\qquad \texttt{else}\; \texttt{lift}\, [\varrho_4, \varrho_3]\, (0\; \texttt{at}\; \varrho_4))\; \texttt{at}\; \varrho_1 \end{array}$$

The annotations are obtained by renaming intermediate code produced by the ML-kit [148]. Some abstracted regions do not appear in the body of the expression because region inference also abstracts regions that are only present in the type of an expression.

The type with place of e is $((\mathtt{int}, \varrho_2) \xrightarrow{\epsilon.\varphi} (\mathtt{int}, \varrho_3), \varrho_1)$ where the effect φ is $\{\varrho_2, \varrho_3\}$ and e's effect $\{\varrho_1\} \cup \varphi$. Next, we apply a constraint analysis to the region-annotated version of our program, which returns:

$$
\begin{array}{l}
e = \lambda\, d.\, (\mathtt{new}\, \varrho_4 : \{\varrho_4 \leq \varrho_3\}. \\
\quad \mathtt{new}\, \varrho_5 : \{\varrho_5 \leq \varrho_4, \varrho_5 \leq \varrho_2, \varrho_5 \leq \varrho_3\}. \\
\quad\ \mathtt{new}\, \varrho_{18} : \{\varrho_2 \leq \varrho_{18}, \varrho_5 \leq \varrho_{18}\}. \\
\quad\ \mathtt{letrec}\ \mathit{apply} = \mathtt{new}\, \varrho_6, \varrho_7, \varrho_8, \varrho_9.\, \lambda\, f.\, (\lambda\, x.\, (f \mathbin{@} x)\ \mathtt{at}\ \varrho_6)\ \mathtt{at}\ \varrho_5\ \mathtt{in} \\
\quad\quad \mathtt{let}\ \ y = \mathtt{new}\, \varrho_{10} : \{\varrho_5 \leq \varrho_{10}, \varrho_{10} \leq \varrho_4\}. \\
\quad\quad\quad\quad \mathtt{new}\, \varrho_{11} : \{\varrho_5 \leq \varrho_{11}, \varrho_{11} \leq \varrho_4\}. \\
\quad\quad\quad\quad\ \mathtt{new}\, \varrho_{12} : \{\varrho_5 \leq \varrho_{12}, \varrho_{11} \leq \varrho_{12}, \varrho_{10} \leq \varrho_{12}\}. \\
\quad\quad\quad\quad\quad (\mathtt{new}\, \varrho_{13} : \left\{\begin{array}{l}\varrho_{13} \leq \varrho_5, \varrho_5 \leq \varrho_{13}, \varrho_{13} \leq \varrho_{10}, \\ \varrho_{13} \leq \varrho_{11}, \varrho_{13} \leq \varrho_4, \varrho_{13} \leq \varrho_{12}\end{array}\right\}. \\
\quad\quad\quad\quad\quad\quad (\mathit{apply}\, [\varrho_{10}, \varrho_{11}, \varrho_4, \varrho_{12}]\ \mathtt{at}\ \varrho_{13}) \\
\quad\quad\quad\quad\quad\quad \mathbin{@} (\lambda\, n.\, (5\ \mathtt{at}\ \varrho_4)\ \mathtt{at}\ \varrho_{11})) \\
\quad\quad\quad\quad\quad \mathbin{@} (\mathtt{new}\, \varrho_{14} : \{\varrho_{14} \leq \varrho_{12}\}.\mathtt{lift}\, [\varrho_{14}, \varrho_{12}]\, (3\ \mathtt{at}\ \varrho_{14})) \\
\quad\quad \mathtt{in}\ \mathtt{let}\ \ z = \mathtt{new}\, \varrho_{15} : \{\varrho_5 \leq \varrho_{15}\}. \\
\quad\quad\quad\quad\quad\ \mathtt{new}\, \varrho_{16} : \{\varrho_5 \leq \varrho_{16}, \varrho_{16} \leq \varrho_2, \varrho_{16} \leq \varrho_{18}\}. \\
\quad\quad\quad\quad\quad\quad (\mathtt{new}\, \varrho_{17} : \left\{\begin{array}{l}\varrho_{17} \leq \varrho_5, \varrho_5 \leq \varrho_{17}, \varrho_{17} \leq \varrho_{18}, \\ \varrho_{17} \leq \varrho_{15}, \varrho_{17} \leq \varrho_{16}\end{array}\right\}. \\
\quad\quad\quad\quad\quad\quad\quad (\mathit{apply}\, [\varrho_{15}, \varrho_{16}, \varrho_{18}, \varrho_{18}]\ \mathtt{at}\ \varrho_{17}) \\
\quad\quad\quad\quad\quad\quad\quad \mathbin{@} (\lambda\, m.\, m\ \mathtt{at}\ \varrho_{16})) \\
\quad\quad\quad\quad\quad\quad \mathbin{@} (\mathtt{lift}\, [\varrho_2, \varrho_{18}]\, d) \\
\quad\quad\quad \mathtt{in}\ \mathtt{if}\ z\ \mathtt{then}\ \mathtt{lift}\, [\varrho_4, \varrho_3]\, y \\
\quad\quad\quad\quad\quad\quad \mathtt{else}\ \mathtt{lift}\, [\varrho_4, \varrho_3]\, (0\ \mathtt{at}\ \varrho_4))\ \mathtt{at}\ \varrho_1
\end{array}
$$

The initial constraint set is $C = \{\varrho_1 \leq \varrho_2, \varrho_1 \leq \varrho_3\}$. Additionally, we have that $\mathit{frv}\,(C, ((\mathtt{int}, \varrho_2) \xrightarrow{\epsilon.\varphi} (\mathtt{int}, \varrho_3), \varrho_1)) \cup \{\varrho_1\} \cup \varphi = \{\varrho_1, \varrho_2, \varrho_3\}$. Hence, the region annotation associated with the constraint completion is $\delta_c = \{\varrho_1 \mapsto \mathtt{d}, \varrho_2 \mapsto \mathtt{d}, \varrho_3 \mapsto \mathtt{d}\}$. Obviously we have that $\delta_{\text{init}} + \delta_c \models C$ and we can specialize e with $\delta_{\text{init}} + \delta_c$ to obtain the following residual program:

$$\underline{\lambda}\, d.\, (\underline{\mathtt{new}}\, \varrho_{18}.\, \mathtt{let}\ z = \underline{\mathtt{lift}}\, [\varrho_2, \varrho_{18}]\, d\ \mathtt{in}\ \underline{\mathtt{if}}\ z\ \underline{\mathtt{then}}\, (5\ \underline{\mathtt{at}}\ \varrho_3)\ \underline{\mathtt{else}}\, (0\ \underline{\mathtt{at}}\ \varrho_3))\ \underline{\mathtt{at}}\ \varrho_1$$

All regions, except ϱ_{18} and the regions ϱ_1, ϱ_2 and ϱ_3 from the constraint set C obtain a static binding time. In this particular example, it is possible to calculate the binding times of the regions before the specialization phase, since we do not have polymorphic region recursion here. Note the binding-time polymorphism in the argument and result of the applications of *apply*.

Example 2

In the second example we show that the conservative treatment of the effect in type rule *(C-Lam)* is necessary to obtain sound specialization. Consider the program:

$$(\mathtt{let}\ g = \lambda\, z.\, z\ \mathtt{in}\ \lambda\, y.\, \mathtt{let}\ f = \lambda\, x.\, (g \mathbin{@} (\lambda\, y.\, y))\ \mathtt{in}\ (\lambda\, f.\, 4) \mathbin{@} f) \mathbin{@} 2$$

The following region-annotated version is typeable in the $\vdash_t^t$ typing judgment of Figure 2.6 on page 27.

$$\begin{array}{l}\texttt{new}\,\varrho_1.\,((\texttt{new}\,\varrho_2.\,\texttt{let}\;\; g = \lambda\, z.\, z\;\texttt{at}\,\varrho_2\\ \qquad\qquad\qquad\texttt{in}\;\; \lambda\, y.\,(\texttt{let}\;\; f = \lambda\, x.\,(g \mathbin{@} (\lambda\, y.\, y\;\texttt{at}\,\varrho_3))\;\texttt{at}\,\varrho_3\\ \qquad\qquad\qquad\qquad\texttt{in}\;\; (\lambda\, f.\,(4\;\texttt{at}\,\varrho_3)\;\texttt{at}\,\varrho_1) \mathbin{@} f)\;\texttt{at}\,\varrho_1) \mathbin{@} (2\;\texttt{at}\,\varrho_1))\end{array}$$

The problem here is the function bound to f. It is effectively *dead* code, which means that it is only allocated but never used. Region inference in the $\vdash_t^t$ typing judgment is clever enough to see this, in particular, the region bound to ϱ_2 is deallocated before the closure for the function f is allocated. This behavior is sound in $\rightarrow$, but is not in $\rightsquigarrow$ if the region ϱ_3 is dynamic. In the latter case, the closure bound to g will be deallocated before specialization tries to reduce under the dynamic function f, causing a dereference of a dangling pointer. The adapted $\vdash_c$ (and $\vdash_t$) typing judgment avoids this undesirable behavior, by imposing a slightly more conservative annotation:

$$\begin{array}{l}\texttt{new}\,\varrho_1.\,\texttt{new}\,\varrho_2.\,((\texttt{let}\;\; g = \lambda\, z.\, z\;\texttt{at}\,\varrho_2\\ \qquad\qquad\qquad\texttt{in}\;\; \lambda\, y.\,(\texttt{let}\;\; f = \lambda\, x.\,(g \mathbin{@} (\lambda\, y.\, y\;\texttt{at}\,\varrho_3))\;\texttt{at}\,\varrho_3\\ \qquad\qquad\qquad\qquad\texttt{in}\;\; (\lambda\, f.\,(4\;\texttt{at}\,\varrho_3)\;\texttt{at}\,\varrho_1) \mathbin{@} f)\;\texttt{at}\,\varrho_1) \mathbin{@} (2\;\texttt{at}\,\varrho_1))\end{array}$$

This λ_{tt}^{region}-program specializes fine even when ϱ_1 and ϱ_2 are static and ϱ_3 dynamic. We omit the details.

6.3.4 On the CPS-transformation

Some of the attractiveness of using region inference as the flow analysis for a binding-time analysis are the coinciding goals of region separation in a memory management context and binding-time separation in a specialization context. This may imply that optimizations that are good for region inference are also good for partial evaluation and the other way around.

It is well-known that a CPS-transformation[2] allows for binding-time improvements [24] to improve program specialization. Unfortunately, a CPS-transformation destroys the memory management goals for the region calculus entirely. The problem is that in a CPS-transformed program, functions do not return and hence, regions are only allocated, not deallocated.

However, it seems that not the CPS-transformation itself is responsible for the problem, but the stack-based nature of region inference. Whenever region inference is solely used for a binding-time analysis (as is the case in this thesis), it does not really matter when a region is deallocated as long as region separation is still good. Moreover, our specializer implements context shifts that make a CPS-transformation redundant [90].

In fact, it is possible to develop a continuation-based region calculus which acts particularly well for a CPS-transformed program. This has been done for the capability calculus [28].

[2]CPS stands for *Continuation-Passing Style* [33].

(C-Constv)
$$\frac{C \vdash_{wft} (\mathtt{int}, \rho) \quad C \rhd \{\rho \leq s\}}{TE, C \vdash_c \langle c \rangle_\rho : (\mathtt{int}, \rho), \emptyset}$$

(C-Lamv)
$$\frac{\begin{array}{c} TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi \\ C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho) \quad \varphi \subseteq \varphi' \quad C \rhd \{\rho \leq s\} \end{array}}{TE, C \vdash_c \langle \lambda\, x.\, E \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset}$$

(C-Letrecv)
$$\frac{\begin{array}{c} P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \quad \sigma = \forall\, \vec{\alpha}\, .\hat{\sigma} \\ \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \quad TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \langle \lambda\, x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\, .C^{\uparrow P} \Rightarrow \tau \quad TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi \end{array}}{TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_\rho\ \mathtt{in}\ E_2 : \mu, \varphi}$$

(C-Regappv)
$$\frac{\begin{array}{c} TE, C' \vdash_c \langle \lambda\, x.\, E \rangle_{\rho'} : (\tau, \rho'), \emptyset \quad S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\} \\ \{\varrho_1, \ldots, \varrho_n\} \cap (frv\,(TE) \cup \{\rho, \rho'\}) = \emptyset \quad \tau' = S_r(\tau) \\ C \rhd S_r(C') \cup \{\rho' \leq \rho, \rho \leq \rho', \rho' \leq \rho'_1, \ldots, \rho' \leq \rho'_n\} \end{array}}{TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

(C-Constraint)
$$\frac{TE, C \vdash_c E : \mu, \varphi \quad C \rhd \{\varrho^\bullet \leq s\}}{TE, C_{\downarrow \varrho^\bullet} \vdash_c E : \mu, \varphi \setminus \{\varrho^\bullet\}}$$

(C-Reglam-s)
$$\frac{TE, C \vdash_c E : \mu, \varphi \quad \varrho \notin frv\,(TE, \mu) \quad C \rhd \{\varrho \leq s\}}{TE, C_{\downarrow \varrho} \vdash_c \mathtt{new}^{\mathtt{s}}\, \varrho . E : \mu, \varphi \setminus \{\varrho\}}$$

Figure 6.8: Static semantics of the λ_2^{region}-calculus - part I

6.3.5 Extending the Static Semantics

To prove soundness of the constraint analysis with respect to specialization, we have to extend the static semantics of Section 6.2.3 to all λ_2^{region}-Terms. All semantic objects retain their definitions, even though region placeholders now also range over deallocated regions from DeadRegionVar and the two virtual binding-time regions. We only have to extend the definition of a substitution S_s over these placeholders in the usual way: for all $\rho \in \mathsf{DeadRegionVar} \cup \mathsf{BTRegion}$, it holds that $S_s(\rho) = \rho$.

The extension of placeholders makes the type rules of Figure 6.2 work over deallocated regions as well. Additionally, we assume that all occurrences of e in these type rules can be read as λ_2^{region}-Terms E. The terms unaccounted for are listed in Figure 6.8 and Figure 6.9.

The rules *(C-Constv)*, *(C-Lamv)* and *(C-Letrecv)* are the pointer counterparts of rules *(C-Const)*, *(C-Lam)* and *(C-Letrec)* in Figure 6.2 respectively. As before (see Section 3.2.3 for the λ_s^{region}-calculus), pointers have no effect. Moreover, the rules *(C-Constv)* and *(C-Lamv)* include binding-time constraints which guarantee that the top-level region of the value is static for any solution of C. This is motivated by the fact that pointers are computational and only occur during specialization. The rule *(C-Regappv)*

types the application of a region closure and is quite similar to rule *(C-Regapp)* in Figure 6.2.

The rule *(C-Constraint)* is an auxiliary type rule, which removes constraints involving deallocated regions. However, this is only allowed if the constraint $\varrho^\bullet \leq \mathtt{s}$ is in C. This is necessary to guarantee that regions that previously had to be static because they were *younger* than $\varrho^\bullet$, still have static binding time for any solution of $C_{\downarrow\varrho^\bullet}$. A static region abstraction is typed by rule *(C-Reglam-s)* and is almost identical to rule *(C-Reglam)*, except for the additional binding-time constraint which guarantees that the abstracted region is statically annotated.

Similarly, rule *(C-Reglam-d)* in Figure 6.9 types a dynamic region abstraction. The additional constraint guarantees that ϱ has dynamic binding time. The rule *(C-Let-d)* is identical to rule *(C-Let)* and, with respect to rule *(C-Letrec)* in Figure 6.2, rule *(C-Letrec-d)* only adds a binding-time constraint, imposing a dynamic `letrec`-bound function.

The type rules for residual expressions are also very similar to their non-residual counterparts from Figure 6.2. We only list those rules that slightly deviate from their non-residual counterpart. The rules *(C-Const-r)* and *(C-Lam-r)* add the constraint $\mathtt{d} \leq \varrho$ to their constraint sets. Similarly, the rules *(C-Regapp-r)*, *(C-Lift-r)* and *(C-Reglam-r)* add a constraint to guarantee that every solution for C maps the top-level region onto `d`. At this point, it may be tempting to empty effects originating from residual syntax since these are not affected during specialization. However, in order to prove the specializer sound with respect to a standard semantics, we have to guarantee that the erasures of the above rules conservatively approximate standard type rules.

Constraints of the form $\varrho \leq \mathtt{s}$ and $\mathtt{d} \leq \varrho$ are only introduced in the type rules for computational terms, *i.e.* terms that are calculated during specialization. Hence, they do not comprise the solvability of the constraint set of a constraint completion.

Under the extended type system for the λ_2^{region}-calculus, it is easy to verify that the properties in Section 6.2.4 remain valid. Recall that the $\vdash_c$ typing judgment induces the $\vdash_t$ typing judgment, *i.e.* $TE, C \vdash_c E : \mu, \varphi$ defines $|TE| \vdash_t |E| : |\mu|, |\varphi|$. This also holds for the extended rules of Figure 6.8 and Figure 6.9. Since the erasure of a λ_2^{region}-expression is a λ_s^{region}-expression, we actually have that the $\vdash_t$ typing judgment now types λ_s^{region}-expressions and Proposition 6.2.12 on page 118 still holds for the extended $\vdash_t$ typing judgment.

6.4 Soundness of the Constraint Analysis

Since the constraint analysis is the static semantics of the specialization rules (which form the dynamic semantics), we can follow then schema depicted in Section 2.2.3 to prove the constraint analysis sound. This strategy is also used by Hatcliff and Danvy for the binding-time analysis of Moggi's computational metalanguage [60].

(C-Reglam-d)
$$\frac{TE, C \vdash_c E : \mu, \varphi \quad \varrho \notin frv\,(TE, \mu) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\, \varrho.E : \mu, \varphi \setminus \{\varrho\}}$$

(C-Let-d)
$$\frac{TE, C \vdash_c R : \mu', \varphi' \quad TE + \{x \mapsto \mu'\}, C \vdash_c E : \mu, \varphi}{TE, C \vdash_c \mathtt{let}^{\mathsf{d}}\, x = R\, \mathtt{in}\, E : \mu, \varphi \cup \varphi'}$$

(C-Letrec-d)
$$\frac{\begin{array}{c} P = \{\varrho_1, \ldots, \varrho_n\} \quad (P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset \\ \hat{\sigma} = \forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\, .C^{\uparrow P} \Rightarrow \tau \quad \{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \lambda\, x.\, R\ \mathtt{at}\ \rho : (\tau, \rho), \varphi' \quad C \rhd \{\mathtt{d} \leq \rho\} \\ \sigma = \forall\, \vec{\alpha}\, .\hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E : \mu, \varphi \end{array}}{TE, C_{\downarrow P} \vdash_c \mathtt{letrec}^{\mathsf{d}}\, f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, R\ \mathtt{at}\ \rho\ \mathtt{in}\, E : \mu, \varphi \cup \varphi'}$$

(C-Const-r)
$$\frac{C \vdash_{wft} (\mathtt{int}, \varrho) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C \vdash_c c\ \underline{\mathtt{at}}\ \varrho : (\mathtt{int}, \varrho), \{\varrho\}}$$

(C-Lam-r)
$$\frac{\begin{array}{c} TE + \{x \mapsto \mu_1\}, C \vdash_c R : \mu_2, \varphi \\ C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho) \quad \varphi \subseteq \varphi' \quad C \rhd \{\mathtt{d} \leq \varrho\} \end{array}}{TE, C \vdash_c \underline{\lambda}\, x.\, R\ \underline{\mathtt{at}}\ \varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \varphi \cup \{\varrho\}}$$

(C-Regapp-r)
$$\frac{\begin{array}{c} TE(f) = (\sigma, \rho) \quad \sigma = \forall\, \vec{\alpha}\, .\forall\, \vec{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n\, .C'' \Rightarrow \tau \\ (C', \tau') \prec \sigma \ \mathit{via}\ (S_t, S_e, S_r) \quad S_r = \{\varrho_1 \mapsto \varrho'_1, \ldots, \varrho_n \mapsto \varrho'_n\} \\ C \rhd C' \cup \{\varrho' \leq \rho, \rho \leq \varrho', \varrho' \leq \varrho'_1, \ldots, \varrho' \leq \varrho'_n, \mathtt{d} \leq \varrho'\} \end{array}}{TE, C \vdash_c f\ [\varrho'_1, \ldots, \varrho'_n]\ \underline{\mathtt{at}}\ \varrho' : (\tau', \varrho'), \{\rho, \varrho'\}}$$

(C-Lift-r)
$$\frac{TE, C \vdash_c R : (\mathtt{int}, \varrho_1), \varphi \quad C \rhd \{\varrho_1 \leq \varrho_2, \mathtt{d} \leq \varrho_1\}}{TE, C \vdash_c \underline{\mathtt{lift}}\, [\varrho_1, \varrho_2]\, R : (\mathtt{int}, \varrho_2), \varphi \cup \{\varrho_1, \varrho_2\}}$$

(C-Reglam-r)
$$\frac{TE, C \vdash_c R : \mu, \varphi \quad \varrho \notin frv\,(TE, \mu) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C_{\downarrow\varrho} \vdash_c \underline{\mathtt{new}}\, \varrho.\, R : \mu, \varphi \setminus \{\varrho\}}$$

Figure 6.9: Static semantics of the λ_2^{region}-calculus - part II

6.4.1 Auxiliary Results

As usual, we formulate some auxiliary result. Values in the λ_2^{region}-calculus also have no effect:

Lemma 6.4.1 *If* $TE, C \vdash_c V : \mu, \varphi$ *then* $\varphi = \emptyset$.

Under certain conditions, we have that residual code has a top-level region variable which can only be dynamic.

Lemma 6.4.2 *Suppose* $TE, C \vdash_c R : (\tau, \varrho), \varphi$; $C \vdash_{wft} TE$; $\delta \models C$ *and for all* $(\sigma, \rho') \in Ran\,(TE)$, *it holds that* $\delta(\rho') = \mathtt{d}$. *Then,* $\delta(\varrho) = \mathtt{d}$.

Proof. Rule induction on the derivation of $TE, C \vdash_c R : (\tau, \varrho), \varphi$. □
The condition that for all $(\sigma, \rho) \in Ran\,(TE)$, it holds that $\delta(\rho) = \mathtt{d}$ is used in several properties. It expresses that a type environment has to be dynamic in a specialization context. Alternatively, one could use the less restrictive condition that for all $x \in fv\,(R)$ where $TE(x) = (\sigma, \rho)$ it holds that $\delta(\rho) = \mathtt{d}$. We choose to use the former.

The substitution lemma for `let`-bound variables is very similar to that of the λ_s^{region}-calculus (see Section 3.3.1).

Lemma 6.4.3 (First Substitution Lemma) *Suppose* $TE + \{x \mapsto \mu\}, C \vdash_c E : \mu', \varphi'$ *and* $TE, C \vdash_c V : \mu, \emptyset$, *then* $TE, C \vdash_c E[x \mapsto V] : \mu', \varphi'$.

Proof. By induction on the derivation of $TE + \{x \mapsto \mu\}, C \vdash_c E : \mu', \varphi'$. The only interesting case is the application of *(C-Var)* to (free occurrences of) x. Suppose $TE' = TE + \{x \mapsto \mu\}$, then because $TE'(x) = TE + \{x \mapsto \mu\}(x) = \mu$ and rule *(C-Var)* we have $TE', C \vdash_c x : \mu, \emptyset$. On the other hand, $x[x \mapsto V] = V$ and, by the second assumption, $TE, C \vdash_c V : \mu, \emptyset$ since $\varphi' = \emptyset$, by Lemma 6.4.1.

All other cases are simple appeals to the induction hypothesis. □

Substitution of variables bound to region polymorphic expressions is an extension of Lemma 3.3.6 on page 35 to handle constraints.

Lemma 6.4.4 (Second Substitution Lemma) *Suppose*

1. $P = \{\varrho_1 \ldots, \varrho_n\}$ *and* $(P \cup \{\vec{\epsilon}\}) \cap (fv\,(TE) \cup \{\rho\}) = \emptyset$ *and* $\{\vec{\alpha}\} \cap ftv\,(TE) = \emptyset$ *and* $\sigma = \forall\,\vec{\alpha}\,.\hat{\sigma}$ *and* $\hat{\sigma} = \forall\,\vec{\epsilon}.\forall\,\varrho_1, \ldots, \varrho_n\,.C^{\uparrow P} \Rightarrow \tau$
2. $TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$
3. $TE + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \langle \lambda\, x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset$

then $TE, C_{\downarrow P} \vdash_c E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_\rho\ \mathtt{in}\ E_1 \rangle_\rho] : \mu, \varphi$.

Proof. By induction on the derivation of $TE + \{f \mapsto (\sigma, \rho)\}, C_{\downarrow P} \vdash_c E_2 : \mu, \varphi$.

Except for free occurrences of f in E_2, all cases are relative straightforward applications of the induction hypothesis. However, to be able to use the induction hypothesis,

we must guarantee that the constraint set in the above judgment always has the form $C_{\downarrow P}$, where P is fixed.

For the type rules where the constraint set does not change, this is trivially true. For the rules where the judgment takes out constraints with respect to the premise (for example *(C-Reglam)*), it is easy to see that the above requirement is satisfied as well because of Lemma 6.2.2.

The only type rule where the premise may have a smaller constraint set in its premise is rule *(C-Regappv)*. In this case, we need to apply the induction hypothesis on a judgment with constraint set C' where we have that $C_{\downarrow P} \rhd S_r(C') \cup \{\rho', \rho, \rho'_1, \ldots, \rho'_n\}$. Therefore we have by definition of consistency that $S_r(C') \subseteq C_{\downarrow P}$ and hence that $P \cap S_r(C') = \emptyset$. By α-renaming the region variables $\{\varrho_1, \ldots, \varrho_n\}$, we can also assume without loss of generality that $\{\varrho_1, \ldots, \varrho_n\} \cap P = \emptyset$. Therefore, we have that $C' = C'_{\downarrow P}$, which makes the induction hypothesis applicable to the judgment in the premise.

Consider now the base case where we assume that E_2 is of the form f. In that case, the final derivation is as follows:

$$\frac{\begin{array}{c} TE(f) = (\underline{\sigma}, \rho) \quad \underline{\sigma} = \forall\, \vec{\alpha}\,.\forall\, \vec{\epsilon}\,.\forall\, \underline{\varrho}_1, \ldots, \underline{\varrho}_n\,.\chi(C)^{\uparrow \underline{P}} \Rightarrow \chi(\tau) \\ (C', \tau') \prec \underline{\sigma} \;\; \textit{via} \;\; S_s \quad S_r = \{\underline{\varrho}_1 \mapsto \rho'_1, \ldots, \underline{\varrho}_n \mapsto \rho'_n\} \quad S_s = (S_t, S_e, S_r) \\ C_0 = \{\rho' \leq \rho, \rho \leq \rho', \rho' \leq \rho'_1, \ldots, \rho' \leq \rho'_n\} \quad C_{\downarrow P} \rhd C' \cup C_0 \end{array}}{TE + \{f \mapsto (\underline{\sigma}, \rho)\}, C_{\downarrow P} \vdash_c f\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}}$$

We have α-renamed the variables $P = \{\varrho_1, \ldots, \varrho_n\}$ into $\underline{P} = \{\underline{\varrho}_1, \ldots, \underline{\varrho}_n\}$ via the *renaming* region substitution $\chi = \{\varrho_1 \mapsto \underline{\varrho}_1, \ldots, \varrho_n \mapsto \underline{\varrho}_n\}$. In addition, we assume that

$$\underline{P} \cap (\mathit{frv}\,(TE, C) \cup \{\rho, \rho'\} \cup P \cup \mathit{Img}\,(S_s)) = \emptyset \tag{6.41}$$

Observe that the type schemes $\underline{\sigma}$ and σ are equal up to α-renaming and similarly for $\underline{\hat{\sigma}}$ and $\hat{\sigma}$.

Abbreviating $A = \langle \Lambda\, \underline{\varrho}_1, \ldots, \underline{\varrho}_n . \lambda\, x.\, E_1 \rangle_\rho$, we must show that

$$TE, C_{\downarrow P} \vdash_c \langle \Lambda\, \underline{\varrho}_1, \ldots, \underline{\varrho}_n . \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1) \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}$$

That is, by rule *(C-Regappv)*, there must exist τ'' and C'' such that

- $TE, C'' \vdash_c \langle \lambda\, x.\, \mathtt{letrec}\ f = A\ \mathtt{in}\ \chi(E_1) \rangle_{\rho'} : (\tau'', \rho'), \emptyset$;
- $\underline{P} \cap (\mathit{frv}\,(TE) \cup \{\rho, \rho'\}) = \emptyset$ (which is immediate by construction);
- $\tau' = S_r(\tau'')$;
- $C_{\downarrow P} \rhd S_r(C'') \cup C_0$

A suitable choice for τ'' is $S'_s(\chi(\tau))$ where $S'_s = (S_t, S_e, I_r)$ and we take $\chi(C)$ for C''. Construction of $\underline{P}$ makes that $\mathit{Supp}\,(S_r) \cap \mathit{Img}\,(S'_s) = \emptyset$, hence we have $S_s = S_r \circ S'_s$. Since $(C', \tau') \prec \underline{\sigma}$ *via* S_s, the type τ'' satisfies the third requirement.

To show the last requirement, note that also via $(C', \tau') \prec \underline{\sigma}$ *via* S_s we have that $S_r(\chi(C)^{\uparrow \underline{P}}) = C' \subseteq C_{\downarrow P}$ and surely $S_r(\chi(C)^{\uparrow \underline{P}}) = S_r(\chi(C^{\uparrow P}))$. We also have by construction of χ that $S_r(\chi(C_{\downarrow P})) = C_{\downarrow P} \subseteq C_{\downarrow P}$. Putting this together yields $S_r(\chi(C_{\downarrow P})) \cup$

$S_r(\chi(C^{\uparrow P})) \subseteq C_{\downarrow P}$ or also that $S_r(\chi(C)) \subseteq C_{\downarrow P}$. The requirement is now proven since $C_0 \subseteq C_{\downarrow P}$ is given by one of the premises above.

It remains to show the judgment of the first requirement, which we can formulate as follows:

$$TE, \chi(C) \vdash_c \begin{array}{l} \langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_\rho\ \texttt{in}\ \chi(E_1) \rangle_{\rho'} \\ : (S'_s(\chi(\tau)), \rho'), \emptyset \end{array} \tag{6.42}$$

To see this, it is first shown that

$$TE, \chi(C) \vdash_c \langle \lambda\, x.\, \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_\rho\ \texttt{in}\ \chi(E_1) \rangle_\rho : (\chi(\tau), \rho), \emptyset \tag{6.43}$$

Item 3 makes that

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \rho)\}, C \vdash_c \langle \lambda\, x.\, E_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{6.44}$$

holds for any μ_1. Applying Lemma 6.2.6 with the renaming substitution χ, we obtain

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \rho)\}, \chi(C) \vdash_c \langle \lambda\, x.\, \chi(E_1) \rangle_\rho : (\chi(\tau), \rho), \emptyset$$

Because of rule *(C-Lamv)*, we can take $\tau = \mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2$ and for some $\varphi'' \subseteq \varphi'$ have the judgment

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \rho)\} + \{x \mapsto \chi(\mu_1)\}, \chi(C) \vdash_c \chi(E_1) : \chi(\mu_2), \chi(\varphi'') \tag{6.45}$$

where

$$\chi(C) \vdash_{wft} (\chi(\mu_1) \xrightarrow{\epsilon'.\chi(\varphi')} \chi(\mu_2), \rho) \tag{6.46}$$

Applying Lemma 6.2.7 to (6.45) for $\hat{\sigma} \prec \sigma$ and observing that $TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \rho)\} + \{x \mapsto \chi(\mu_1)\} = TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\hat{\sigma}, \rho)\}$ because $x \neq f$, yields

$$TE + \{x \mapsto \chi(\mu_1)\} + \{f \mapsto (\sigma, \rho)\}, \chi(C) \vdash_c \chi(E_1) : \chi(\mu_2), \chi(\varphi'') \tag{6.47}$$

Observe now that $\chi(C) = \chi(C)_{\downarrow P}$ since $frv\,(\chi(C)) \cap P = \emptyset$. Hence, from rule *(C-Letrecv)*, Item 1, assumptions (6.41), and the judgments (6.44) and (6.47), we obtain

$$TE + \{x \mapsto \chi(\mu_1)\}, \chi(C) \vdash_c \begin{array}{l} \texttt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_\rho\ \texttt{in}\ \chi(E_1) \\ : \chi(\mu_2), \chi(\varphi'') \end{array} \tag{6.48}$$

Using rule *(C-Lamv)* and the judgments (6.46) and (6.48), we can derive judgment (6.43). Now, applying Lemma 6.2.6 with S'_s to judgment (6.43) yields

$$S'_s(TE), S'_s(\chi(C)) \vdash_c S'_s(\langle \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ \chi(E_1) \rangle_\rho) : S'_s(\chi(\tau), \rho), S'_s(\emptyset) \tag{6.49}$$

The support of $S'_s = (S_t, S_e, I_r)$ is a subset of $\{\vec{\epsilon}, \vec{\alpha}\}$ since we have the instantiation $(C', \tau') \prec \sigma$ *via* (S_t, S_e, S_r). By Item 1, the set $\{\vec{\epsilon}, \vec{\alpha}\}$ is disjoint from $fv\,(TE)$. Therefore,

- $S'_s(TE) = TE$;
- $S'_s(\langle \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ \chi(E_1) \rangle_\rho) = \langle \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ \chi(E_1) \rangle_\rho$ due to I_r;

- $S'_s(\chi(C)) = \chi(C)$ due to I_r; and
- $S'_s(\chi(\tau), \rho) = (S'_s(\chi(\tau)), \rho)$.

So, we have $TE, \chi(C) \vdash_c \langle \lambda\, x.\, \texttt{letrec}\ f = A\ \texttt{in}\ \chi(E_1) \rangle_\rho : (S'_s(\chi(\tau)), \rho), \emptyset$.

Given (6.46), we also have $\chi(C) \vdash_{wft} (\chi(\mu_1) \xrightarrow{\epsilon'.\chi(\varphi')} \chi(\mu_2), \rho')$ since $\{\rho \leq \rho', \rho' \leq \rho\} \subseteq C_0 \subseteq C_{\downarrow P} \subseteq C$ and therefore, by Item 1, $\{\rho \leq \rho', \rho' \leq \rho\} \subseteq \chi(C)$. Judgment (6.42) can now be derived by replacing ρ for ρ' in rule *(C-Lamv)*. This concludes the proof. □

The *canonical forms* lemma specifies the form of an answer if its type is known.

Lemma 6.4.5 (Canonical Forms Lemma) *Suppose $TE, C \vdash_c Q^\mu : \mu, \varphi$; $\mu = (\tau, \rho)$; $C \vdash_{wft} TE$; $\delta \models C$ and for all $(\sigma, \rho') \in Ran(TE)$, it holds that $\delta(\rho') = \mathtt{d}$, then*

1. *if $\delta(\rho) = \mathtt{d}$, then Q^μ is residual code of the form R*
2. *if $\delta(\rho) = \mathtt{s}$, then Q^μ is a static answer $\mathrm{Q}^\mathtt{s}$. Additionally, if $\mathrm{Q}^\mathtt{s}$ is a value V, then we also have one of the following:*
   (a) *if $\tau = \texttt{int}$, then V has the form $\langle c \rangle_\rho$ for some constant c*
   (b) *if $\tau = \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$, then V has the form $\langle \lambda\, x.\, E \rangle_\rho$ for some x and E.*

Proof. The proof is by induction on the derivation of $TE, C \vdash_c Q^\mu : \mu, \varphi$. The assumption that $\delta \models C$ and the binding-time constraints in the rules of Figure 6.8 and Figure 6.9 are critically used to identify canonical forms. □

6.4.2 Type Preservation

Type preservation for the λ_2^{region}-calculus is somewhat more technical as usual due to region annotations and constraint sets.

Proposition 6.4.6 (Type Preservation) *Suppose $TE, C \vdash_c E^\mu : \mu, \varphi$ and $\delta \models C$ for $\delta_{\text{init}} \subseteq \delta$. If $\delta, E^\mu \rightsquigarrow \delta', E'$ then there exists a C' such that $TE, C' \vdash_c E' : \mu, \varphi'$ where $\varphi' \subseteq \varphi$; $\delta' \models C'$; $C \subseteq C'$ and $\delta \subseteq \delta'$.*

The fact that both the constraint set C and region annotation δ grow is a consequence of binding-time calculations during specialization. The new constraint set C' adds at most constraints of the form $\rho \leq \mathtt{s}$ or $\mathtt{d} \leq \rho$ to reflect binding-time knowledge into the type system: in particular one could prove that $C_{\downarrow\{\mathtt{s},\mathtt{d}\}} = C'_{\downarrow\{\mathtt{s},\mathtt{d}\}}$.

Proof. By induction on the number of subsequent uses of the type rules *(C-Effect)* and *(C-Constraint)* at the end of E's type derivation. The base case for this induction is in turn proven by rule induction on the definition of $\rightsquigarrow$

If $TE, C \vdash_c E : \mu, \varphi$ derives from $TE, C \vdash_c E : \mu, \varphi \cup \{\epsilon\}$ by rule *(C-Effect)*, for some $\epsilon \notin fev(TE, \mu, \varphi)$, then induction yields that $TE, C \vdash_c E' : \mu, \varphi'$ where $\varphi' \subseteq \varphi \cup \{\epsilon\}$. If $\epsilon \notin \varphi'$ then $\varphi' \subseteq \varphi$ and the claim holds. Otherwise, use *(C-Effect)* to get $TE, C \vdash_c E' : \mu, \varphi' \setminus \{\epsilon\}$ where $\varphi' \setminus \{\epsilon\} \subseteq \varphi \cup \{\epsilon\} \setminus \{\epsilon\} = \varphi$, as required.

If $TE, C_{\downarrow\varrho^\bullet} \vdash_c E : \mu, \varphi$ derives from $TE, C \vdash_c E : \mu, \varphi$ by rule *(C-Constraint)*, then we also have $C \rhd \{\varrho^\bullet \leq \mathtt{s}\}$. We are given that $\delta \models C_{\downarrow\varrho^\bullet}$ and $\delta_{\text{init}} \subseteq \delta$. This together with the fact that $C \setminus C_{\downarrow\varrho^\bullet} = C^{\uparrow\varrho^\bullet}$ by Lemma 6.2.1.1, makes that $\delta \models C^{\uparrow\varrho^\bullet}$ and therefore $\delta \models C$. Hence, applying the induction hypothesis gives that $TE, C' \vdash_c E' : \mu, \varphi'$ where $C \subseteq C'$, $\varphi' \subseteq \varphi$, $\delta' \models C'$ and $\delta \subseteq \delta'$. Since $C \subseteq C'$, we trivially have $C' \rhd \{\varrho^\bullet \leq \mathtt{s}\}$. Applying rule *(C-Constraint)* proves the case.

If the last rule in the proof of $TE, C \vdash_c E : \mu, \varphi$ is not *(C-Effect)* or *(C-Constraint)*, we proceed by rule induction on $\rightsquigarrow$. When one of the accompanying facts $\varphi' \subseteq \varphi$, $\delta \subseteq \delta'$, $C \subseteq C'$ or $\delta \models C$ is immediate (for example, because the objects are identical or an immediate result by induction) we ignore them in the proof of the particular case. Also, recall that $\widetilde{C}$ builds the least transitive closure of C.

Case $\delta, c\ \mathtt{at}\ \varrho \rightsquigarrow \delta, \langle c\rangle_\varrho$ *if* $\delta(\varrho) = \mathtt{s}$**.**
By rule *(C-Const)*, we have that $C \vdash_{wft} (\mathtt{int}, \varrho)$. So, take $C'' = C \cup \{\varrho \leq \mathtt{s}\}$ and $C' = \widetilde{C''}$. Surely, $C \subseteq C'$ and since $\delta \models C$ and $\delta \models \{\varrho \leq \mathtt{s}\}$, we also have $\delta \models C'$ by Lemma 6.2.9. Of course we have $C' \vdash_{wft} (\mathtt{int}, \varrho)$ and hence by *(C-Constv)* that $TE, C' \vdash_c \langle c\rangle_\varrho : (\mathtt{int}, \varrho), \emptyset$. Since $\emptyset \subseteq \varphi$, the case is proven.

Cases for the rules (6.2) and (6.3). Analogous.

Case $\delta, \mathtt{new}^{\mathtt{s}}\ \varrho.Q \rightsquigarrow \delta, Q[\varrho \mapsto \varrho^\bullet]$**.**
We have by assumption that $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}\ \varrho.\, Q : \mu, \varphi \setminus \{\varrho\}$ and $\delta \models C_{\downarrow\varrho}$. Hence, from rule *(C-Reglam-s)*, this implies that $TE, C \vdash_c Q : \mu, \varphi$ with $\varrho \notin \mathit{frv}\,(TE, \mu)$ and $C \rhd \{\varrho \leq \mathtt{s}\}$. Using Lemma 6.2.6, we obtain that $TE, C[\varrho \mapsto \varrho^\bullet] \vdash_c Q[\varrho \mapsto \varrho^\bullet] : \mu, \varphi[\varrho \mapsto \varrho^\bullet]$. The fact that $C \rhd \{\varrho \leq \mathtt{s}\}$ implies that $C[\varrho \mapsto \varrho^\bullet] \rhd \{\varrho^\bullet \leq \mathtt{s}\}$. So from rule *(C-Constraint)*, we obtain $TE, C[\varrho \mapsto \varrho^\bullet]_{\downarrow\varrho^\bullet} \vdash_c Q[\varrho \mapsto \varrho^\bullet] : \mu, (\varphi[\varrho \mapsto \varrho^\bullet]) \setminus \{\varrho^\bullet\}$. The claim is proven by observing that $C[\varrho \mapsto \varrho^\bullet]_{\downarrow\varrho^\bullet} = C_{\downarrow\varrho}$ and $(\varphi[\varrho \mapsto \varrho^\bullet]) \setminus \{\varrho^\bullet\} = \varphi \setminus \{\varrho\}$.

Case $\delta, \mathtt{lift}\,[\varrho_1, \varrho_2]\,\langle c\rangle_{\varrho_1} \rightsquigarrow \delta, \langle c\rangle_{\varrho_2}$ *if* $\delta(\varrho_2) = \mathtt{s}$**.**
So, $TE, C \vdash_c \mathtt{lift}\,[\varrho_1, \varrho_2]\,\langle c\rangle_{\varrho_1} : (\mathtt{int}, \varrho_2), \varphi \cup \{\varrho_1, \varrho_2\}$. Because of rule *(C-Lift)* and rule *(C-Constv)*, we have that $\varphi = \emptyset$; $C \rhd \{\varrho_1 \leq \varrho_2, \varrho_1 \leq \mathtt{s}\}$ and $C \vdash_{wft} (\mathtt{int}, \varrho_1)$. Take C' such that for $C'' = C \cup \{\varrho_1 \leq \mathtt{s}\}$, $C' = \widetilde{C''}$. Now, we obviously have $C' \vdash_{wft} (\mathtt{int}, \varrho_1)$ and also $\delta \models C'$ because of $\delta \models C$; $\delta \models \{\varrho_2 \leq \mathtt{s}\}$ and Lemma 6.2.9. Type rule *(C-Constv)* makes that $TE, C \vdash_c \langle c\rangle_{\varrho_2} : (\mathtt{int}, \varrho_2), \emptyset$ and since $\emptyset \subseteq \{\varrho_1, \varrho_2\}$, the case is proven.

Case $\delta, \langle \lambda\, x.\, E\rangle_\varrho\, @\, Q \rightsquigarrow \delta, \mathtt{let}\ x = Q\ \mathtt{in}\ E$**.**
By assumption and Lemma 6.4.1 we have $TE, C \vdash_c \langle \lambda\, x.E\rangle_\varrho\, @\, Q : \mu_2, \varphi \cup \varphi' \cup \{\epsilon, \varrho\}$ via *(C-App)*, where $TE, C \vdash_c \langle \lambda\, x.\, E\rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \varrho), \emptyset$ and $TE, C \vdash_c Q : \mu_1, \varphi'$. So, by rule *(C-Lamv)* this makes $TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi''$ too, with $\varphi'' \subseteq \varphi$. Using *(C-Let)* on the latter two judgments for Q and E, we obtain $TE, C \vdash_c \mathtt{let}\ x = Q\ \mathtt{in}\ E : \mu_2, \varphi'' \cup \varphi'$, where $\varphi'' \cup \varphi' \subseteq \varphi \cup \varphi' \subseteq \varphi \cup \varphi' \cup \{\epsilon, \varrho\}$.

Case $\delta, \mathtt{let}\ x = V\ \mathtt{in}\ E \rightsquigarrow \delta, E[x \mapsto V]$**.**
Immediate by rule *(C-Let)* and Lemma 6.4.3.

Case $\dfrac{\delta, \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E\rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho_0 \rightsquigarrow}{\delta, \langle \lambda\, x.\, E[\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n]\rangle_{\varrho_0}}$.

So, rule *(C-Regappv)* applies:

$$\frac{\begin{array}{c} TE, C' \vdash_c \langle \lambda\, x.\, E\rangle_{\varrho_0} : (\tau, \varrho_0), \emptyset \quad \{\varrho_1, \ldots, \varrho_n\} \cap (\mathit{frv}\,(TE) \cup \{\varrho, \rho_0\}) = \emptyset \\ S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\} \quad \tau' = S_r(\tau) \\ C \rhd S_r(C') \cup \{\varrho_0 \leq \varrho, \varrho \leq \varrho_0, \varrho_0 \leq \rho'_1, \ldots, \varrho_0 \leq \rho'_n\} \end{array}}{TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E\rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \varrho_0 : (\tau', \varrho_0), \{\varrho, \varrho_0\}}$$

By Lemma 6.2.6, we can apply S_r on the derivation for the pointer to the lambda abstraction. This yields: $TE, S_r(C') \vdash_c \langle \lambda\, x.\, S_r(E)\rangle_{\varrho_0} : (S_r(\tau), \varrho_0), \emptyset$. Since $\emptyset \subseteq \{\varrho, \varrho_0\}$; $S_r(\tau) = \tau'$; and $C \rhd S_r(C') \cup \{\varrho_0 \leq \varrho, \varrho \leq \varrho_0, \varrho_0 \leq \rho'_1, \ldots, \varrho_0 \leq \rho'_n\}$, the case is proven by Lemma 6.2.5.

Case $\dfrac{\delta, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E_1\rangle_\rho\ \mathtt{in}\ E_2 \rightsquigarrow}{\delta, E_2[f \mapsto \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, \mathtt{letrec}\ f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E_1\rangle_\rho\ \mathtt{in}\ E_1\rangle_\rho]}$.

The left-hand side can only be typed by rule *(C-Letrecv)*. This, however, gives exactly the requirements for Lemma 6.4.4, which on its turn proves the case.

Case $\dfrac{\delta, E \rightsquigarrow \delta', E'}{\delta, E \mathbin{@} E'' \rightsquigarrow \delta', E' \mathbin{@} E''}$.

The last step in the type derivation for $E \mathbin{@} E''$ must be rule *(C-App)*, so we have $TE, C \vdash_c E \mathbin{@} E'' : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$. This means that $TE, C \vdash_c E : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1$ and $TE, C \vdash_c E'' : \mu_1, \varphi_2$ holds too. By induction, we have that $TE, C \vdash_c E' : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi'_1$ for $\varphi'_1 \subseteq \varphi_1$, so the typing is immediately proven using rule *(C-App)*.

Cases for the rules (6.11) through (6.14).

Analogous to the previous case.

Case $\delta, \mathtt{new}\ \varrho : C_0\, .E \rightsquigarrow \delta + \{\varrho \mapsto \mathtt{s}\}, \mathtt{new}^{\mathtt{s}}\ \varrho.E$ *if* $\varrho \notin \mathit{Dom}\,(\delta)$ *and* $\delta + \{\varrho \mapsto \mathtt{s}\} \models C_0$.

First, observe that $\delta \subseteq \delta + \{\varrho \mapsto \mathtt{s}\}$. By assumption there exists a C such that $C_0 = C^{\uparrow\varrho}$ and $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}\ \varrho : C_0.\, E : \mu, \varphi \setminus \{\varrho\}$ via *(C-Reglam)*, and $\delta \models C_{\downarrow\varrho}$. Hence, we have $TE, C \vdash_c E : \mu, \varphi$ where $\varrho \notin \mathit{frv}\,(\mu, TE)$. Take $C' = \widetilde{C''}$ where $C'' = C \cup \{\varrho \leq \mathtt{s}\}$. So, by Lemma 6.2.5, we have that $TE, C' \vdash_c E : \mu, \varphi$. As $C' \rhd \{\varrho \leq \mathtt{s}\}$, we can apply rule *(C-Reglam-s)* to obtain $TE, C'_{\downarrow\varrho} \vdash_c \mathtt{new}\ \varrho.\, E : \mu, \varphi \setminus \{\varrho\}$. Trivially we have $C \subseteq C'$ and hence also $C_{\downarrow\varrho} \subseteq C'_{\downarrow\varrho}$. Moreover, since we have that $\delta \models C_{\downarrow\varrho}$ and $\delta + \{\varrho \mapsto \mathtt{s}\} \models C_0 (= C^{\uparrow\varrho})$, applying Lemma 6.2.11 yields that $\delta + \{\varrho \mapsto \mathtt{s}\} \models C$. Also, we have $\delta + \{\varrho \mapsto \mathtt{s}\} \models \{\varrho \leq \mathtt{s}\}$, so Lemma 6.2.9 makes that $\delta + \{\varrho \mapsto \mathtt{s}\} \models C'$. The case is now proven as we obviously have $\delta + \{\varrho \mapsto \mathtt{s}\} \models C'_{\downarrow\varrho}$ too.

Case $\delta, \mathtt{new}\ \varrho : C\, .E \rightsquigarrow \delta + \{\varrho \mapsto \mathtt{d}\}, \mathtt{new}^{\mathtt{d}}\ \varrho.E$ *if* $\varrho \notin \mathit{Dom}\,(\delta)$ *and* $\delta + \{\varrho \mapsto \mathtt{d}\} \models C$.

Analogous to the previous case.

Case $\delta, \mathtt{lift}\ [\varrho_1, \varrho_2]\ \langle c\rangle_{\varrho_1} \rightsquigarrow \delta, c\ \mathtt{at}\ \varrho_2$ *if* $\delta(\varrho_2) = \mathtt{d}$.

Analogous to the case for `lift` with $\delta(\varrho_2) = \mathtt{s}$.

Cases for the rules (6.17) through (6.24).
These rules are all fairly straightforward by the design of the type rules for residual expressions. Reductions (6.19), (6.20) and (6.23) rely on the design of the type rules *(C-Letrec-d)*, *(C-Reglam-d)* and *(C-Let-d)* respectively. Reduction rule (6.21) needs Lemma 6.4.2 and reduction (6.24) relies on the assumption about the type environment.

Cases for the rules (6.25) and (6.26)
Trivial by the type rules *(C-Let)*, *(C-Let-d)*, *(C-Letrec)* and *(C-Letrec-d)*.

Case $\dfrac{\delta, E \rightsquigarrow \delta', E'}{\delta, \lambda\, x.\, E\ \mathtt{at}\ \varrho \rightsquigarrow \delta', \lambda\, x.\, E'\ \mathtt{at}\ \varrho}$ *if* $\delta(\varrho) = \mathtt{d}$.
We have to type the expressions with rule *(C-Lam)*, which gives us that $TE, C \vdash_c \lambda\, x.\, E\ \mathtt{at}\ \varrho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho), \{\varrho\} \cup \varphi$, where we have that $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho)$; $\varphi \subseteq \varphi'$; and $TE + \{x \mapsto \mu_1\}, C \vdash_c E : \mu_2, \varphi$. The induction hypothesis, the premise of (6.27) and the latter judgment give us that $TE + \{x \mapsto \mu_1\}, C \vdash_c E' : \mu_2, \varphi''$ with $\varphi'' \subseteq \varphi$. So we can apply *(C-Lam)* to prove the case.

Cases for the rules (6.28) through (6.31).
All similarly simple appeals to the induction hypothesis.

Case $\delta, \mathtt{lift}\,[\rho_1, \rho_2]\,(\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}}) \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{lift}\,[\rho_1, \rho_2]\,\mathrm{Q}^{\mathsf{s}})$ *if* $\varrho \notin \{\rho_1, \rho_2\}$.
To type the left-hand side, we first use *(C-Lift)* and then *(C-Reglam-d)*, giving us the following derivation:

$$\dfrac{\dfrac{\dfrac{TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_1), \varphi \quad \varrho \notin \mathit{frv}\,(TE) \cup \{\rho_1\} \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_1), \varphi \setminus \{\varrho\}} \qquad C_{\downarrow\varrho} \rhd \{\rho_1 \leq \rho_2\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{lift}\,[\rho_1, \rho_2]\,(\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}}) : (\mathtt{int}, \rho_2), (\varphi \setminus \{\varrho\}) \cup \{\rho_1, \rho_2\}}$$

The case is now proven by re-ordering: first, we have $TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_1), \varphi$. Now, obviously we have $\{\rho_1 \leq \rho_2\} \subseteq C$ and by Lemma 6.2.4 therefore also that $C \rhd \{\rho_1 \leq \rho_2\}$. By rule *(C-Lift)*, this makes $TE, C \vdash_c \mathtt{lift}\,[\rho_1, \rho_2]\,\mathrm{Q}^{\mathsf{s}} : (\mathtt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}$. Using the fact that $\varrho \notin \{\rho_1, \rho_2\}$, we have that $\varrho \notin \mathit{frv}\,(TE) \cup \{\rho_2\}$. Since $C \rhd \{\mathtt{d} \leq \varrho\}$, we can apply *(C-Reglam-d)* to obtain $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{lift}\,[\rho_1, \rho_2]\,\mathrm{Q}^{\mathsf{s}}) : (\mathtt{int}, \rho_2), (\varphi \cup \{\rho_1, \rho_2\}) \setminus \{\varrho\}$. And obviously we have $(\varphi \cup \{\rho_1, \rho_2\}) \setminus \{\varrho\} \subseteq (\varphi \setminus \{\varrho\}) \cup \{\rho_1, \rho_2\}$, concluding the case.

Case $\delta, \mathtt{let}\ x = (\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E^{\mu} \rightsquigarrow \delta, \mathtt{new}^{\mathsf{d}}\,\varrho.(\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E^{\mu})$ *if* $\varrho \notin \mathit{frv}\,(\mu)$.
Construct the following (part of a) type derivation for the left-hand side, using *(C-Let)* and *(C-Reglam-d)*:

$$\dfrac{\dfrac{\dfrac{TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : \mu', \varphi' \quad \varrho \notin \mathit{frv}\,(\mu', TE) \quad C \rhd \{\mathtt{d} \leq \varrho\}}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}} : \mu', \varphi' \setminus \{\varrho\}} \qquad TE + \{x \mapsto \mu'\}, C_{\downarrow\varrho} \vdash_c E^{\mu} : \mu, \varphi}{TE, C_{\downarrow\varrho} \vdash_c \mathtt{let}\ x = (\mathtt{new}^{\mathsf{d}}\,\varrho.\mathrm{Q}^{\mathsf{s}})\ \mathtt{in}\ E^{\mu} : \mu, \varphi \cup (\varphi' \setminus \{\varrho\})}$$

First, observe by Lemma 6.2.1.1, Lemma 6.2.4 and Lemma 6.2.5, that given $TE + \{x \mapsto \mu'\}, C_{\downarrow\varrho} \vdash_c E : \mu, \varphi$, we also have $TE + \{x \mapsto \mu'\}, C_{\downarrow\varrho} \cup C^{\uparrow\varrho} \vdash_c E : \mu, \varphi$. This, together with $TE, C \vdash_c \mathrm{Q}^{\mathsf{s}} : \mu', \varphi'$ and rule *(C-Let)* makes $TE, C \vdash_c \mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E : \mu, \varphi \cup \varphi'$. Since we have that $\varrho \notin frv(TE)$; $C \rhd \{\mathtt{d} \leq \varrho\}$ and also that $\varrho \notin frv(\mu)$, we can apply rule *(C-Reglam-d)* to obtain $TE, C_{\downarrow\varrho} \vdash_c \mathtt{new}^{\mathsf{d}}\, \varrho.(\mathtt{let}\ x = \mathrm{Q}^{\mathsf{s}}\ \mathtt{in}\ E) : \mu, (\varphi \cup \varphi') \setminus \{\varrho\}$. The claim now follows since $(\varphi \cup \varphi') \setminus \{\varrho\} = (\varphi \setminus \{\varrho\}) \cup (\varphi' \setminus \{\varrho\}) \subseteq \varphi \cup (\varphi' \setminus \{\varrho\})$.

Cases for the rules (6.34) through (6.40).
Proven analogous to the two previous cases: the derivation tree of the original expression is reordered to obtain a proof for the result expression.

□

6.4.3 Progress

Given a well-typed expression E (and some additional conditions), it is always possible to make a specialization step unless E is an answer.

Proposition 6.4.7 (Progress) *Suppose for a λ_2^{region}-Term E^μ that $TE, C \vdash_c E^\mu : \mu, \varphi$. Additionally, assume that $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$; $Clean(\varphi)$; $C \vdash_{wft} TE$; and for all $(\sigma, \rho') \in Ran(TE)$, it holds that $\delta(\rho') = \mathsf{d}$. Then we either have that*

1. *there exists an E' and δ' such that $\delta, E^\mu \rightsquigarrow \delta', E'$; or*
2. *E is an answer,* i.e. *of the form Q*

The requirement that the environment only contains expressions living in dynamic regions reflects the fact that during specialization we immediately reduce static expression binders such as `let` and lambda. However, since we reduce inside dynamic lambdas and `let` expressions, we cannot assume that the type environment is empty as it is usually the case in progress proofs (see for example Proposition 3.3.8 on page 39). The condition *Clean* (φ) avoids that deallocated regions or the global static or dynamic region are touched during reduction.

Proof. By rule induction on $TE, C \vdash_c E^\mu : \mu, \varphi$. As in Proposition 6.4.6, we can assume that the derivation does not end with *(C-Effect)* or *(C-Constraint)*. Otherwise, we can use exactly the same arguments to apply the induction hypothesis, using Proposition 6.4.6 to conclude. The other judgments are proven by a case distinction.

Case $TE, C \vdash_c x : \mu, \emptyset$**.** A variable x is always residual code, *i.e.* an answer.

Case $TE, C \vdash_c c\ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}$**.** Since *Clean* $(\{\rho\})$, we have that $\rho \in \mathsf{RegionVar}$. Either $\delta(\rho) = \mathtt{s}$ in which case reduction rule (6.1) applies or, if $\delta(\varrho) = \mathsf{d}$, we can use rule (6.17).

Case $TE, C \vdash_c \lambda\, x.\, E'\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\} \cup \varphi$**.** Whenever $\delta(\rho) = \mathsf{s}$ we can reduce with rule (6.2), otherwise, we have that $\delta(\rho) = \mathsf{d}$. From type rule *(C-Lam)* we also have that

$$TE + \{x \mapsto \mu_1\}, C \vdash_c E' : \mu_2, \varphi \tag{6.50}$$

where $\varphi \subseteq \varphi'$ and $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \varrho)$. Since $\delta(\varrho) = \mathsf{d}$, we have for $\mu_1 = (\tau_1, \varrho_1)$ and $\mu_2 = (\tau_2, \varrho_2)$ that $\delta(\varrho_1) = \mathsf{d}$ and $\delta(\varrho_2) = \mathsf{d}$ due to well-formedness. Moreover, it now also holds that $C \vdash_{wft} TE + \{x \mapsto \mu_1\}$ with for all $(\sigma, \rho') \in Ran\,(TE)$ that $\delta(\rho') = \mathsf{d}$.

Applying the induction hypothesis, we either have $\delta, E' \rightsquigarrow \delta', E''$, in which case we can reduce with the context rule (6.27). Alternatively, we have that E' is answer of the form Q. However, we can apply the canonical forms Lemma 6.4.5 to see that Q must be of the form R. Hence, the expression is reducible with rule (6.18).

Case $TE, C_{\downarrow P} \vdash_c \mathtt{letrec}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\, E_1\ \mathtt{at}\ \rho\ \mathtt{in}\ E_2 : \mu, \varphi \cup \varphi'$**.**
Either we have that $\delta(\rho) = \mathsf{s}$ and we apply reduction rule (6.3). Alternatively, we have $\delta(\rho) = \mathsf{d}$. By an analogous argument as in the previous case (using well-formedness of C) we can either reduce the expression with the context rule (6.28) or with rule (6.26).

Case $TE, C \vdash_c E_1 @\, E_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$**.** So, by rule *(C-App)*, we have

$$TE, C \vdash_c E_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \tag{6.51}$$

and

$$TE, C \vdash_c E_2 : \mu_1, \varphi_2 \tag{6.52}$$

Now, apply the induction hypothesis on judgment (6.51). Either $\delta, E_1 \rightsquigarrow \delta', E_1'$ which means we can use reduction rule (6.10) to reduce $E_1 @\, E_2$ to $E_1' @\, E_2$. Alternatively $E_1 = Q_1$, so then use the induction hypothesis on judgment (6.52). If we have that $\delta, E_2 \rightsquigarrow \delta', E_2'$, we can apply reduction rule (6.11) on $Q_1 @\, E_2$. If $E_2 = Q_2$, we make a case distinction for Q_1.

So, suppose first $\delta(\rho) = \mathsf{d}$, then because of judgment (6.51) and the canonical forms lemma, it must be that Q_1 is residual code R_1. However, by Lemma 6.2.8, it also holds that $C \vdash_{wft} (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)$, and since $\delta \models C$, we have that for $\mu_1 = (\tau_1, \rho_1)$ it must be that $\delta(\rho_1) = \mathsf{d}$. So, by the canonical forms lemma and judgment (6.52), it must be that Q_2 is also residual code R_2. But then we have that $E_1 @\, E_2$ is actually $R_1 @\, R_2$, allowing us to apply rule (6.22).

Alternatively, suppose $\delta(\rho) = \mathsf{s}$. Then by judgment (6.51) and the canonical forms lemma, it must be that Q_1 is a static answer, *i.e.* of the form Q_1^{s}. If Q_1^{s} is value, then again by the canonical forms lemma, Q_1^{s} is of the form $\langle \lambda\, x.\, E' \rangle_\varrho$ and reduction rule (6.6) applies. If Q_1^{s} is not a value, we can shift the static expression context with the rules (6.34), (6.37) or (6.40).

Case $TE, C \vdash_c \texttt{let}\, x = E_1\, \texttt{in}\, E_2 : \mu, \varphi \cup \varphi'$**.**
By type rule *(C-Let)* we have that $TE, C \vdash_c E_1 : \mu', \varphi'$, so we can immediately appeal to the induction hypothesis. Suppose first $\delta, E_1 \rightsquigarrow \delta', E_1'$. Then we can use reduction rule (6.13) and Item 1 applies. If E_1 is of the form Q_1, then we have the following case distinction: if Q_1 is a value V, then we can reduce with rule (6.7). If Q_1 is a static answer Q_1^{s}, but not a value, then either one of the rules (6.33), (6.36) or (6.39) applies. If Q_1 is residual code R, then we use reduction (6.25).

Case $TE, C \vdash_c f\ [\rho_1', \ldots, \rho_n']\ \texttt{at}\, \rho' : (\tau', \rho'), \{\rho, \rho'\}$**.**
Immediately reducible with rule (6.24).

Case $TE, C_{\downarrow \varrho} \vdash_c \texttt{new}\, \varrho : C^{\uparrow \varrho}.E : \mu, \varphi \setminus \{\varrho\}$**.** By Lemma 6.2.10, we can satisfy the conditions of rule (6.15).

Case $TE, C \vdash_c \texttt{lift}\, [\rho_1, \rho_2]\, E : (\texttt{int}, \rho_2), \varphi \cup \{\rho_1, \rho_2\}$**.** By type rule *(C-Lift)* we have that $TE, C \vdash_c E : (\texttt{int}, \rho_1), \varphi$ and $C \rhd \{\rho_1 \leq \rho_2\}$. Since *Clean* $(\varphi \cup \{\rho_1, \rho_2\})$, we have that $\{\rho_1, \rho_2\} \subseteq \mathsf{RegionVar}$. By application of the induction hypothesis we either have that $\delta, E \rightsquigarrow \delta', E'$, in which case we can reduce with rule (6.12), or we have that E is an answer Q. First, consider the case that $\delta(\rho_1) = \mathsf{d}$. Then obviously also $\delta(\rho_2) = \mathsf{d}$, implying by the canonical forms lemma that Q is residual code R. In this case, we can reduce with rule (6.21).

However, if $\delta(\rho_1) = \mathsf{s}$, we have by the canonical forms lemma that Q is a static answer Q^{s}. If Q^{s} is not a value, then the context rules (6.32), (6.35) or (6.38) are applicable. On the other hand, if Q^{s} is a value V, then by the canonical forms lemma it must be that $V = \langle c \rangle_{\rho_1}$. If now $\delta(\rho_2) = \mathsf{s}$, we reduce with rule (6.5), otherwise, if $\delta(\rho_2) = \mathsf{d}$, we reduce with rule (6.16).

Case $TE, C \vdash_c \langle c \rangle_\rho : (\texttt{int}, \rho), \emptyset$**.** Constant pointers are answers.

Case $TE, C \vdash_c \langle \lambda\, x.\, E \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset$**.** Function pointers are answers.

Case $TE, C_{\downarrow P} \vdash_c \texttt{letrec}\, f = \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E_1 \rangle_\rho\, \texttt{in}\, E_2 : \mu, \varphi$**.** We reduce this term with reduction rule (6.9).

Case $TE, C \vdash_c \langle \Lambda\, \varrho_1, \ldots, \varrho_n . \lambda\, x.\, E \rangle_\rho\ [\rho_1', \ldots, \rho_n']\ \texttt{at}\, \rho' : (\tau', \rho'), \{\rho, \rho'\}$**.**
Since *Clean* $(\{\rho, \rho'\})$ we have that $\{\rho, \rho'\} \subseteq \mathsf{RegionVar}$. Hence, we reduce the expression with rule (6.8).

Case $TE, C \vdash_c \texttt{new}^{\mathsf{s}}\, \varrho.E : \mu, \varphi \setminus \{\varrho\}$**.** Since *Clean* $(\varphi \setminus \{\varrho\})$, we obviously have *Clean* (φ). Take a region annotation $\delta' = \delta + \{\varrho \mapsto \mathsf{s}\}$ where $\varrho \notin (\mathit{Dom}\,(\delta) \cup \mathit{frv}\,(C))$ and $C' = \widetilde{C''}$ where $C'' = C \cup \{\varrho \leq s\}$. It is given that $\delta \models C$, hence also $\delta' \models C$. Obviously we have $\delta' \models \{\varrho \leq s\}$ too, so by Lemma 6.2.9 this implies that $\delta' \models C'$. Since $C = C'_{\downarrow \varrho}$, type rule *(C-Reglam-s)* gives us that $TE, C' \vdash_c E : \mu, \varphi$ with $\varrho \notin \mathit{frv}\,(TE, \mu)$. Now, all conditions are satisfied to apply the induction hypothesis. Either $\delta', E \rightsquigarrow \delta'', E'$ and we can reduce the left-hand side with rule (6.14) or if E is an answer, we can use rule (6.4).

Case $TE, C_{\downarrow\varrho} \vdash_c \texttt{new}^{\texttt{d}}\ \varrho.E : \mu, \varphi \setminus \{\varrho\}$**.** With almost identical arguments (using the constraint $\boldsymbol{d} \leq \varrho$ instead of $\varrho \leq \boldsymbol{s}$) as in the previous case we can apply the induction hypothesis. If E is a static answer $\mathrm{Q}^{\mathtt{s}}$, then the whole term is a static answer. If E is residual code R, then we reduce the expression with (6.20). Finally, if E can be further reduced, we can use transition rule (6.29) to further reduce the expressions.

Case $TE, C \vdash_c \texttt{let}^{\texttt{d}}\ x = R\ \texttt{in}\ E' : \mu, \varphi \cup \varphi'$**.**
So, by type rule *(C-Let-d)*, we have $TE, C \vdash_c R : (\tau', \rho'), \varphi'$ and $TE + \{x \mapsto \mu'\}, C \vdash_c E' : \mu, \varphi$. By Lemma 6.4.2 we have that $\delta(\rho') = \texttt{d}$ and hence, we can apply the induction hypothesis on E'. So, either we have $\delta, E' \rightsquigarrow \delta', E''$ and reduction rule (6.30) applies. Whenever E' is an answer Q, we consider two cases. If Q is residual code R', we can reduce with rule (6.23). Otherwise, if Q is a static answer $\mathrm{Q}^{\mathtt{s}}$, then the whole expression is a static answer as well.

Case $TE, C_{\downarrow P} \vdash_c \texttt{letrec}^{\texttt{d}}\ f = \Lambda\, \varrho_1, \ldots, \varrho_n.\lambda\, x.\ R\ \texttt{at}\ \rho\ \texttt{in}\ E : \mu, \varphi \cup \varphi'$**.** Analogous to the previous case using rules (6.19) and (6.31).

Case $TE, C \vdash_c c\ \underline{\texttt{at}}\ \varrho : (\texttt{int}, \varrho), \{\varrho\}$**.** Residual terms are immediately answers.

Remaining cases for residual terms. All immediate, since residual terms are answers.

□

6.4.4 Constraint Analysis Soundness

The constraint analysis can now be proven sound by combining type preservation and progress in the usual way:

Theorem 6.4.8 (Constraint Analysis Soundness) *Suppose E^{μ} is a λ_2^{region}-program for which $\{\,\}, C \vdash_c E^{\mu} : \mu, \varphi$. Additionally, assume that $\delta \models C$; $\delta_{\text{init}} \subseteq \delta$ and that Clean (φ), then either*

1. *there exists an answer Q, a region annotation δ', constraint set C' and effect φ' such that $\delta, E \overset{*}{\rightsquigarrow} \delta', Q$ and $\{\,\}, C' \vdash_c Q : \mu, \emptyset$ for which $\delta' \models C'$; $\varphi' \subseteq \varphi$; and $\delta \subseteq \delta'$; or*

2. *for each E' and δ' where $\delta, E \overset{*}{\rightsquigarrow} \delta', E'$, there exist E'' and δ'' with $\delta', E' \rightsquigarrow \delta'', E''$.*

Proof. Assume $\delta, E \overset{*}{\rightsquigarrow} \delta', E'$. We proceed by induction on the length of the reduction. For the base case, assume that $E = E'$. Then, by Proposition 6.4.7 (Progress) either Item 1 or 2 applies. Alternatively, we have $\delta, E \rightsquigarrow \delta'', E''$ and $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', E'$. By Proposition 6.4.6 (Type Preservation), we can apply the induction hypothesis on $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', E'$ by which Item 1 or 2 applies, proving the theorem. □

As a corollary of the soundness theorem and the canonical forms lemma, we can derive that specialization of a constraint completion always yields a residual program, provided partial evaluation terminates.

Corollary 6.4.9 (Residual Code) *Suppose e is a constraint completion with region annotation δ_c,* i.e. $\{\,\}, C \vdash_c e^{\mu} : \mu, \varphi$*;* $\delta_c \models C$ *and for all* $\rho' \in (frv\,(C, \mu) \cup \varphi)$ *holds* $\delta_c(\rho') = \mathsf{d}$.

Then, either we have non-termination of specialization or we obtain a residual program R with region annotation δ' such that $\delta_{\text{init}} + \delta_c, E \overset{*}{\rightsquigarrow} \delta', R$.

Proof. Take $\delta = \delta_{\text{init}} + \delta_c$ and $\mu = (\tau, \rho)$. The condition that for all $\rho' \in (frv\,(C, \mu) \cup \varphi)$ holds $\delta_c(\rho') = \mathsf{d}$, makes $Clean\,(\varphi)$, $\delta \models C$ and $\delta(\rho) = \mathsf{d}$. Because of constraint-analysis soundness, we either have non-termination or $\delta, e \overset{*}{\rightsquigarrow} \delta', Q$, where $\{\,\}, C' \vdash_c Q : (\tau, \varrho), \varphi'$ holds with $\delta' \models C'$; $\varphi' \subseteq \varphi$; and $\delta \subseteq \delta'$. Since $\delta'(\varrho) = \mathsf{d}$ (as a result of $\delta \subseteq \delta'$), we can apply the canonical forms lemma (Item 1), guaranteeing that Q is of the form R, *i.e.* residual code. □

A residual program resulting from the specialization of a constraint completion is again amenable to specialization since we have that $\{\,\}, C \vdash_c R : \mu, \varphi$ implies $\{\,\} \vdash_t |R| : |\mu|, |\varphi|$ and $|R| \in \lambda_{tt}^{region}$-Term.

6.5 Soundness of the Specializer

At this point, we have the nice property that the specializer of Section 6.3.2 produces a residual program $R \in \mathsf{ResidualTerm}$ for a constraint completion $e \in \lambda_C^{region}$-Term, provided the process terminates. In analogy with the results by Hatcliff and Danvy [60], full specialization correctness requires that the erasure of the residual program R is *semantically equivalent* to the erasure of the completion e.

In Chapter 5 we already developed an operational notion of equivalence for the λ_f^{region}-calculus. Additionally, an equational theory has been formulated and proven sound with respect to contextual equivalence. In this section, we exploit the theory of Chapter 5 to prove the desired soundness property, *i.e.* we show that the reductions of the specializer are equalities within the equational theory.

6.5.1 Contextual Equivalence for the λ_s^{region}-calculus

Recall that the erasure of a λ_2^{region}-Term (and hence also a λ_C^{region}-Term) is an element of the λ_s^{region}-calculus. Hence, we have to recast the definition of contextual equivalence for the λ_f^{region}-calculus (see Section 5.4.3) to the λ_s^{region}-calculus.

Instead of rephrasing all definitions and properties from Chapter 5 in terms of the objects and syntax of the λ_s^{region}-calculus, we recall that the λ_f^{region}-calculus is a conservative extension of the λ_s^{region}-calculus. Therefore, we give an indirect definition of contextual equivalence for λ_s^{region}-Terms in terms of the $\simeq$ relation (see Definition 5.4.1 on page 84) for λ_f^{region}-Terms.

Definition 6.5.1 *Suppose a type environment TE, a type with place μ and effect φ. Then we say that the λ_s^{region}-Terms e_1 and e_2 are contextual equivalent (denoted by the typed relation $\cong$) whenever their translations in the λ_f^{region}-calculus are contextually equivalent.*

Formally, $TE \rhd e_1 \cong e_2 : \mu, \varphi$ *iff* $TE \rhd [\![e_1]\!] \simeq [\![e_2]\!] : \mu, \varphi$.

Because of Proposition 5.2.10 on page 78 and Proposition 5.2.11 on page 78, this definition coincides with the more *traditional* notion of contextual equivalence in terms of adequacy.

Theorem 5.8.3 on page 104 and Theorem 5.9.4 on page 108, taken together with the above definition of contextual equivalence for λ_s^{region}-Terms, lead to the following corollary:

Corollary 6.5.1 *For any two* λ_s^{region}*-expressions* e_1 *and* e_2*, we have that*

$$TE \rhd [\![e_1]\!] \triangleq [\![e_2]\!] : \mu, \varphi \Rightarrow TE \rhd e_1 \cong e_2 : \mu, \varphi$$

This gives us sufficient material to prove specialization soundness.

6.5.2 Specialization Soundness

We now prove that the specialization rules from Figure 6.5, Figure 6.6, and Figure 6.7 are sound. However, for brevity, we do not handle polymorphism and constants. Adding these is a straightforward exercise.

We need one auxiliary lemma:

Lemma 6.5.2 *Suppose* $TE, C \vdash_c Q^\mu : \mu, \varphi$ *such that* $C \vdash_{wft} TE$ *and* $\delta \models C$ *and for all* $(\sigma, \rho') \in Ran(TE)$ *it holds that* $\delta(\rho') = \mathsf{d}$*. Then,* $TE, C \vdash_c Q^\mu : \mu, \varphi_0$ *with* $\varphi_0 \subseteq \varphi$ *and for all* $\rho' \in \varphi_0$ *it holds that* $\delta(\rho') = \mathsf{d}$.

Proof. By induction on $TE, C \vdash_c Q^\mu : \mu, \varphi$ using Lemma 6.4.1. □

The following proposition states that every erased and translated specialization step is an equality in the equational theory of Figure 5.6 on page 87.

Proposition 6.5.3 *Suppose* $TE, C \vdash_c E : \mu, \varphi$ *where* $\delta \models C$ *and* $C \vdash_{wft} TE$ *such that for all* $(\sigma, \rho') \in Ran(TE)$ *it holds that* $\delta(\rho') = \mathsf{d}$ *and also* $Clean(\varphi)$*. Then,* $\delta, E \rightsquigarrow \delta', E'$ *implies* $TE \rhd [\![|E|]\!] \triangleq [\![|E'|]\!] : \mu, \varphi$.

Proof. By Proposition 6.4.6 (Type Preservation) we have that $TE, C' \vdash_c E' : \mu, \varphi'$ and $\varphi' \subseteq \varphi$. Therefore, we also have $Clean(\varphi')$. We prove by rule induction on the definition of $\rightsquigarrow$ and consider each reduction rule from Figure 6.5, Figure 6.6 and Figure 6.7, that does not reduce a polymorphic or constant term.

Rule (6.2). Immediate by *(Eq-Beta-1)*.

Rule (6.4). By type rule *(C-Reglam-s)* we have $TE, C \vdash_c \mathtt{new}^{\mathtt{s}}\, \varrho.Q : \mu, \varphi_0 \setminus \{\varrho\}$, and hence $TE, C \vdash_c Q : \mu, \varphi_0$ such that $\varrho \notin frv(TE, \mu)$ and $C \rhd \{\mathtt{s} \leq \varrho\}$. From Lemma 6.5.2 and the assumptions, we derive that $TE, C \vdash_c \mathtt{new}^{\mathtt{s}}\, \varrho.Q : \mu, \varphi_1$ where $\varrho \notin frv(TE, \mu) \cup \varphi_1$ and $\varphi_1 \subseteq \varphi_0$. Via Proposition 6.2.12, this implies $|TE| \vdash_t^t |\mathtt{new}^{\mathtt{s}}\, \varrho.Q| : |\mu|, \varphi_1'$ with $\varphi_1' \subseteq |\varphi_1| \subseteq |\varphi_0|$ and hence also $\varrho \notin frv(|TE|, |\mu|) \cup \varphi_1'$. The equality now easily follows from the rules *(Eq-Drop)* and *(Eq-Dealloc)*.

Rule (6.6). Immediate by equational rule *(Eq-App-Cbv)*.

Rule (6.7). Follows from rule *(Eq-Beta-5)*.

Rules (6.10), (6.11), (6.13) and (6.14). Immediate applications of the induction hypothesis.

Rules (6.15), (6.18), (6.20), (6.22), (6.23) and (6.25). Here, both the left-hand and right-hand side yield identical (erased and translated) expressions and reflexivity of $\triangleq$ proves these cases.

Rules (6.27), (6.29) and (6.30). Immediate applications of the induction hypothesis.

Rule (6.33). Within *(Comp-Let)*, apply *(Eq-Drop)* via *(Eq-Comp)* on the left-hand side. Using *(Eq-Drop)* on the right-hand side and combining with *(Eq-Trans)* proves the case.

Rule (6.34). Analogous the previous case but with *(Comp-App)* instead.

Rule (6.36). Immediate from rule *(Eq-Let-Let)*.

Rule (6.37). Use rule *(Eq-App-let)* on the left-hand side, followed by *(Eq-Let-Let)*. Application of *(Eq-App-Let)* then proves the case.

□

We conclude with the soundness theorem for specialization:

Theorem 6.5.4 (Specialization Soundness) *Suppose* $\{\,\}, C \vdash_c E : \mu, \varphi$*;* $\delta \models C$*;* $\delta_{\text{init}} \subseteq \delta$*; and Clean* (φ)*. Additionally, we have* $\delta, E \overset{*}{\rightsquigarrow} \delta', Q$*. Then, it holds that* $\{\,\} \rhd |E| \cong |Q| : \mu, \varphi$

Proof. By induction the length of the specialization. The base case is trivial by reflexivity of $\triangleq$ and Corollary 6.5.1. Otherwise we have that $\delta, E \rightsquigarrow \delta'', E''$ and $\delta'', E'' \overset{*}{\rightsquigarrow} \delta', Q$. We apply Proposition 6.5.3 to obtain $\{\,\} \rhd [\![|E|]\!] \triangleq [\![|E''|]\!] : \mu, \varphi$. With Corollary 6.5.1, this yields $\{\,\} \rhd |E| \cong |E''| : \mu, \varphi$.

Using Proposition 6.4.6 (Type Preservation), we also have a C'' and φ'' such that $\{\,\}, C'' \vdash_c E'' : \mu, \varphi''$ where $\delta \subseteq \delta''$; $C \subseteq C''$; $\varphi'' \subseteq \varphi$ and $\delta'' \models C''$. The induction hypothesis now yields $\{\,\} \rhd |E''| \cong |Q| : \mu, \varphi$ and we prove the claim by transitivity of contextual equivalence (see Lemma 5.4.6 on page 85). □

6.6 Related Work

Dussart, Henglein and Mossin [36, 73] specify a polymorphic binding-time analysis with polymorphic recursion as an ad-hoc extension of a monomorphic analysis. Their framework builds on top of a simply-typed lambda calculus. It includes a powerful notion of

binding-time subtyping, where only the binding times change, but not the structure of the type itself. Dependencies between as yet unknown binding times are also expressed by means of qualified type schemes. Their work contains constraint simplification rules and a polynomial-time fixpoint algorithm. They do not have correctness results for the analysis (except for the first-order case [73], which has a problem due to the way name generation is formalized in their semantics). In contrast, our base language is polymorphically typed and we do not use subtyping since it may lead to code duplication. Name generation problems do not occur since our approach is entirely operational.

Heldal and Hughes [62, 64] extend the work by Dussart, Henglein and Mossin by considering a polymorphic base language and by representing coercions in their type language. While our work does not address subtyping and coercions, it provides a correctness proof of the BTA and specialization.

Glynn, Stuckey, Sulzmann and Søndergaard [46] generalize a polymorphic binding-time analysis to polymorphic programs, by encoding binding-time subtyping constraints as boolean formulas. This is done within the HM(X)-framework [113], in order to exploit fast boolean constraint solvers. Their approach lacks a corresponding specialization semantics and soundness proof.

Thiemann [139] considers another variant of the region calculus [136] as the basis of a binding-time analysis for ML with references. In that work, the emphasis is on the imperative aspects of ML. Both language and analysis are monomorphic and the correctness proof is indirect through a translation to a pure lambda calculus.

The Dependency Core Calculus of Abadi et al. [1] is an extension of Moggi's computational lambda calculus with dependencies. It can be specialized to a monomorphic binding-time analysis. Continuing that work, Banerjee and others [9] define a polymorphic lambda calculus that provides a denotational proof for the soundness of the region calculus. However, they do not give a direct connection between binding times and regions.

Our syntactic soundness proof for the constraint analysis is based on work elaborated in the previous chapters (see also [18, 68]). There, we also discuss other syntactic approaches to the type soundness result.

Mogensen [105] was the first to specify a binding-time analysis for a language with ML-style polymorphism. However, except for the polyvariance introduced by type-polymorphism, *i.e.* different type instantiations allow for different binding-time descriptions, the analysis is monovariant. Also, Mogensen's binding-time analysis is not directly related (or proven correct) to an actual specializer.

The first use of effects in a binding-time analysis seems to be in a paper by Consel et al. [26]. Starting from the simply-typed lambda calculus, they annotate function arrows with boolean functions that determine the binding time of the result of the function from the binding times of the function's parameters. They prove a result comparable with subject reduction, give an inference algorithm, and prove its soundness and completeness. Their calculus relies on binding-time polyvariance to allow for a modular binding-time analysis. However, the paper does not consider recursion or polymorphism and there is no connection to regions. The authors suggest an extension to ML-style polymorphism by expanding `let`-expressions. However, this defeats their goal of modularity.

Hornof and Noyé [75] define a binding-time analysis for imperative languages that is flow, context, and return sensitive. Their context sensitivity corresponds to polyvariant binding time analysis, the other sensitivities are not applicable in our framework. However, their approach to polyvariance relies on duplicating function definitions and their corresponding data flow equations. It is not based on a notion of polymorphism. There is no formal correctness proof.

Earlier papers dealing with polyvariant analysis rely on abstract interpretation and use ad-hoc techniques to guarantee termination of the analysis [16,20,22,23,43,124,145]. Some of them do not formally relate the analysis with the specialization phase and some only cater for first-order functional languages.

6.7 Chapter Summary

In Section 6.2, we designed a binding-time analysis for offline partial evaluation of ML-like languages as a constraint analysis on top of region inference. We introduced the λ_C^{region}-calculus as an extension of the λ_{tt}^{region}-calculus in Section 6.2.1. After having defined some new semantic objects in Section 6.2.2, most notably constraints between regions, the constraint analysis was formulated as an enriched type and effect system for the λ_C^{region}-calculus in Section 6.2.3. Several properties and a relation with the λ_s^{region}-calculus were discussed in Section 6.2.4 in order to formulate a *constraint completion* in Section 6.2.5.

Such a constraint completion forms the input of the partial evaluator as it was implemented with the small-step reduction rules of Section 6.3.2. The specialization rules operate on a two-level extension of the λ_s^{region}-calculus as formulated in Section 6.3.1. Two examples were given in Section 6.3.3. Section 6.3 ended with an extension of the static semantics for the λ_2^{region}-calculus in Section 6.3.5.

The second part of the chapter dealt with the soundness of the constraint analysis (Section 6.4) and of the specializer (Section 6.5). The former property was proven analogous to the syntactic type soundness schema of Section 2.2.3. After elaborating some auxiliary lemmas in Section 6.4.1, a type preservation and progress property were proven in Section 6.4.2 and Section 6.4.3 respectively. As usual, taken together, they yield a soundness property, in Section 6.4.4 fortified by the fact that the result of a specialization is always residual code.

In Section 6.5 we proved specialization soundness making use of an equational theory from Chapter 5, which was already proven sound with respect to Morris-style contextual equivalence.

7 CONCLUSIONS

7.1 Assessment

As suggested by the title, the primary goal of this thesis is the specification of a specializer or partial evaluator for ML-style polymorphic programming languages. An important aspect of such a tool is to prove the specification *correct* in the sense that it yields a result that does not violate the semantics of the transformed program. In general, program transforming tools are of limited value if they are not based on a formally correct model.

Pursuing these goals, we tributed to three areas of programming language research as we sketched them in Section 1.1.

Operational Methods in Semantics

This thesis makes an important case for the use of syntactic operational techniques in the formal correctness of program transformation tools. Such methods to reason about programs and programming language semantics are becoming increasingly popular since they require relative little mathematical background. Moreover, their close relation to an actual implementation is appealing to compiler and language tool designers. Throughout this thesis, we illustrated how these methods scale to complex typed region calculi on many levels, be them type soundness properties or operational equivalence models. In particular, we have applied operational techniques since we believe that denotational models for such systems have a mathematical complexity that prohibit an intuitive understanding. A good example of denotational methods applied to region calculi can be found in the work by Banerjee et al. [9].

As observed by Stoy [133], operational methods are especially useful to reason about *program transformation* correctness. This is motivated by the fact that such models are close to an actual implementation, in contrast with denotational techniques that refer to an abstract mathematical model that does not necessarily has much to do with a compiler or program specializer.

Although a denotational semantics can usually be read as an interpretative definition of the object language, modern partial evaluators are usually not specified in an interpretative way. Moreover, it is known that the formal correctness of efficient program generating combinators from a denotational description needs a significant additional formal step [141]. Hence, the operational approach fits well for the design of a program specializer.

Semantic Results for the Region Calculus

The application of operational techniques to region calculi in this thesis yields several new results. First, we formulated a new, concise and elegant storeless execution semantics in Plotkin's operational style [122]. This gave rise to a straightforward type soundness proof, contrasted with the monolithic approach of the original safety proof [150].

Second, we extended the storeless model with an explicitely passed store and proved type soundness for an imperative region calculus. Interestingly, relating the small-step semantics — whether it had a store or not — to the Tofte-Talpin style big-step semantics was more technical. Still, inductive proofs would have been sufficient here as well, as long as no destructive update operators were added. Although we did not handle the addition of store operations, we used co-inductive methods to make them straightforward to add. Moreover, co-induction was only required for one auxiliary lemma.

Third, we defined an equational theory for a region calculus similar to that of Dal Zilio and Gordon [160]. The equational rules are somewhat more general since we employ a more flexible notion of typing. Additionally, we give a direct correctness proof. The latter was achieved by introducing an appropriate notion of bisimilarity based on a labeled transition system for the region calculus. We formulated such a notion with a minimal amount of observations, but still equivalent to the more classical Morris-style contextual equivalence. By showing that the equational theory is sound with respect to bisimilarity, we designed a proof mechanism for proving semantical equivalences within the region calculus.

The employed techniques were adapted from Gordon's work on bisimilarity in functional programming languages [49,50]. Bisimilarity turns out to be a suitable proof mechanism for showing contextual equivalences in deterministic stateless program calculi. The addition of type, effect and region polymorphism lead to new technical problems, but did not hinder the use of bisimulation techniques.

In their seminal paper on region inference [150], Tofte and Talpin prove more than just type soundness. They also show that region reconstruction as it is used in the ML-kit [148] is *correct*. Roughly speaking, they prove that a region-annotated expression is semantically equivalent to the un-annotated version. Although the equational theory of Chapter 5 is not strong enough to prove a similar property, the bisimilarity notion can be used to prove that any region-annotated program is contextually equivalent to the same program with only one global region and no local region allocations. Then, we conjecture that it is straightforward to show an equivalence (in some sense to be defined) between the program with the one global region and its original version. Perhaps more importantly, the equational theory explicitly shows that region operations do not affect the value semantics of an expression. In other words, the program semantics is not influenced by region-based memory management.

Although these syntactic operational techniques were applied on Tofte-Talpin style region calculi, we expect them to be of more general value and be usable to perform correctness proofs for many other existing region calculi [42, 54, 94, 155].

Program Specialization

The work on region calculi, which forms a large part of this thesis, formed the foundations for an offline partial evaluator for ML-like languages. This idea has been triggered by some important observations. It is generally a good property for an offline partial evaluation system to modularize the problem of the binding-time analysis by separating the flow analysis from the binding-time annotation phase [140]. Region inference performs a fairly accurate data-flow dependency analysis where regions can be interpreted as abstract program points. Moreover, region inference was explicitely designed to do memory management for Standard ML and has been implemented in the ML-kit as such. The polymorphic region calculus blends particularly well with Hindley-Milner type polymorphism.

In the light of these facts, the step towards a partial evaluator for an ML-like language is not surprising. The flow-analysis part of the BTA is entirely covered by region inference. Only the binding-time annotation phase has to be added, which in our case boils down to a constraint analysis. Due to polymorphic recursion with regions, binding times cannot be calculated statically and hence, the constraints are solved at specialization time. Extending the region calculus into a two-level version and designing an appropriate specialization semantics was a technical task. Correctness proofs for our novel specialization mechanism followed from our results for the region calculus: the constraint analysis is proven sound using standard syntactic techniques. The specializer is proven sound using the equational theory for the region calculus.

One attraction of using region inference for a binding-time analysis is the coinciding goal of region separation in a memory management and binding-time separation in a specialization context. This makes it natural to reuse existing region reconstruction algorithms [11, 146] for program specialization.

Unlike the work by Heldal and Hughes [62, 63] and earlier results by Dussart, Henglein and Mossin [36, 73], we do not consider higher-order binding-time coercions and subtyping. This is primarily motivated by the fact that such coercions may cause uncontrollable code duplication. Our current scheme does not entirely prevent this either, but there are standard remedies for these as we will discuss in the next section.

7.2 Future Work

The above mentioned results of this dissertation leave room for improvement and extensions. We first discuss work to be done on the semantics of region calculi. Then, we examine the necessary steps to make the program specialization mechanism of this thesis amenable for a real system. Finally, we discuss how our connection between binding times and regions suggests a method to optimize low-level systems programs.

Semantics and Region Calculi

Although the equational theory developed in Chapter 5 is sufficient to prove the soundness of a program specializer for a side-effect free subset of Standard ML, it is desirable to extend it to handle exceptions and a destructive updatable store. Proving operational

program equivalences for calculi with local state is known to be hard [118, 135]. In particular, to date, no bisimilarity notion has been developed for such a calculus and it is not known whether bisimilarity scales to imperative extensions [135].

The λ_i^{region}-calculus of Chapter 4 provides for local state already. Since it is formulated with a small-step transition semantics, it is a natural strategy to tackle the open problem depicted above by reworking the development of Chapter 5 in the light of the λ_i^{region}-calculus. The fact that locations are first-class values in the λ_i^{region}-calculus is of particular interest [118].

Although our thesis focuses on the use of operational methods as we want to prove a program transformer correct, it is certainly worth to examine other semantical models for the region calculus. To the best of our knowledge, no denotational semantics has been constructed for a region calculus, except for an indirect model of an extended polymorphic lambda calculus [9]. The alternative denotational track deserves attention since it might provide insights of a more general nature.

Program Specialization

In Chapter 6, we have given the first steps towards a powerful offline partial evaluator for a real programming language like Standard ML. However, there are several issues that still have to be explored before an actual tool can be developed. We list the main ones here:

- Although region inference is well-explored and two algorithms currently exist [11, 146], we still have to develop a constraint analysis algorithm. In particular, we have to address the question whether a unique minimal completion exists and develop an efficient (polynomial-time) algorithm for it.

- In the previous section, we already mentioned that the current specialization model still allows for code duplication. In many practical cases the specializer may not terminate because of recursively defined functions. In particular, static calls under dynamic control are known to cause infinite specialization.

  A standard solution to this problem is to *memoize* the function's binding-time description [82, 138] and to insert so-called dynamic program points. The current formalization does not cover memoization: it either unrolls recursive functions or keeps them with a specialized body. The presence of polymorphic binding times (in the form of polymorphic regions) requires an extended form of memoization. Next to memoizing the static values of a function definition, we also have to memoize the binding-time pattern of the abstracted region variables. Whenever the function contains dead code, it is possible that this "region pattern" does not change in the same way as the static memoized values do.

- Modern specializers employ a *program generator generator*, often abbreviated as *cogen*[1]. The cogen-approach [15, 137] to offline partial evaluation is essential for

[1]The word *cogen* is historical and stands for "compiler generator". It is really a special instance of a "program generator generator".

the effective specialization of ML-like programming languages due to type-encoding problems for a self-applicable interpretative partial evaluator [13, 88]. A cogen for our specialization technique is particular challenging since it may have to resolve statically unknown binding times. Since it is desirable to code a cogen for SML in SML itself, the presence of binding times at runtime may cause tagging. If the cogen does not need to be implemented in ML itself, a richer dependent type system may solve the problem. Whenever a solution within ML is desirable, it should be examined whether for instance *ML-functors* (a parameter module concept for SML [93]) can be used to statically verify specialization-time binding-time dependencies.

- Regions and effects have been used previously to specialize side-effects on reference cells [139] with destructive updates. The imperative extension of the region calculus from Chapter 4 for ML-style reference cells could provide a basis for specialization-time reductions of store operations. However, such additions are not straightforward in practice [144]. Also, it will complicate the soundness proof of the specializer as we explained earlier. Similar considerations have to be made for exceptions if we do not want to defer them all to runtime.

Low-level Systems Programming

In recent years, there is a tendency towards manual binding-time annotations (see for example Meta-ML [107, 134]) or manual region annotations (see for example Cyclone [54, 80]). The idea is to transfer more binding-time and memory management control to the user.

This approach seems particular attractive for safe low-level systems programming. A prominent example of such a language is Cyclone, a low-level systems language with high-level abstraction mechanisms built-in. In particular, it provides for manually inserted regions and region operations, very much like `malloc` and `free` in C. However, unlike these C-constructs, Cyclone *region type-checks* the program, *i.e.* it statically verifies whether there is an attempt to dereference a dangling pointer. The advantage of this approach over region inference as it is implemented in the ML-kit, is that the user retains direct control over memory management and, hence has the ability to improve memory foot prints.

Following the design ideas of our region-based program specializer, we could allow a programmer to annotate the manually inserted regions with extra information, for instance binding times. An optimizing compiler could implement a binding-time or constraint analysis and use the annotations to perform aggressive program optimizations. In other words, the systems programmer would be encouraged to think in terms of *data flow* and tune it for both memory usage and optimizations.

7.3 Concluding Remarks

Our thesis formulates basic building blocks for the design of a powerful program specializer for the Standard ML functional programming language. Its potential power is

due to its region-based nature, a technique originally developed for the static memory management of dynamic data structures. The specialization design is entirely proven correct in a syntactic operational fashion. Along the way, we presented a semantical basis for reasoning about types and equivalences for a widely used region calculus.

Bibliography

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In Aiken [5], pages 147–160.

[2] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, 1990.

[3] Samson Abramsky and Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[4] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.

[5] Alexander Aiken, editor. *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, Texas, USA, January 1999. ACM Press.

[6] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the 1995 Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, CA, USA, June 1995. ACM Press.

[7] Alexander Aiken and David Gay. Memory management with explicit regions. In Keith Cooper, editor, *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, pages 313–324, Montreal, Canada, June 1998. ACM. Volume 33(5) of SIGPLAN Notices.

[8] Lennart Augustsson. Partial evaluation in aircraft crew planning. In Charles Consel, editor, *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 127–136, Amsterdam, The Netherlands, June 1997. ACM Press.

[9] Anindya Banerjee, Nevin Heintze, and Jon Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 1999 IEEE Symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, July 1999. IEEE Computer Society Press.

[10] Lennart Beckman, Anders Haraldsson, Östen Oskarsson, and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[11] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 58:299–392, 2001.

[12] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In POPL1996 [123], pages 171–183.

[13] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.

[14] Anders Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, May 1993.

[15] Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–10, Orlando, Fla., June 1994. University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science.

[16] Mikhail Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In Schmidt [126], pages 59–65.

[17] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In Hanne Riis Nielson, editor, *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 155–165, London, England, January 2001. ACM Press.

[18] Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002.

[19] Morten Christiansen, Fritz Henglein, Henning Niss, and Per Velschow. Safe region-based memory management for objects. Technical report, DIKU, University of Copenhagen, October 1998.

[20] Charles Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université de Paris VI, Paris, France, June 1989.

[21] Charles Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, 1990. ACM Press.

[22] Charles Consel. Polyvariant binding-time analysis for applicative languages. In Schmidt [126], pages 66–77.

[23] Charles Consel. A tour of Schism. In Schmidt [126], pages 134–154.

[24] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [78], pages 496–519.

[25] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[26] Charles Consel, Pierre Jouvelot, and Peter Ørbæk. Separate polyvariant binding-time reconstruction. Technical Report CRI-A/261, Ecole des Mines, Paris, October 1994.

[27] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[28] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In Aiken [5], pages 262–275.

[29] Roy Crole and Andrew Gordon. A sound metalogical semantics for input/output. In *Computer Science Logic, CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 339–353, Kazimierz, Poland, 1995. Springer-Verlag.

[30] Haskell Curry and Robert Feys. *Combinatory Logic*, volume I. North Holland, 1958.

[31] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 1982 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[32] Olivier Danvy. Type-directed partial evaluation. In POPL1996 [123], pages 242–257.

[33] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[34] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In PLDI2001 [120], pages 59–69.

[35] Dirk Dussart, Rogardt Heldal, and John Hughes. Module-sensitive program specialisation. In PLDI1997 [119], pages 206–214.

[36] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proceedings of the 1995 International Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag.

[37] Andrei P. Ershov. Mixed computation: Potential applications and problems for study. In *Mathematical Logic Methods in AI Problems and Systematic Programming, Part 1*, pages 26–55. Vil'nyus, USSR, 1980. (In Russian).

[38] Matthias Felleisen and Daniel Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[39] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

[40] Robert Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[41] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[42] David Gay and Alexander Aiken. Language support for regions. In PLDI2001 [120], pages 70–80.

[43] Marc Gengler and Bernhard Rytz. A polyvariant binding time analysis handling partially known values. In *Proceedings of the 2nd Workshop on Static Analysis*, volume 81–82 of *Bigre Journal*, pages 322–330, Rennes, France, 1992. IRISA.

[44] David Gifford and John Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[45] Robert Glück, Olivier Danvy, and Peter Thiemann, editors. *Dagstuhl Seminar 9607: Partial Evaluation*, IBFI GmbH, Schloß Dagstuhl, D-66687 Wadern, Germany, February 1996. Seminar report 134.

[46] Kevin Glynn, Peter Stuckey, Martin Sulzmann, and Harald Søndergaard. Boolean constraints for binding-time analysis. In *Programs as Data Objects II*, number 2053 in Lecture Notes in Computer Science, pages 39–62, Aarhus, Denmark, May 2001. Springer-Verlag.

[47] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[48] Andrew Gordon. *Functional Programming and Input/Output*. Cambridge University Press, 1994.

[49] Andrew Gordon. Operational equivalences for untyped and polymorphic object calculi. In Gordon and Pitts [51], pages 9–54.

[50] Andrew Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1-2):5–47, October 1999.

[51] Andrew Gordon and Andrew Pitts, editors. *Higher Order Operational Techniques in Semantics (HOOTS)*, Publications of the Newton Institute. Cambridge University Press, 1998.

[52] Andrew Gordon and Andrew Pitts, editors. *Higher Order Operational Techniques in Semantics (HOOTS)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, October 1999. Elsevier Science.

[53] Andrew Gordon and Gareth Rees. Bisimilarity for a first-order calculus of objects with subtyping. In POPL1996 [123], pages 386–395.

[54] Dan Grossman, Greg Morrisett, Trevor Jim, Mike Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[55] Carl Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, Cambridge, MA, 1992.

[56] Carl Gunter and Dana Scott. *Semantic Domains*, volume B of *Handbook of Theoretical Computer Science*, chapter 12. Elsevier Science Publishers, Amsterdam, 1990.

[57] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [58].

[58] Robert Harper. A note on: "A simplified account of polymorphic references". *Information Processing Letters*, 57(1):15–16, January 1996. See also [57].

[59] John Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In Hatcliff et al. [61], pages 20–82.

[60] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

[61] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, 1999. Springer-Verlag.

[62] Rogardt Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator*. PhD thesis, Chalmers University of Technology, April 2001.

[63] Rogardt Heldal and John Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science*, 248:99–145, 2000.

[64] Rogardt Heldal and John Hughes. Binding-time analysis for polymorphic types. In *PSI-01: Andrei Ershov Fourth International Conference, Perspectives of System Informatics*, number 2244 in Lecture Notes in Computer Science, pages 191–204, Novosibirsk, Russia, July 2001. Springer-Verlag.

[65] Simon Helsen. An equational theory for a region calculus - revised. Technical Report 179, Institut für Informatik, University of Freiburg, Germany, October 2002. (submitted for publication).

[66] Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In J. Hsiang and A. Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998.

[67] Simon Helsen and Peter Thiemann. Fragmental specialization. In Walid Taha, editor, *Semantics, Applications and Implementation of Program Generation (SAIG'00)*, number 1927 in Lecture Notes in Computer Science, pages 51–71, Montreal, Canada, September 2000. Springer-Verlag.

[68] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *ACM Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*, pages 1–20, Montreal, Canada, September 2000. Elsevier Science.

[69] Simon Helsen and Peter Thiemann. Polymorphic specialization for ML. Technical Report 173, Institut für Informatik, University of Freiburg, Germany, April 2002. (submitted for publication).

[70] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [78], pages 448–472.

[71] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[72] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In Harald Søndergaard, editor, *Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001.

[73] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.

[74] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[75] Luke Hornof and Jacques Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2):3–27, October 2000.

[76] Douglas Howe. Equality in lazy computation systems. In LICS1989 [92], pages 198–203.

[77] Douglas Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[78] John Hughes, editor. *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, MA, 1991. Springer-Verlag.

[79] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 183–215, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.

[80] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[81] Neil Jones. An introduction to partial evaluation. *Computing Surveys*, 28(3):480–504, September 1996.

[82] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[83] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proc. 1991 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991. ACM Press.

[84] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[85] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[86] Peter Landin. An abstract machine for designers of computing languages. In *IFIP Congress*, pages 438–439. North-Holland, 1965.

[87] Søren Lassen. Relational reasoning about contexts. In Gordon and Pitts [51], pages 91–136.

[88] John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.

[89] Julia Lawall. Faster fourier transforms via automatic program specialization. In Hatcliff et al. [61], pages 338–355.

[90] Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.

[91] Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of the Theoretical Aspects of Computer Software*, number 1281 in Lecture Notes in Computer Science, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.

[92] *Proceedings of the 1989 IEEE Symposium on Logic in Computer Science*, Pacific Grove, CA, June 1989. IEEE Computer Society Press.

[93] David MacQueen. Modules for standard ML. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1985.

[94] Henning Makholm. Region-based memory management in Prolog. Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, March 2000.

[95] Ian Mason and Carolyn Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 1989 International EATCS Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 1989.

[96] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–328, July 1991.

[97] Ian Mason and Carolyn Talcott. Inferring the equivalence of functional programs that mutate data. In *Proceedings of the 1992 IEEE Symposium on Logic in Computer Science*, pages 186–197. IEEE Computer Society Press, June 1992.

[98] Albert Meyer and Stravos Cosmadakis. Semantical paradigms: notes for an invited lecture. In *Proceedings of the 1988 IEEE Symposium on Logic in Computer Science*, pages 236–253, Edinburgh, Scotland, 1988.

[99] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1978.

[100] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[101] Robin Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[102] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

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

[104] John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[105] Torben Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, number 351,352 in Lecture Notes in Computer Science, pages II, 298–312, Barcelona, Spain, March 1989. Springer-Verlag.

[106] Eugenio Moggi. Computational lambda-calculus and monads. In LICS1989 [92], pages 14–23.

[107] Eugenio Moggi, Walid Taha, Zino El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In Doaitse Swierstra, editor, *Proceedings of the 1999 European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, Amsterdam, The Netherlands, April 1999. Springer-Verlag.

[108] James Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT Press, December 1968.

[109] Flemming Nielson. A bibliography on abstract interpretation. Technical Report 5, ACM SIGPLAN Notices, 1986.

[110] Flemming Nielson and Hanne Riis Nielson. Automatic binding-time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988.

[111] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[112] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[113] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[114] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In Neil Jones, editor, *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[115] Andrew Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Technical Report BRICS Notes Series NS-94-5, Aarhus University, 1994.

[116] Andrew Pitts. Reasoning about local variables with operationally-based logical relations. In *Proceedings of the 1996 IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[117] Andrew Pitts. Operationally-based theories of program equivalence. In Andrew M. Pitts and Peter Dybjer, editors, *Semantics and Logics of Computation*, pages 241–298. Cambridge University Press, 1997.

[118] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In Gordon and Pitts [51], pages 309–326.

[119] *Proceedings of the 1997 Conference on Programming Language Design and Implementation*, Las Vegas, NV, USA, June 1997. ACM Press.

[120] *Proceedings of the 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001. ACM Press.

[121] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

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

[123] *Proceedings of the 1996 ACM SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, Fla., January 1996. ACM Press.

[124] Bernhard Rytz and Marc Gengler. A polyvariant binding time analysis. In Charles Consel, editor, *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.

[125] Davide Sangiorgi. On the bisimulation proof method. Technical Report ECS-LFCS-94-299, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1993.

[126] David Schmidt, editor. *Proceedings of the 1993 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[127] Dana Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. Unpublished manuscript, see [128], 1969.

[128] Dana Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Originally unpublished manuscript, see [127].

[129] Dana Scott and Christopher Strachey. Towards a mathematical model for computer languages. In J. Fox, editor, *Symposium on Computers and Automata*, pages 19–46, New York, 1971. Polytechnic Institute of Brooklyn Press.

[130] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In PLDI1997 [119], pages 215–225.

[131] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

[132] Guy Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2nd edition, 1990.

[133] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981.

[134] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 2000.

[135] Caroline Talcott. Reasoning about functions with effects. In Gordon and Pitts [51], pages 346–390.

[136] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–272, July 1992.

[137] Peter Thiemann. Cogen in six lines. In Kent Dybvig, editor, *Proceedings of the 1996 International Conference on Functional Programming*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.

[138] Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, number 1140 in Lecture Notes in Computer Science, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.

[139] Peter Thiemann. Correctness of a region-based binding-time analysis. In *Proceedings of the 1997 Conference on Mathematical Foundations of Programming Semantics*, volume 6 of *Electronic Notes in Theoretical Computer Science*, page 26, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[140] Peter Thiemann. A unified framework for binding-time analysis. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 742–756, Lille, France, April 1997. Springer-Verlag.

[141] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, September 1999.

[142] Peter Thiemann. *The PGG System—User Manual*. Universität Freiburg, Freiburg, Germany, March 2000. Available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.

[143] Peter Thiemann. Enforcing safety properties using type specialization. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming*, Lecture Notes in Computer Science, Genova, Italy, April 2001. Springer-Verlag.

[144] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen, April 1997.

[145] Peter Thiemann and Michael Sperber. Polyvariant expansion and compiler generators. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, number 1181 in Lecture Notes in Computer Science, pages 285–296, Novosibirsk, Russia, June 1996. Springer-Verlag.

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

[147] Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[148] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olensen, and Peter Sestoft. *Programming with Regions in the ML Kit, Version 4.0.0*, September 2001. Available from `http://www.it.edu/research/mlkit/dist/mlkit-4.0.0.pdf`.

[149] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 1994 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 188–201, Portland, OR, January 1994. ACM Press.

[150] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[151] Valentin F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

[152] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science—Formal Models and Semantics*, volume B. Elsevier Science Publishers, 1990.

[153] Per Velschow and Morten Voetman. Region-based memory management in Java. Master's thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, May 1998.

[154] David Walker, Carl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

[155] David Walker and Kevin Watkins. On regions and linear types. In Xavier Leroy, editor, *Proceedings of the 2001 International Conference on Functional Programming*, pages 181–192, Florence, Italy, September 2001. ACM Press, New York.

[156] Paul Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, 1999.

[157] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[158] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[159] Silvano Dal Zilio and Andrew Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of the 2000 Conference on Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*, pages 1–20, 2000.

[160] Silvano Dal Zilio and Andrew Gordon. Region analysis and a pi-calculus with groups. *Journal of Functional Programming*, 12(3):229–292, May 2002.

Index

Symbols

A

B

M

N

O

P

Q

R

S

T

U

V

W