CAPSULES AND NON-WELL-FOUNDED COMPUTATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jean-Baptiste Jeannin

August 2013

© Jean-Baptiste Jeannin 2013

ALL RIGHTS RESERVED

CAPSULES AND

NON-WELL-FOUNDED COMPUTATION

Jean-Baptiste Jeannin, Ph.D.

Cornell University 2013

Several recent programming languages, for example Python, C# and Javascript, are not strictly imperative or functional, but provide features from both paradigms. In this dissertation, we introduce *capsules*, an algebraic representation of the state of a computation in such higher-order functional and imperative languages. A capsule is essentially a finite coalgebraic representation of a regular closed λ-coterm. One can give an operational semantics based on capsules for a higher-order programming language with functional and imperative features, including mutable bindings. Static (lexical) scoping is captured purely algebraically without stacks, heaps, or closures. Definitions and applications of functions, including recursive functions, are typable with simple types, yet the language is Turing complete. Recursive functions are represented directly as capsules without the need for fixpoint combinators. In this disseration we precisely compare a capsule-based semantics to a closure-based semantics. We also study a formulation of separation logic using capsules, prove soundness of the frame rule in this context and investigate alternative formulations with weaker side conditions.

Capsules, by their coinductive nature, also provide a clean internal representation of coalgebraic datatypes. Recursive functions defined on a such coalgebraic datatypes may not converge if there are cycles in the input — that is, if the input object is not well-founded. Even so, there is often a useful solution. Unfortunately, current functional programming languages provide no support for specifying alternative solution methods. There are numerous examples in which it would be useful to do so: free variables, α-conversion, and substitution in infinitary lambda-terms; halting probabilities and expected running times of probabilistic protocols; abstract interpretation; and constructions involving finite automata. In each case the function would diverge under the standard semantics of recursion. In this dissertation, we first prove some theoretical results characterizing the well-founded case. We then propose programming language constructs that allow the specification of alternative solutions and methods to compute them. Finally, we introduce CoCaml, a functional programming language extending OCaml that implements those constructs and allows the programmer to define functions on coinductive datatypes parameterized by an equation solver.

Biographical Sketch

Jean-Baptiste Jeannin, son of Benoît and Isabelle Jeannin, grew up in Paris, France and London, United Kingdom. After graduating high school from the Lycée Stanislas in Paris, France in 2002, he studied in the Classes Préparatoires MPSI and MP* of the Lycée Stanislas. He entered École polytechnique in 2004 where he specialized in computer science. Under the guidance of Gilles Dowek, François Pottier, Dale Miller and Laurent Mauborgne, he got interested in Programming Languages. In the Spring of 2007 he interned under César Muñoz at the National Institute of Aerospace in Langley, Virginia.

In August 2007, Jean-Baptiste entered the Master of Engineering program in Computer Science at Cornell University. After starting a project under the guidance of Dexter Kozen, he soon realized that he wanted to do research and applied to the Ph.D. program. He nevertheless graduated with a Master of Engineering in May 2008, before entering the Ph.D. program in January 2009. He continued working with Dexter Kozen, and interned at Facebook under Yoann Padioleau, Microsoft Research Redmond under Nikhil Swamy and Microsoft Research Bangalore (India) under Prasad Naldurg. He visited Alexandra Silva in Amsterdam and Nijmegen, The Netherlands. Jean-Baptiste graduated with a Ph.D. in Computer Science and a minor in Mechanical and Aerospace Engineering in August 2013.

À mes parents

À Bénédicte et Raphaëlle

Acknowledgements

This thesis would never have been possible without my advisor, Dexter. I learnt so much by your side! You are an amazing teacher, because you really care about teaching and never get tired of explaining the same thing again and again. What always impresses me the most is your constant ability to give me a mini-lecture about any topic about which I had asked a question, without preparing or looking up anything. You also have an extremely balanced life, between family, research, sports and music, and I hope to achieve such balance in my life too. Thank you for being such an example!

I would also like to thank the other members of my committee, Ashutosh, Hadas and Nate, for always being available when I wanted to chat or ask more general questions about my Ph.D. or my future.

Alexandra, you have become much more than a co-author but a real friend. You arrived as a postdoc with Dexter at a moment when I needed some encouragements and new research directions, and you sure provided that! Thank you so much for always being there to listen, be it in Ithaca, Amsterdam or Nijmegen, and for hosting me in the Netherlands twice. I hope to keep you as a friend and continue collaborating!

Andrew, by teaching CS 6110 in the Fall of 2007, ingrained my love for program-

ming languages. Without such a good teacher, I might not have continued in that field. Bob, thank you for your advice and encouragements, and your remarks on the history of programming languages and theorem proving.

The Programming Languages Discussion Groups has become the place of many interesting and enriching discussions over the years. Thanks in particular to Andrew, Basu, Mark, Mike, Nate, Owen and Ross for your organizing skills, remarks and discussions. My officemates Kostas, Stefano, Wenzel and Yao have always been there for interesting discussions about Graphics, Systems, Artificial Intellingence, Programming Languages or life in general. Thanks to all the players of the Computer Science Hockey games for making me (re)discover this great game, and to their czars Jeff and Owen.

Becky and Stephanie, thank you so much for always being there for us students, never lose patience and always try to work out problems. And also for fun conversations! You are doing an amazing job every single day and I am so grateful. Thanks also to Kelly, Randy and Michelle for putting up with my hate of administrative tasks.

Yoann, Haiping, Nik and Prasad, thank you for mentoring me during my different internships at Facebook, and Microsoft Research. You have helped me a lot with looking at different research problems and getting a broader view of the field. Thanks also to all the researchers I discussed with at conferences, in particular Matthias Felleisen, Jean-Christophe Filliâtre, Neelakantan Krishnaswami, François Pottier, and on trips to the Netherlands, in particular Helle Hansen, Stefan Milius and Jan Rutten. Thanks also to Hersh Mehta for help with the implementation of CoCaml.

The Cornell Catholic Community has played a very important role in my life in those years at Cornell, both spiritually and socially. Thank you especially to Father

Dan, Father Bob, Father Carsten and Joe Mazzawi for running it and making it the place that it is. It has also been a great place to make friends. The Souldiers Community and its Tuesday night prayers and discussions have been an amazing source of support and growth. Special thanks goes to our servants over the years, Katie, Monica, Carolina, Danielle and Greg.

The East Hill Flying Club is an amazing place for flying! Thank you for providing such amazing, well-maintainted planes, and great instruction! Flying is what makes me really happy, and months, or even weeks, without a flight always feels like it is missing something. Being a board member over the last two and half years gave me a glimpse of what it takes to run such a club. I hope to find a similar place wherever I go.

Jim, my housemate over the last four years, is the best housemate one can ever dream of, and has become a great friend. As I come to think about it, I don't think we've had any argument in four years, which is pretty impressive. Thanks for always being there to listen when I needed, or discrete when I needed, and for talking about airplanes and going flying together. It's always been a great experience! Also, we're taking the Mooney to San Diego and Seattle next summer, right?

Bri, thanks for being here, for supporting me or talking *seriously* even when I don't want it, and for making me grow so much! You have been an amazing support over the last fifteen months!

I would like to end with my family, who is the closest to my heart. To my parents and to my sisters Bénédicte and Raphaëlle, to whom this thesis is dedicated. Papa et maman, thank you for raising me and making me love science and airplanes. Bénédicte, you'll become an amazing doctor, and Raphaëlle, a great genealogist. Thank you for all your support throughout this Ph. D.

Table of Contents

1 Introduction . . . **1**
- 1.1 Capsules . . . 4
- 1.2 Non-Well-Founded Computation . . . 4
- 1.3 Thesis Outline . . . 5

I Capsules . . . **8**

2 Computing with Capsules . . . **9**
- 2.1 Introduction . . . 9
- 2.2 Definitions . . . 13
  - 2.2.1 Capsules . . . 13
  - 2.2.2 Scope, Free and Bound Variables . . . 13
- 2.3 Scoping Issues . . . 14
  - 2.3.1 The λ-Calculus . . . 15
  - 2.3.2 Dynamic Scoping . . . 16
  - 2.3.3 Static Scoping with Closures . . . 18
  - 2.3.4 Static Scoping with Capsules . . . 19
- 2.4 Soundness . . . 20
  - 2.4.1 Evaluation Rules for Capsules . . . 20
  - 2.4.2 β-Reduction . . . 22
  - 2.4.3 Soundness . . . 22
  - 2.4.4 Closure Conversion . . . 26
- 2.5 A Functional/Imperative Language . . . 33
  - 2.5.1 Expressions . . . 33
  - 2.5.2 Types . . . 34
  - 2.5.3 Small-Step Evaluation . . . 35
  - 2.5.4 Garbage Collection . . . 38
  - 2.5.5 Big-Step Evaluation . . . 39
- 2.6 Conclusion . . . 41

3 Capsules and Closures 43
3.1 Introduction . . . 43
3.2 Closure semantics . . . 45
3.2.1 Definitions . . . 45
3.2.2 Big-step . . . 46
3.2.3 Small-step . . . 47
3.3 Equivalence of the semantics . . . 52
3.3.1 Definitions . . . 52
3.3.2 Big-step . . . 54
3.3.3 Small-step . . . 67
3.4 Capsules encode less information . . . 73
3.5 Discussion . . . 76
3.5.1 Capsules and Closures: a strong correspondence . . . 76
3.5.2 Suppression of the environment σ or the stack Σ . . . 76

4 Capsules and Separation 77
4.1 Introduction . . . 77
4.2 Assertions . . . 78
4.3 Partial Correctness . . . 79
4.4 Capsules and Separation Logic . . . 81
4.4.1 Definitions . . . 81
4.4.2 The Frame Rule . . . 83
4.4.3 Discussion . . . 84
4.4.4 Alternative Conditions . . . 84
4.5 Conclusion and Future Work . . . 88

II Non-Well-Founded Computation 90

5 Well-Founded Coalgebras, Revisited 91
5.1 Introduction . . . 92
5.2 Realization of Coinductive Types . . . 96
5.2.1 Directed Multigraphs . . . 96
5.2.2 Type Signatures . . . 97
5.2.3 Coalgebras and Realizations . . . 98
5.2.4 Final Coalgebras . . . 99
5.3 Characterization of Well-Founded Coalgebras . . . 100
5.3.1 Well-Founded Coalgebras . . . 101
5.3.2 Induction Principle . . . 102
5.3.3 Main Theorem . . . 103
5.3.4 Non-Well-Founded Coalgebras . . . 106
5.4 Well-Founded Examples . . . 106
5.4.1 Integer GCD . . . 107
5.4.2 Towers of Hanoi . . . 108

5.4.3 Mutually Recursive Functions: `even-odd` 109
5.4.4 Ackermann Function 111
5.5 Non-Well-Founded Examples 112
5.5.1 Descending Sequences 113
5.5.2 Alternating Turing Machines and IND Programs 114
5.6 Discussion 115

6 Language Constructs for Non-Well-Founded Computation 116
6.1 Introduction 117
6.2 Motivating Examples 122
6.2.1 Substitution 122
6.2.2 Probabilistic Protocols 123
6.2.3 Abstract Interpretation 127
6.2.4 Finite Automata 130
6.3 A Framework for Non-Well-Founded Computation 131
6.3.1 Generating Equations 133
6.4 A First Implementation 134
6.4.1 Equations and Solvers 134
6.4.2 Least Fixpoints 139
6.4.3 Generating Coinductive Elements and Substitution 140
6.4.4 Gaussian Elimination 142
6.5 Automatic Partitioning 144
6.6 Conclusion 145

7 CoCaml: Functional Programming with Regular Coinductive Types 148
7.1 Preliminaries 149
7.1.1 ML with Coalgebraic Datatypes 149
7.1.2 Capsule Semantics 151
7.2 Equations and Solvers 153
7.2.1 Equation Generation 154
7.2.2 The `iterator` Solver 155
7.2.3 The `constructor` Solver 156
7.2.4 The `gaussian` Solver 157
7.2.5 The `separate` Solver 159
7.2.6 User-defined Solvers 160
7.3 Examples 162
7.3.1 Finite and Infinite Lists 162
7.3.2 A Library for p-adic Numbers 166
7.3.3 Equality 173
7.4 Implementation 174
7.4.1 Overview 174
7.4.2 Partial Evaluation 175
7.4.3 Equality of Regular Coinductive Terms 176
7.5 Conclusions 179

8 Related Work **180**
8.1 Representation of the state of computation 180
8.2 Separation logic 181
8.3 Non-Well-Founded Computation 183

9 Summary and Future Directions **186**

A Derivations of examples **189**
A.1 Capsules 189
A.2 Closures 192

Bibliography **197**

Chapter 1

Introduction

Two paradigms have historically dominated the field of programming languages: the functional and the imperative. Examples of functional programming languages include LISP, Scheme, the ML family and Haskell, while Algol60, C and Java are examples of imperative programming languages. Functional programs consist of values and functions acting on those values, thus forming expressions that are evaluated to obtain a result. In a pure functional programming language, there is no notion of state or of variables changing value throughout the computation. In contrast, imperative programs consist of commands that change a state, along with control structures. Theoretically, functional programming is extensively studied as the λ-calculus, while imperative programming is usually modeled as transitions on states.

The functional and the imperative views might seem incompatible. However, several functional languages provide some way of modifying the state of computation: LISP and Scheme have mutable variables, ML has references, and Haskell provides monads. Moreover, while the most popular languages are imperative, some recent designs have also incorporated functional features like first-class functions;

examples include Python, C# and Javascript. Despite this convergence, theoretical models continue to be either functional or imperative, traditionally functional languages treat imperative features as an afterthought, and traditionally imperative languages treat functional features as an afterthought as well.

From a verification point of view, imperative programming constructs are closer to the hardware implementation, and are thus easier to compile. Several successful verification techniques for imperative programs are based on Hoare logic, including separation logic. Pure functional programming is closer to mathematics and preserves referential transparency. Dependent typing has recently been successful in verifying functional programs. But here again, the imperative and the functional views seem incompatible.

The functional and the imperative views differ widely in their treatment of variables. In imperative programs, variables can be explicitly reassigned during the execution, thus changing their value. This reassigning of variables is fundamental to most computations. In functional programs however, variables are not variable, they are merely identifiers that represent a value, and they cannot be reassigned. However, in many cases the functional programmer would like to use some imperative programming mechanisms involving mutable variables. To give this ability, references simulate mutable variables in ML, using the constructs ref, :=, and !, but these constructs alter the types. Other treatments, including monads [Mog91] and explicit heaps [MS77, Sco72, Sto81, HMT84] , seem to look at the state as an afterthought rather than something fundamental.

Another important aspect of studying programming languages that are both functional and imperative is getting a better understanding of how the functional and imperative paradigms interact. For example, the interaction between vari-

able assignment and first-class functions gives us recursive functions for free, using Landin's knot, also known as back-patching. More generally, circular data structures and coinductive types can be built using variable assignment and back-patching. In imperative programming languages, such circular data structures are usually manipulated using pointer manipulations. This is inelegant, prone to errors and difficult to maintain. It is in great contrast with the elegant way of using recursive functions and pattern-matching to manipulate inductive types in ML. We would like to have a similarly elegant way to manipulate circular data structures.

It is our thesis that a language can be both functional and imperative, without having one paradigm be an afterthought, and that a much more elegant treatment of functions on circular coinductive types in those languages is possible. This leads to the three goals of this dissertation:

1. a theoretical goal: we reconcile the imperative and the functional paradigms by introducing *capsules*, a simple, algebraic semantics for programming languages that is both functional and imperative;

2. a verification goal: we show how to use capsules to perform some verification tasks; we apply capsules to get a much simpler formulation of separation logic than usually found in the literature;

3. a practical goal: we provide some elegant programming language constructs that extend the concept of recursive functions to run on circular coinductive datastructures.

1.1 Capsules

Capsules reconcile the functional and the imperative approach by providing a simple semantics for a language that is both functional and imperative. They are based on three simple ideas: mutable bindings, α-conversion before β-reduction, and representation of the state as a λ-coterm. Mutable bindings allow identifiers to be bound exactly as in λ-expressions and let-expressions in functional languages, but are also mutable by explicit assignment; they existed in the early functional languages LISP and Scheme, but were dropped in the ML family. Performing α-conversion of any λ-term with a *fresh* variable before every β-reduction provides static scoping without the need for closures. Finally, representing the state as a λ-coterm — essentially a λ-term with possible loops in its abstract syntax tree — allows for a very clean and compact representation of the state of computation. An important advantage of this approach is that the interaction of higher-order functions and mutable variables gives recursive functions for free; they form loops in the state represented as a λ-coterm; this is also how recursive functions are often implemented. Finally, capsules provide a purely algebraic representation, without stacks, heaps, closures or combinators.

1.2 Non-Well-Founded Computation

An advantage of capsules is that they provide an elegant representation of circular (coinductive) elements. One then wonders whether those coinductive elements are useful and how one can do interesting computation with them. In OCaml for example, such coinductive elements can be created statically, but not dynamically, and not much can be done with them. Nevertheless they arise in numerous exam-

ples, including operations on infinitary λ-terms, probabilistic protocols and p-adic numbers; in abstract interpretation; and in constructions involving finite automata. In ML, standard recursion, along with pattern matching, provides a very intuitive way of defining functions on inductive types. For coinductive types, the same input might come back again and again and standard reduction just does not halt. Thus termination is guaranteed for *well-founded* inputs. But what happens for *non-well-founded* inputs, i.e., coinductive inputs with cycles? It turns out that, as long as the set of inputs that are encountered is finite, equations can often be written down, then solved, to provide a sensible answer. The programming language CoCaml provides a `corec` keyword allowing the programmer to write recursive functions on coinductive data in a way that is very similar to writing recursive functions on inductive data in ML.

1.3 Thesis Outline

Chapter 2 Computing with Capsules is a precise theoretical presentation of *capsules*. Capsules capture static scoping in an algebraic way instead of the combinatorial way of closures. The chapter provides soundness of capsules with respect to both the λ-calculus and closures, for a higher-order language without mutable bindings. It then presents a complete semantics of capsules for a higher-order language with mutable bindings. This chapter is based on the paper [JK12b].

Chapter 3 Capsules and Closures precisely compares capsules and closures for a higher-order language with mutable bindings. Capsules are the mathematical concept behind closures: in a sense, capsules are to closures what graphs are to their adjacency list representations. The chapter provides two comparisons: a comparison

based on big-step semantics, which is simpler and makes the relationship between capsules and closures easier to understand; and another comparison based on small-step semantics, more complicated but ensuring soundness of capsules even for infinite computations. This chapter is based on the papers [Jea11] and [Jea12].

Chapter 4 Capsules and Separation studies a formulation of separation logic using capsules. It proves the frame rule in this context and investigates alternative formulations with other side conditions. This chapter is based on the paper [JK12a].

Chapter 5 Well-Founded Coalgebras, Revisited proves some theoretical results characterizing well-founded coalgebras that slightly extend results of Adámek, Lücke and Milius [ALM07]. It also gives several examples for which this extension is useful, including mutually recursive functions, the Ackermann function, and the greatest common divisor of two integers. This chapter is based on the technical report [JKS13b].

Chapter 6 Language Constructs for Non-Well-Founded Computation introduces the problem of non-well-founded computation with numerous examples. It provides theoretical foundations, introduces programming language constructs to allow recursive computation on non-well-founded terms, and proposes a first implementation. This chapter is based on the paper [JKS13a].

Chapter 7 CoCaml: Programming with Coinductive Types describes CoCaml, a functional language extending OCaml, which allows the programmer to define functions on coinductive datatypes parameterized by an equation solver. It shows its implementation, based on representing the state as a capsule. It also provides more examples attesting to the usefulness of the new constructs. This chapter

is based on the technical report [JKS12].

Chapter 8 Related Work provides a detailed literature survey of the areas related to this dissertation — representing and reasoning about the state of computation, reasoning about locality, and computing with coinductive types.

Chapter 9 Future Directions concludes and opens up on ideas to extend the work of this dissertation.

Part I

Capsules

Chapter 2

Computing with Capsules

Capsules provide an algebraic representation of the state of a computation in higher-order functional and imperative languages. A capsule is essentially a finite coalgebraic representation of a regular closed λ-coterm. One can give an operational semantics based on capsules for a higher-order programming language with functional and imperative features, including mutable bindings. Static (lexical) scoping is captured purely algebraically without stacks, heaps, or closures. All operations of interest are typable with simple types, yet the language is Turing complete. Recursive functions are represented directly as capsules without the need for fixpoint combinators.

2.1 Introduction

Capsules provide an algebraic representation of the state of a computation in higher-order functional and imperative programming languages. They conservatively extend the classical λ-calculus with mutable variables and assignment, enabling the construction of certain regular coterms (infinite terms) representing recursive func-

tions without the need for fixpoint combinators. They have a well-defined statically-scoped evaluation semantics, are typable with simple types, and are Turing complete.

Representations of state have been studied in the past by many authors. Approaches include syntactic theories of control and state [FFF09, FH92a], the semantics of local storage [HMT84], functional languages with effects [MT, MT89a, MT91], monads [Mog91], closure structures [AH01, AHK06, AHK07] and denotational semantics [MS77, Sco72, Sto81]. Capsules provide a purely algebraic alternative in that no combinatorial structures are needed. Perhaps the most important aspect of capsules is that static scoping and local variables are captured without the need for closures. Cumbersome combinatorial machinery such as heaps, stores, stacks, and pointers are replaced with the single mathematical concept of variable binding. Nevertheless, capsules are equally expressive and represent the same data dependencies and liveness structure. In a sense, capsules are to closures what graphs are to their adjacency list representations.

Formally, a capsule is a particular syntactic representation of a finite coalgebra of the same signature as the λ-calculus. A capsule represents a regular closed λ-coterm (infinite λ-term) under the unique morphism to the final coalgebra of this signature. This final coalgebra has been studied under the name *infinitary λ-calculus*, focusing mostly on infinitary rewriting [BK09, KdV05]. It has been observed that the infinitary version does not share many of the desirable properties of its finitary cousin; for example, it is not confluent, and there exist coterms with no computational significance. However, all coterms represented by capsules are computationally meaningful.

One can give an operational semantics based on capsules for a higher-order programming language with both functional and imperative features, including recur-

sion and mutable variables, and this is one of the primary motivations of this work. All operations of interest are typable with simple types. Recursive functions are constructed directly using *Landin's knot* [Lan64] without the need for fixpoint combinators, which involve self-application and are untypable with simple types. Moreover, the traditional Y combinator forces a normal-order (lazy) evaluation strategy to ensure termination. Other more complicated fixpoint combinators can be used with applicative order by encapsulating the self-application in a thunk to delay evaluation, but this is even more unnatural. In contrast, the construction of recursive functions with Landin's knot is direct and simply typable, and corresponds more closely to implementations. Turing completeness is impossible with finite types and finite terms, as the simply-typed λ-calculus is strongly normalizing; so we must have either infinitary types or infinitary terms. Whereas the former is more conventional, we believe the latter is more natural and closer to implementations.

Dynamic scoping, which was the scoping discipline in early versions of LISP and Python, and which still exists in many languages today, can be regarded as an implementation of lazy β-reduction that fails to observe the principle of safe substitution (α-conversion to avoid capture of free variables). We explain this view more fully with a detailed example in §6.2. In contrast, the λ-calculus with β-reduction and safe substitution is statically scoped. Both capsules and closures provide static scoping, but capsules do so without any extra combinatorial machinery. Moreover, capsules work correctly in the presence of mutable variables, whereas closures, naively implemented, do not (a counterexample is given in §2.4.4). To correctly handle mutable variables, closures require some form of indirection, and care must be taken to perform updates nondestructively.

Capsules provide a common framework for representing the global state of com-

putation for both functional and imperative programs. Valuations of mutable variables used in the semantics of imperative programs and closure structures used in the operational semantics of functional programs can be simulated. Capsules also allow a clean mathematical definition of garbage collection: there is a natural notion of morphism, and the garbage-collected version of a capsule is the unique (up to isomorphism) initial object among its monomorphic preimages.

This chapter is organized as follows. In §2.2, we give formal definitions of capsules. In §2.3, we give a detailed motivating example comparing how closures and capsules deal with scoping issues. In §2.4 we prove two theorems. The first (Theorem 2.4.1) establishes that capsule evaluation faithfully models β-reduction in the λ-calculus with safe substitution. The second (Theorem 2.4.7) defines closure conversion for capsules and proves soundness of the translation, provided there is no variable assignment. Taken together, these two theorems establish that closures also correctly model β-reduction in the λ-calculus with safe substitution. The same results hold in the presence of assignment, but the definition of closures must be extended; the definition of capsules remains the same, as we will see in chapter 3. The proof techniques in this section are purely algebraic and involve some interesting applications of coinduction. Finally, in §2.5, we describe a simply-typed functional/imperative language with mutable bindings and give an operational semantics in terms of capsules.

2.2 Definitions

2.2.1 Capsules

Consider the simply-typed λ-calculus with typed constants (e.g., $3 : \mathsf{int}$, $\mathsf{true} : \mathsf{bool}$, $+ : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $\leq : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$). The set of λ-abstractions is denoted λ-Abs and the set of constants is denoted Const. A λ-term is *irreducible* if it is either a λ-abstraction $\lambda x.e$ or a constant c. The set of irreducible terms is $\mathsf{Irred} = \lambda\text{-}\mathsf{Abs} + \mathsf{Const}$. Note that variables x are not irreducible.

Let $\mathsf{FV}(e)$ denote the set of free variables of e. A *capsule* is a pair $\langle e, \sigma \rangle$, where e is a λ-term and $\sigma : \mathsf{Var} \rightharpoonup \mathsf{Irred}$ is a partial function with finite domain $\mathsf{dom}\,\sigma$, such that

(i) $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma$

(ii) if $x \in \mathsf{dom}\,\sigma$, then $\mathsf{FV}(\sigma(x)) \subseteq \mathsf{dom}\,\sigma$.

A capsule $\langle e, \sigma \rangle$ is *irreducible* if e is.

Note that cycles are allowed; this is how recursive functions are represented. For example, we might have $\sigma(f) = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot f(n-1)$.

2.2.2 Scope, Free and Bound Variables

Let $\langle e, \sigma \rangle$ be a capsule and let d be either e or $\sigma(y)$ for some $y \in \mathsf{dom}\,\sigma$. The *scope* of an occurrence of a binding operator λx in d is its scope in the λ-term d as normally defined.

Consider an occurrence of a variable x in d. The closure conditions (i) and (ii) of §2.2.1 ensure that one of the following two conditions holds:

- that occurrence of x falls in the scope of a binding operator λx in d, in which case it is bound to the innermost binding operator λx in d in whose scope it lies; or
- it is free in d, but $x \in \mathsf{dom}\,\sigma$, in which case it is bound by σ to the value $\sigma(x)$.

Thus every variable x in a capsule is essentially bound. These conditions thus preclude catastrophic failure due to access of unbound variables.

It is important to note that scope does not extend through bindings in σ. For example, consider the capsule $\langle \lambda x.y,\ [y = \lambda z.x,\ x = 2] \rangle$. The free occurrence of x in $\lambda z.x$ is not bound to the λx in $\lambda x.y$, but rather to the value 2. The coalgebra represented by the capsule has three states and represents the closed term $\lambda x.\lambda z.2$. For this reason, one cannot simply substitute $\sigma(y)$ for y in e without α-conversion. This is also reflected in the evaluation rules to be given in §2.4.1. In a capsule $\langle e,\, \sigma \rangle$, all free variables in e or $\sigma(y)$ are in $\mathsf{dom}\,\sigma$, therefore bound to a value; thus every capsule represents a closed coterm.

The term *α-conversion* refers to the renaming of bound variables. With a capsule $\langle e,\, \sigma \rangle$, this can happen in two ways. The traditional form maps a subterm $\lambda x.d$ to $\lambda y.d[x/y]$, provided y would not be captured in d. One can also rename a variable $x \in \mathsf{dom}\,\sigma$ and all free occurrences of x in e and $\sigma(z)$ for $z \in \mathsf{dom}\,\sigma$ to y, provided $y \notin \mathsf{dom}\,\sigma$ already and y would not be captured.

2.3 Scoping Issues

We motivate the results of §2.4 with an example illustrating how dynamic scoping arises from a naive implementation of lazy substitution and how capsules and closures remedy the situation.

2.3.1 The λ-Calculus

The oldest and simplest of all functional languages is the λ-calculus. In this system, a *state* is a closed λ-term, and *computation* consists of a sequence of β-reductions

$$(\lambda x.d)\ e\ \rightarrow\ d[x/e],$$

where $d[x/e]$ denotes the safe substitution of e for all free occurrences of x in d. *Safe substitution* means that bound variables in d may have to be renamed (α-converted) to avoid capturing free variables of the substituted term e.

For example, consider the closed λ-term $(\lambda y.(\lambda z.\lambda y.z\,4)\,\lambda x.y)\,3\,2$. Evaluating this term in (shallow) applicative order[1], we get the following sequence of terms leading to the value 3:

$$\begin{aligned}(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 &\ \rightarrow\ (\lambda z.\lambda y.z\ 4)\ (\lambda x.3)\ 2\\ &\ \rightarrow\ (\lambda y.(\lambda x.3)\ 4)\ 2 \rightarrow (\lambda x.3)\ 4 \rightarrow 3 \qquad (2.1)\end{aligned}$$

No α-conversion was necessary. In fact, no α-conversion is *ever* necessary with applicative-order evaluation of closed terms, because the argument substituted for a parameter in a β-reduction is closed, thus has no free variables to be captured. It is key that the term being evaluated be closed, as studied in the combinatorial weak λ-calculus [cH98] and closed reductions [FMS05].

However, the λ-calculus is confluent, and we may choose a different order of evaluation; but an alternative order may require α-conversion. For example, the

[1] Also known as *left-to-right call-by-value order*, the order of evaluation in which the leftmost innermost redex is reduced first, except that redexes in the scope of binding operators λx are ineligible for reduction.

following reduction sequence is also valid:

$$\begin{aligned}(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 &\rightarrow (\lambda y.\lambda w.(\lambda x.y)\ 4)\ 3\ 2 \\ &\rightarrow (\lambda w.(\lambda x.3)\ 4)\ 2 \rightarrow (\lambda x.3)\ 4 \rightarrow 3 \end{aligned} \tag{2.2}$$

A change of bound variable was required in the first step to avoid capturing the free occurrence of y in $\lambda x.y$ substituted for z. Failure to do so results in the erroneous value 2:

$$\begin{aligned}(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 &\rightarrow (\lambda y.\lambda y.(\lambda x.y)\ 4)\ 3\ 2 \\ &\rightarrow (\lambda y.(\lambda x.y)\ 4)\ 2 \rightarrow (\lambda x.2)\ 4 \rightarrow 2 \end{aligned} \tag{2.3}$$

2.3.2 Dynamic Scoping

In the early development of functional programming, specifically with the language LISP, it was quickly determined that physical substitution is too inefficient because it requires copying [McC81]. This led to the introduction of *environments*, used to effect lazy substitution. Instead of doing the actual substitution when performing a β-reduction, one can defer the substitution by saving it in an environment, then look up the value when needed.

An *environment* is a partial function $\sigma : \mathsf{Var} \rightharpoonup \mathsf{Irred}$ with finite domain. A *state* is a pair $\langle e, \sigma \rangle$, where e is the term to be evaluated and σ is an environment with bindings for the free variables in e. Environments need to be updated, which

requires a *rebinding operator*[2]

$$(\sigma[x/e])(y) = \begin{cases} e, & \text{if } x = y, \\ \sigma(y), & \text{if } x \neq y. \end{cases}$$

Naively implemented, the rules are

$$\begin{aligned} \langle (\lambda x.d)\, e,\, \sigma \rangle &\rightarrow \langle d,\, \sigma[x/e] \rangle \\ \langle y,\, \sigma \rangle &\rightarrow \langle \sigma(y),\, \sigma \rangle \end{aligned}$$

where the first rule saves the deferred substitution in the environment and the second looks up the value. This is quite easy to implement. Moreover, it stands to reason that if β-reduction in applicative order does not require any α-conversions, then the lazy approach should not either. After all, the same terms are being substituted, just at a later time.

However, this is not the case. In the example above, we obtain the following sequence of states leading to the value 2:

$$\begin{aligned} \langle (\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2,\ \texttt{[ ]} \rangle &\rightarrow \langle (\lambda z.\lambda y.z\ 4)\ (\lambda x.y)\ 2,\ \texttt{[}y = 3\texttt{]} \rangle \\ &\rightarrow \langle (\lambda y.z\ 4)\ 2,\ \texttt{[}y = 3,\ z = \lambda x.y\texttt{]} \rangle \\ &\rightarrow \langle z\ 4,\ \texttt{[}y = 2,\ z = \lambda x.y\texttt{]} \rangle \\ &\rightarrow \langle (\lambda x.y)\ 4,\ \texttt{[}y = 2,\ z = \lambda x.y\texttt{]} \rangle \\ &\rightarrow \langle y,\ \texttt{[}y = 2,\ z = \lambda x.y,\ x = 4\texttt{]} \rangle \\ &\rightarrow \langle 2,\ \texttt{[}y = 2,\ z = \lambda x.y,\ x = 4\texttt{]} \rangle \end{aligned}$$

The issue is that the lazy approach fails to observe safe substitution. This example effectively performs the deferred substitutions in the order (2.3) without the change

[2]Note that we use the same notation for replacing all free occurrences of a variable in a expression, and rebinding a variable in an environment. This is on purpose, as these two operations are of similar nature.

of bound variable. Nevertheless, this was the strategy adopted by early versions of LISP [McC81]. It was not considered a bug but a feature and was called *dynamic scoping*.

2.3.3 Static Scoping with Closures

The semantics of evaluation was brought more in line with the λ-calculus with the introduction of *closures* [Lan64, McC81]. Formally, a *closure* is defined as a pair $\{\lambda x.e,\, \sigma\}$, where the $\lambda x.e$ is a λ-abstraction and σ is a partial function from variables to values that is used to interpret the free variables of $\lambda x.e$. When a λ-abstraction is evaluated, it is paired with the environment σ at the point of the evaluation, and the value is the closure $\{\lambda x.e,\, \sigma\}$. Thus we have

$$\sigma : \mathsf{Var} \rightharpoonup \mathsf{Val} \qquad\qquad \mathsf{Val} = \mathsf{Const} + \mathsf{Cl}$$

where Cl denotes the set of closures. We require that for a closure $\{\lambda x.e,\, \sigma\}$, $\mathsf{FV}(\lambda x.e) \subseteq \mathsf{dom}\,\sigma$. Note that the definitions of values and closures are mutually dependent.

The new reduction rules are

$$\begin{aligned} \langle \lambda x.d,\, \sigma\rangle &\rightarrow_{\mathsf{cl}} \{\lambda x.d,\, \sigma\} \\ \langle \{\lambda x.d,\, \sigma\}\ e,\, \tau\rangle &\rightarrow_{\mathsf{cl}} \langle d,\, \sigma[x/e]\rangle \\ \langle y,\, \sigma\rangle &\rightarrow_{\mathsf{cl}} \sigma(y). \end{aligned}$$

The second rule says that an application uses the context σ that was in effect when the closure was created, not the context τ of the call. Turning to our running

example,

$$\begin{aligned}
\langle(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2,\ [\,]\rangle &\to_{\mathsf{cl}} \langle(\lambda z.\lambda y.z\ 4)\ (\lambda x.y)\ 2,\ [y=3]\rangle \\
&\to_{\mathsf{cl}} \langle(\lambda y.z\ 4)\ 2,\ [y=3,\ z=\{\lambda x.y,\ [y=3]\}]\rangle \\
&\to_{\mathsf{cl}} \langle z\ 4,\ [y=2,\ z=\{\lambda x.y,\ [y=3]\}]\rangle \\
&\to_{\mathsf{cl}} \langle\{\lambda x.y,\ [y=3]\}\ 4,\ [y=2,\ z=\{\lambda x.y,\ [y=3]\}]\rangle \\
&\to_{\mathsf{cl}} \langle(\lambda x.y)\ 4,\ [y=3]\rangle \\
&\to_{\mathsf{cl}} \langle y,\ [y=3,\ x=4]\rangle \\
&\to_{\mathsf{cl}} \langle 3,\ [y=3,\ x=4]\rangle
\end{aligned}$$

2.3.4 Static Scoping with Capsules

Closures correctly capture the semantics of β-reduction with safe substitution, but at the expense of introducing extra combinatorial machinery to represent and manipulate pairs $\{\lambda x.e,\, \sigma\}$. Capsules allow us to revert to a purely λ-theoretic framework without losing the benefits of closures.

Capsules were defined formally in §2.2.1. Capsule evaluation semantics looks very much like the original evaluation semantics of LISP, with the added twist that a fresh variable is substituted for the parameter in β-reductions. The small-step reduction rules for capsules are

$$\langle(\lambda x.e)\ v,\, \sigma\rangle \to \langle e[x/y],\, \sigma[y/v]\rangle \quad (y \text{ fresh})$$

$$\langle y,\, \sigma\rangle \to \langle\sigma(y),\, \sigma\rangle$$

In the original evaluation semantics of LISP, the right-hand side of the first rule is $\langle e,\, \sigma[x/v]\rangle$, which gives dynamic scoping. The key difference here is the introduction of the fresh variable y in the application rule. This is tantamount to performing an

α-conversion on the parameter of a function just before applying it. Turning to our running example, we see that this approach gives the correct result.

$$\begin{aligned}
\langle(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2,\ \texttt{[ ]}\rangle &\rightarrow_{\mathsf{ca}} \langle(\lambda z.\lambda y.z\ 4)\ (\lambda x.y')\ 2,\ \texttt{[}y'=3\texttt{]}\rangle\\
&\rightarrow_{\mathsf{ca}} \langle(\lambda y.z'\ 4)\ 2,\ \texttt{[}y'=3,\ z'=\lambda x.y'\texttt{]}\rangle\\
&\rightarrow_{\mathsf{ca}} \langle z'\ 4,\ \texttt{[}y'=3,\ z'=\lambda x.y',\ y''=2\texttt{]}\rangle\\
&\rightarrow_{\mathsf{ca}} \langle(\lambda x.y')\ 4,\ \texttt{[}y'=3,\ z'=\lambda x.y',\ y''=2\texttt{]}\rangle\\
&\rightarrow_{\mathsf{ca}} \langle y',\ \texttt{[}y'=3,\ z'=\lambda x.y',\ y''=2,\ x'=4\texttt{]}\rangle\\
&\rightarrow_{\mathsf{ca}} \langle 3,\ \texttt{[}y'=3,\ z'=\lambda x.y',\ y''=2,\ x'=4\texttt{]}\rangle
\end{aligned}$$

We prove soundness formally in §2.4.

2.4 Soundness

In this section we show that capsule evaluation is statically scoped under applicative-order evaluation and correctly models β-reduction in the λ-calculus with safe substitution.

2.4.1 Evaluation Rules for Capsules

Let $d, e, \ldots$ denote λ-terms and $u, v, \ldots$ irreducible λ-terms (λ-abstractions and constants). Variables are denoted $x, y, \ldots$ and constants c, f. For any constant f denoting a function in the language, there exists an application function from terms to terms, that is also written f.

The small-step evaluation rules for capsules consist of reduction rules

$$\langle(\lambda x.e)\ v,\, \sigma\rangle \to_{\mathsf{ca}} \langle e[x/y],\, \sigma[y/v]\rangle \quad (y \text{ fresh}) \tag{2.4}$$

$$\langle f\ c,\, \sigma\rangle \to_{\mathsf{ca}} \langle f(c),\, \sigma\rangle \tag{2.5}$$

$$\langle y,\, \sigma\rangle \to_{\mathsf{ca}} \langle \sigma(y),\, \sigma\rangle \tag{2.6}$$

and context rules

$$\frac{\langle d,\, \sigma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle d',\, \tau\rangle}{\langle d\ e,\, \sigma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle d'\ e,\, \tau\rangle} \qquad \frac{\langle e,\, \sigma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e',\, \tau\rangle}{\langle v\ e,\, \sigma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle v\ e',\, \tau\rangle} \tag{2.7}$$

where $\xrightarrow{*}_{\mathsf{ca}}$ denotes the repetition of zero or more steps of $\to_{\mathsf{ca}}$. The reduction rules (2.4)–(2.6) identify three forms of redex: an application $(\lambda x.e)\ v$, an application $f\ c$ where f and c are constants, or a variable $y \in \mathsf{dom}\,\sigma$. The context rules (2.7) uniquely identify a redex in a well-typed non-irreducible capsule according to an applicative-order reduction strategy.

The corresponding large-step rules are

$$\langle y,\, \sigma\rangle \Downarrow_{\mathsf{ca}} \langle \sigma(y),\, \sigma\rangle \tag{2.8}$$

$$\frac{\langle d,\, \sigma\rangle \Downarrow_{\mathsf{ca}} \langle f,\, \tau\rangle \qquad \langle e,\, \tau\rangle \Downarrow_{\mathsf{ca}} \langle c,\, \rho\rangle}{\langle d\ e,\, \sigma\rangle \Downarrow_{\mathsf{ca}} \langle f(c),\, \rho\rangle} \tag{2.9}$$

$$\frac{\langle d,\, \sigma\rangle \Downarrow_{\mathsf{ca}} \langle \lambda x.a,\, \tau\rangle \quad \langle e,\, \tau\rangle \Downarrow_{\mathsf{ca}} \langle v,\, \rho\rangle \quad \langle a[x/y],\, \rho[y/v]\rangle \Downarrow_{\mathsf{ca}} \langle u,\, \pi\rangle}{\langle d\ e,\, \sigma\rangle \Downarrow_{\mathsf{ca}} \langle u,\, \pi\rangle}\ (y \text{ fresh}) \tag{2.10}$$

These rules are best understood in terms of the interpreter they generate:

$$\begin{aligned}
\mathsf{Eval}(c, \sigma) &= \langle c,\, \sigma\rangle \\
\mathsf{Eval}(\lambda x.e, \sigma) &= \langle \lambda x.e,\, \sigma\rangle \\
\mathsf{Eval}(y, \sigma) &= \langle \sigma(y),\, \sigma\rangle \\
\mathsf{Eval}(d\ e, \sigma) &= \mathsf{let}\ \langle u,\, \tau\rangle = \mathsf{Eval}(d, \sigma)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \langle v,\, \rho\rangle = \mathsf{Eval}(e, \tau)\ \mathsf{in}
\end{aligned} \tag{2.11}$$

$$\mathsf{Apply}(u, v, \rho)$$

$$\begin{aligned}\mathsf{Apply}(f, c, \sigma) &= \langle f(c), \sigma\rangle \\ \mathsf{Apply}(\lambda x.e, v, \sigma) &= \mathsf{Eval}(e[x/y], \sigma[y/v]) \quad (y \text{ fresh})\end{aligned} \tag{2.12}$$

2.4.2 β-Reduction

The small-step evaluation rules for β-reduction in applicative order are the same as for capsules, except we replace (2.4) with

$$\langle(\lambda x.e)\ v,\ \sigma\rangle \to \langle e[x/v],\ \sigma\rangle \tag{2.13}$$

(substitution instead of rebinding). The other rules (2.5)–(2.7) are the same. This makes sense even in the presence of cycles (recursive functions).

Note that the initial valuation σ persists unchanged throughout the computation. We might suppress it to simplify notation, giving

$$(\lambda x.e)\ v \to e[x/v] \qquad f\ c \to f(c) \qquad y \to \sigma(y)$$

$$\frac{d \xrightarrow{*} d'}{(d\ e) \xrightarrow{*} (d'\ e)} \qquad \frac{e \xrightarrow{*} e'}{(v\ e) \xrightarrow{*} (v\ e')}$$

However, it is still implicitly present, as it is needed to evaluate variables y.

The corresponding interpreter Eval_β is defined exactly like Eval except for rule (2.12), which we replace with

$$\mathsf{Apply}_\beta(\lambda x.e, v, \sigma) = \mathsf{Eval}_\beta(e[x/v], \sigma).$$

2.4.3 Soundness

We are now ready to prove soundness. We do so by proving that the evaluation using capsules of an expression e is mirrored by an evaluation of e using β-reduction.

Let S denote a sequential composition of rebinding operators $[y_1/v_1]\cdots[y_k/v_k]$, applied from left to right. Applied to a partial valuation $\sigma : \mathsf{Var} \rightharpoonup \mathsf{Irred}$, the operator S sequentially rebinds y_1 to v_1, then y_2 to v_2, and so on. The result is denoted σS. Formally, $\sigma(S[y/v]) = (\sigma S)[y/v]$.

To every rebinding operator $S = [y_1/v_1]\cdots[y_k/v_k]$ there corresponds a safe substitution operator $S^- = [y_k/v_k]\cdots[y_1/v_1]$, also applied from left to right. Applied to a λ-term e, S^- safely substitutes v_k for all free occurrences of y_k in e, then v_{k-1} for all free occurrences of y_{k-1} in $e[y_k/v_k]$, and so on. The result is denoted eS^-. Formally, $e(S^-[y/v]) = (eS^-)[y/v]$. Note that $(ST)^- = T^-S^-$.

If $S = [y_1/v_1]\cdots[y_k/v_k]$, we assume that y_i does not occur in v_j for $i \geq j$; however, y_i may occur in v_j if $i < j$. This means that if $\mathsf{FV}(e) \subseteq \{y_1, \ldots, y_k\}$ and $\mathsf{FV}(v_j) \subseteq \{y_1, \ldots, y_{j-1}\}$, $1 \leq j \leq k$, then eS^- is closed.

The following theorem establishes soundness of capsule evaluation with respect to β-reduction in the λ-calculus.

Theorem 2.4.1 *$\mathsf{Eval}_\beta(e, \sigma) = \langle v,\, \sigma\rangle$ if and only if there exist irreducible terms $v_1, \ldots, v_k, u$ and a rebinding operator $S = [y_1/v_1]\cdots[y_k/v_k]$, where $y_1, \ldots, y_k$ do not occur in e, v, or σ, such that $\mathsf{Eval}(e, \sigma) = \langle u,\, \sigma S\rangle$ and $v = uS^-$.*

Proof. We show the implication in both directions by induction on the number of steps in the evaluation. The result is trivially true for inputs of the form $\langle c,\, \sigma\rangle$, $\langle \lambda x.e,\, \sigma\rangle$, and $\langle \sigma(y),\, \sigma\rangle$, and this gives the basis of the induction.

($\Rightarrow$) For an input of the form $\langle d\ e,\, \sigma\rangle$, we show the implication in both directions. We first show that if $\mathsf{Eval}(d\ e, \sigma)$ is defined, then so is $\mathsf{Eval}_\beta(d\ e, \sigma)$, and the relationship between the two values is as described in the statement of the theorem. By definition of Eval, we have

$$\mathsf{Eval}(d, \sigma) = (u, \sigma S) \qquad\qquad \mathsf{Eval}(e, \sigma S) = (v, \sigma ST)$$

for some $S = [y_1/v_1]\cdots[y_m/v_m]$ and $T = [y_{m+1}/v_{m+1}]\cdots[y_n/v_n]$, where $y_1,\ldots,y_n$ are the fresh variables and $v_1,\ldots,v_n$ the irreducible terms bound to them in applications of the rule (2.12) during the evaluation of d and e. By the induction hypothesis, we have

$$\mathsf{Eval}_\beta(d,\sigma) = \langle uS^-,\, \sigma\rangle \qquad \mathsf{Eval}_\beta(e,\sigma S) = \langle vT^-,\, \sigma S\rangle.$$

Since the variables $y_1,\ldots,y_m$ do not occur in e, they are not accessed in its evaluation, thus $\mathsf{Eval}_\beta(e,\sigma) = \langle vT^-,\, \sigma\rangle$. Also, since $y_{m+1},\ldots,y_n$ do not occur in u and $y_1,\ldots,y_m$ do not occur in v, we have $uS^- = u(ST)^-$ and $vT^- = v(ST)^-$, thus

$$\mathsf{Eval}_\beta(d,\sigma) = \langle u(ST)^-,\, \sigma\rangle \qquad \mathsf{Eval}_\beta(e,\sigma) = \langle v(ST)^-,\, \sigma\rangle.$$

We thus have

$$\mathsf{Eval}(d\ e,\sigma) = \mathsf{Apply}(u,v,\sigma ST) \qquad \mathsf{Eval}_\beta(d\ e,\sigma) = \mathsf{Apply}_\beta(u(ST)^-,v(ST)^-,\sigma)$$

If u and v are constants, say $u = f$ and $v = c$, then

$$\begin{aligned}\mathsf{Eval}(d\ e,\sigma) &= \mathsf{Apply}(f,c,\sigma ST) = \langle f(c),\, \sigma ST\rangle\\ \mathsf{Eval}_\beta(d\ e,\sigma) &= \mathsf{Apply}_\beta(f,c,\sigma) = \langle f(c),\, \sigma\rangle,\end{aligned}$$

and the implication holds. If u is a λ-abstraction, say $u = \lambda x.a$, then $u(ST)^- = \lambda x.a(ST)^-$. Then

$$\begin{aligned} a(ST)^-[x/v(ST)^-] &= a[x/v](ST)^- = a[x/y_{n+1}][y_{n+1}/v](ST)^-\\ &= a[x/y_{n+1}](ST[y_{n+1}/v])^-,\end{aligned}$$

therefore

$$\begin{aligned}\mathsf{Eval}(d\ e,\sigma) &= \mathsf{Apply}(\lambda x.a,v,\sigma ST) = \mathsf{Eval}(a[x/y_{n+1}],\sigma ST[y_{n+1}/v])\\ \mathsf{Eval}_\beta(d\ e,\sigma) &= \mathsf{Apply}_\beta(\lambda x.a(ST)^-,v(ST)^-,\sigma) = \mathsf{Eval}_\beta(a(ST)^-[x/v(ST)^-],\sigma)\\ &= \mathsf{Eval}_\beta(a[x/y_{n+1}](ST[y_{n+1}/v])^-,\sigma),\end{aligned}$$

and the implication holds in this case as well.

($\Leftarrow$) For the reverse implication, assume that $\mathsf{Eval}_\beta(d\ e, \sigma)$ is defined. Let $\langle u,\, \sigma\rangle = \mathsf{Eval}_\beta(d, \sigma)$ and $\langle v,\, \sigma\rangle = \mathsf{Eval}_\beta(e, \sigma)$. By the induction hypothesis, there exist variables $y_1, \ldots, y_m$ and irreducible terms $v_1, \ldots, v_m$ and r such that

$$u = rS^- \qquad\qquad \mathsf{Eval}(d, \sigma) = \langle r,\, \sigma S\rangle,$$

where $S = [y_1/v_1]\cdots[y_m/v_m]$. We also have $\langle v,\, \sigma S\rangle = \mathsf{Eval}_\beta(e, \sigma S)$, since the evaluation of e does not depend on the variables $y_1, \ldots, y_m$. Again by the induction hypothesis, there exist variables $y_{m+1}, \ldots, y_n$ and irreducible terms $v_{m+1}, \ldots, v_n$ and s such that

$$v = sT^- = sT^-S^- = s(ST)^- \qquad\qquad \mathsf{Eval}(e, \sigma S) = \langle s,\, \sigma ST\rangle,$$

where $T = [y_{m+1}/v_{m+1}]\cdots[y_n/v_n]$. Then $ST = [y_1/v_1]\cdots[y_n/v_n]$ and

$$\mathsf{Eval}_\beta(d\ e, \sigma) = \mathsf{Apply}_\beta(u, v, \sigma) \qquad\qquad \mathsf{Eval}(d\ e, \sigma) = \mathsf{Apply}(r, s, \sigma ST).$$

If u and v are constants, say $u = f$ and $v = c$, then $r = f$ and $s = c$. In this case we have

$$\mathsf{Eval}_\beta(d\ e, \sigma) = \mathsf{Apply}_\beta(f, c, \sigma) = \langle f(c),\, \sigma\rangle$$
$$\mathsf{Eval}(d\ e, \sigma) = \mathsf{Apply}(f, c, \sigma ST) = \langle f(c),\, \sigma ST\rangle,$$

and the implication holds. If u is a λ-abstraction, then $r = \lambda x.a$ and $u = \lambda x.aS^- = \lambda x.a(ST)^-$. In this case

$$\begin{aligned} a(ST)^-[x/s(ST)^-] &= a[x/s](ST)^- = a[x/y_{n+1}][y_{n+1}/s](ST)^- \\ &= a[x/y_{n+1}](ST[y_{n+1}/s])^-, \end{aligned}$$

thus

$$\begin{aligned} \mathsf{Eval}_\beta(d\ e, \sigma) &= \mathsf{Apply}_\beta(\lambda x.a(ST)^-, v, \sigma) = \mathsf{Eval}_\beta(a(ST)^-[x/s(ST)^-], \sigma) \\ &= \mathsf{Eval}_\beta(a[x/y_{n+1}](ST[y_{n+1}/s])^-, \sigma), \\ \mathsf{Eval}(d\ e, \sigma) &= \mathsf{Apply}(\lambda x.a, s, \sigma ST) = \mathsf{Eval}(a[x/y_{n+1}], \sigma ST[y_{n+1}/s]), \end{aligned}$$

so the implication holds in this case as well. □

2.4.4 Closure Conversion

Closure conversion is an important step of compiling any functional language. In this section we demonstrate how to closure-convert a capsule and show that the transformation is sound with respect to the evaluation semantics of closures and capsules in applicative-order evaluation, provided variables are not mutable.

Closures do not work in the presence of mutable variables without introducing the further complication of references and indirection. This is because closures fix the environment once and for all when the closure is formed, whereas mutable variables allow the environment to be subsequently changed. An example is given by $(\lambda y.(\lambda x.y)\,(y := 4; y))\,3$, for which capsules give 4 and closures, implemented naively as above, give 3. Capsules handle the assignment correctly, but with closures, the assignment has no effect.

Care must also be taken to implement updates nondestructively so as not to overwrite parameters and local variables of recursive procedures, an issue that is usually addressed at the implementation level. Again, the issue does not arise with capsules.

Even without indirection, the types of closures and closure environments are more involved than those of capsules. A closer look at the definitions of §2.3.3 shows that the definitions are mutually dependent and require a recursive, coinductive type definition [Rut00, §11]. The types are

$$\begin{aligned} &\mathsf{Env} = \mathsf{Var} \rightharpoonup \mathsf{Val} && \text{closure environments}\\ &\mathsf{Val} = \mathsf{Const} + \mathsf{Cl} && \text{values}\\ &\mathsf{Cl} = \lambda\text{-}\mathsf{Abs} \times \mathsf{Env} && \text{closures} \end{aligned}$$

We use boldface for closure environments $\boldsymbol{\sigma} : \mathsf{Env}$ to distinguish them from the simpler capsule environments. Closures $\{\lambda x.e,\ \boldsymbol{\sigma}\}$ must satisfy the additional requirement that $\mathsf{FV}(\lambda x.e) \subseteq \mathsf{dom}\,\boldsymbol{\sigma}$.

A *state* is now a pair $\langle e,\ \boldsymbol{\sigma}\rangle$, where $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\boldsymbol{\sigma}$, but the result of an evaluation is a Val. The evaluation semantics for closures, expressed as an interpreter $\mathsf{Eval_c}$, is

$$\begin{aligned}
\mathsf{Eval_c}(c, \boldsymbol{\sigma}) &= c \\
\mathsf{Eval_c}(\lambda x.e, \boldsymbol{\sigma}) &= \{\lambda x.e,\ \boldsymbol{\sigma}\} \\
\mathsf{Eval_c}(y, \boldsymbol{\sigma}) &= \boldsymbol{\sigma}(y) \\
\mathsf{Eval_c}(d\ e, \boldsymbol{\sigma}) &= \mathsf{let}\ u = \mathsf{Eval_c}(d, \boldsymbol{\sigma})\ \mathsf{in} \\
&\ \ \mathsf{let}\ v = \mathsf{Eval_c}(e, \boldsymbol{\sigma})\ \mathsf{in} \\
&\ \ \mathsf{Apply_c}(u, v)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Apply_c}(f, c) &= f(c) \\
\mathsf{Apply_c}(\{\lambda x.a,\ \boldsymbol{\rho}\}, v) &= \mathsf{Eval_c}(a, \boldsymbol{\rho}[x/v])
\end{aligned} \tag{2.14}$$

The types are

$$\mathsf{Eval_c} : \mathsf{Exp} \times \mathsf{Env} \rightharpoonup \mathsf{Val} \qquad\qquad \mathsf{Apply_c} : \mathsf{Val} \times \mathsf{Val} \rightharpoonup \mathsf{Val}.$$

The correspondence with capsules becomes simpler to state if we modify the interpreter to α-convert the term $\lambda x.a$ to $\lambda y.a[x/y]$ just before applying it, where y is the fresh variable that would be chosen by the capsule interpreter. Accordingly, we replace (2.14) with

$$\mathsf{Apply_c}(\{\lambda x.a,\ \boldsymbol{\rho}\}, v) = \mathsf{Eval_c}(a[x/y], \boldsymbol{\rho}[y/v]) \quad (y \text{ fresh})$$

The corresponding large-step rules are

$$\langle c,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{} c \qquad\qquad \langle \lambda x.e,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{} \{\lambda x.e,\ \boldsymbol{\sigma}\} \qquad\qquad \langle y,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{} \boldsymbol{\sigma}(y) \tag{2.15}$$

$$\frac{\langle d,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f \qquad \langle e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} c}{\langle d\ e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f(c)} \tag{2.16}$$

$$\frac{\langle d,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} \{\lambda x.a,\, \boldsymbol{\rho}\} \qquad \langle e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} v \qquad \langle a[x/y],\, \boldsymbol{\rho}[y/v]\rangle \xrightarrow[c]{*} u}{\langle d\ e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} u}\ (y \text{ fresh}) \tag{2.17}$$

The closure-converted form of a capsule $\langle e,\, \sigma\rangle$ is $\langle e,\, \overline{\sigma}\rangle$, where for any σ, we define $\overline{\sigma}$ as a map with $\mathsf{dom}\,\sigma = \mathsf{dom}\,\overline{\sigma}$ and

$$\overline{\sigma}(y) = \begin{cases} \{\sigma(y),\, \overline{\sigma}\}, & \text{if } \sigma(y) : \lambda\text{-}\mathsf{Abs}, \\ \sigma(y), & \text{if } \sigma(y) : \mathsf{Const}. \end{cases}$$

This definition is not circular, it is a well-defined coinductive definition. A thorough explanation of coinductive definitions and why they are well-defined, as well as similar examples of coinductive definitions, can be found in [Rut00, §11].

To state the relationship between capsules and closures, we define a binary relation $\sqsubseteq$ on capsule environments, closure environments, and values. For capsule environments, define $\sigma \sqsubseteq \tau$ if $\mathsf{dom}\,\sigma \subseteq \mathsf{dom}\,\tau$ and for all $y \in \mathsf{dom}\,\sigma$, $\sigma(y) = \tau(y)$. The definition for values and closure environments is by mutual coinduction: $\sqsubseteq$ is defined to be the largest relation such that

- on closure environments, $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau}$ if
  - $\mathsf{dom}\,\boldsymbol{\sigma} \subseteq \mathsf{dom}\,\boldsymbol{\tau}$, and
  - for all $y \in \mathsf{dom}\,\boldsymbol{\sigma}$, $\boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\tau}(y)$; and
- on values, $u \sqsubseteq v$ if either
  - u and v are constants and $u = v$; or
  - $u = \{\lambda x.e,\, \boldsymbol{\rho}\}$, $v = \{\lambda x.e,\, \boldsymbol{\pi}\}$, and $\boldsymbol{\rho} \sqsubseteq \boldsymbol{\pi}$.

Lemma 2.4.2 *The relation $\sqsubseteq$ is transitive.*

Proof. This is obvious for capsule environments.

For closure environments and values, we proceed by coinduction. Suppose $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau} \sqsubseteq \boldsymbol{\rho}$. Then $\mathsf{dom}\,\boldsymbol{\sigma} \subseteq \mathsf{dom}\,\boldsymbol{\tau} \subseteq \mathsf{dom}\,\boldsymbol{\rho}$, so $\mathsf{dom}\,\boldsymbol{\sigma} \subseteq \mathsf{dom}\,\boldsymbol{\rho}$, and for all $y \in \mathsf{dom}\,\boldsymbol{\sigma}$, $\boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\tau}(y) \sqsubseteq \boldsymbol{\rho}(y)$, therefore $\boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\rho}(y)$ by the transitivity of $\sqsubseteq$ on values.

For values, suppose $u \sqsubseteq v \sqsubseteq w$. If $u = c$, then $v = c$ and $w = c$. If $u = \{\lambda x.e,\, \boldsymbol{\sigma}\}$, then $v = \{\lambda x.e,\, \boldsymbol{\tau}\}$ and $w = \{\lambda x.e,\, \boldsymbol{\rho}\}$ and $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau} \sqsubseteq \boldsymbol{\rho}$, therefore $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\rho}$ by the transitivity of $\sqsubseteq$ on closure environments. □

Lemma 2.4.3 *Closure conversion is monotone with respect to $\sqsubseteq$. That is, if $\sigma \sqsubseteq \tau$, then $\overline{\sigma} \sqsubseteq \overline{\tau}$.*

Proof. We have $\mathsf{dom}\,\overline{\sigma} = \mathsf{dom}\,\sigma \subseteq \mathsf{dom}\,\tau = \mathsf{dom}\,\overline{\tau}$. Moreover, for $y \in \mathsf{dom}\,\sigma$,

$$\begin{aligned}
\overline{\sigma}(y) &= \begin{cases} \{\lambda x.e,\, \overline{\sigma}\}, & \text{if } \sigma(y) = \lambda x.e, \\ c, & \text{if } \sigma(y) = c \end{cases} &= \begin{cases} \{\lambda x.e,\, \overline{\sigma}\}, & \text{if } \tau(y) = \lambda x.e, \\ c, & \text{if } \tau(y) = c \end{cases} \\
&\sqsubseteq \begin{cases} \{\lambda x.e,\, \overline{\tau}\}, & \text{if } \tau(y) = \lambda x.e, \\ c, & \text{if } \tau(y) = c \end{cases} &= \overline{\tau}(y).
\end{aligned}$$

The $\sqsubseteq$ step in the above reasoning is by the coinduction hypothesis. □

Define a map $V : \mathsf{Cap} \to \mathsf{Val}$ on irreducible capsules as follows:

$$V(\lambda x.a, \sigma) = \{\lambda x.a,\, \overline{\sigma}\} \qquad\qquad V(c, \sigma) = c. \tag{2.18}$$

Lemma 2.4.4 $\overline{\sigma}(y) = V(\sigma(y), \sigma)$.

Proof.

$$\begin{aligned}
\overline{\sigma}(y) &= \begin{cases} \{\lambda x.e,\, \overline{\sigma}\}, & \text{if } \sigma(y) = \lambda x.e, \\ c & \text{if } \sigma(y) = c \end{cases} &= \begin{cases} V(\lambda x.e, \sigma), & \text{if } \sigma(y) = \lambda x.e, \\ V(c, \sigma) & \text{if } \sigma(y) = c \end{cases} \\
&= V(\sigma(y), \sigma).
\end{aligned}$$

□

Lemma 2.4.5 *If $y \notin \mathsf{dom}\,\sigma$, then $\overline{\sigma}[y/V(v,\sigma)] \sqsubseteq \overline{\sigma[y/v]}$.*

Proof. By Lemma 2.4.4,

$$\overline{\sigma[y/v]}(y) = V(\sigma[y/v](y), \sigma[y/v]) = V(v, \sigma[y/v]). \tag{2.19}$$

If $y \notin \mathsf{dom}\,\sigma$, then

$$\overline{\sigma}[y/V(v,\sigma)] \;\sqsubseteq\; \overline{\sigma[y/v]}[y/V(v,\sigma)] \;\sqsubseteq\; \overline{\sigma[y/v]}[y/V(v,\sigma[y/v])] \;=\; \overline{\sigma[y/v]},$$

the first two inequalities by Lemma 2.4.3 and the last equation by (2.19). □

Lemma 2.4.6 *If $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau}$, then $\mathsf{Eval_c}(e,\boldsymbol{\sigma})$ exists if and only if $\mathsf{Eval_c}(e,\boldsymbol{\tau})$ does, and $\mathsf{Eval_c}(e,\boldsymbol{\sigma}) \sqsubseteq \mathsf{Eval_c}(e,\boldsymbol{\tau})$. Moreover, they are derivable by the same large-step proofs.*

Proof. We proceed by induction on the proof tree under the large-step rules (2.15)–(2.17). For the single-step rules (2.15), we have

$$\begin{aligned}
&\mathsf{Eval_c}(c,\boldsymbol{\sigma}) = c = \mathsf{Eval_c}(c,\boldsymbol{\tau})\\
&\mathsf{Eval_c}(\lambda x.a,\boldsymbol{\sigma}) = \{\lambda x.a,\ \boldsymbol{\sigma}\} \sqsubseteq \{\lambda x.a,\ \boldsymbol{\tau}\} = \mathsf{Eval_c}(\lambda x.a,\boldsymbol{\tau})\\
&\mathsf{Eval_c}(y,\boldsymbol{\sigma}) = \boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\tau}(y) = \mathsf{Eval_c}(y,\boldsymbol{\tau}).
\end{aligned}$$

For the rule (2.16), $\langle d\ e,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f(c)$ is derivable by an application of (2.16) iff $\langle d,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f$ and $\langle e,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} c$ are derivable by smaller proofs. Similarly, $\langle d\ e,\ \boldsymbol{\tau}\rangle \xrightarrow[c]{*} f(c)$ is derivable by an application of (2.16) iff $\langle d,\ \boldsymbol{\tau}\rangle \xrightarrow[c]{*} f$ and $\langle e,\ \boldsymbol{\tau}\rangle \xrightarrow[c]{*} c$ are derivable by smaller proofs. By the induction hypothesis, $\langle d,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f$ and $\langle d,\ \boldsymbol{\tau}\rangle \xrightarrow[c]{*} f$ are derivable by the same proof, and similarly $\langle e,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} c$ and $\langle e,\ \boldsymbol{\tau}\rangle \xrightarrow[c]{*} c$ are derivable by the same proof.

Finally, for the rule (2.17), $\langle d\ e,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} u_1$ is derivable by an application of (2.17) iff $\langle d,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} \{\lambda x.a,\ \boldsymbol{\rho}_1\}$, $\langle e,\ \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} v_1$, and $\langle a[x/y],\ \boldsymbol{\rho}_1[y/v_1]\rangle \xrightarrow[c]{*} u_1$ are derivable by smaller proofs. Similarly, $\langle d\ e,\ \boldsymbol{\tau}\rangle \xrightarrow[c]{*} u_2$ is derivable by an application of (2.17) iff

$\langle d,\, \boldsymbol{\tau}\rangle \xrightarrow[c]{*} \{\lambda x.a,\, \boldsymbol{\rho}_2\}$, $\langle e,\, \boldsymbol{\tau}\rangle \xrightarrow[c]{*} v_2$, and $\langle a[x/y],\, \boldsymbol{\rho}_2[y/v_2]\rangle \xrightarrow[c]{*} u_2$ are derivable by smaller proofs. By the induction hypothesis, $\langle d,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} \{\lambda x.a,\, \boldsymbol{\rho}_1\}$ and $\langle d,\, \boldsymbol{\tau}\rangle \xrightarrow[c]{*} \{\lambda x.a,\, \boldsymbol{\rho}_2\}$ are derivable by the same proof, and $\boldsymbol{\rho}_1 \sqsubseteq \boldsymbol{\rho}_2$. Similarly, $\langle e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} v_1$ and $\langle e,\, \boldsymbol{\tau}\rangle \xrightarrow[c]{*} v_2$ are derivable by the same proof, and $v_1 \sqsubseteq v_2$. It follows that $\boldsymbol{\rho}_1[y/v_1] \sqsubseteq \boldsymbol{\rho}_2[y/v_2]$. Again by the induction hypothesis, $\langle a[x/y],\, \boldsymbol{\rho}_1[y/v_1]\rangle \xrightarrow[c]{*} u_1$ and $\langle a[x/y],\, \boldsymbol{\rho}_2[y/v_2]\rangle \xrightarrow[c]{*} u_2$ are derivable by the same proof, and $u_1 \sqsubseteq u_2$. □

The following theorem establishes the soundness of closure conversion for capsules.

Theorem 2.4.7 $\mathsf{Eval}(e,\sigma)$ *exists if and only if* $\mathsf{Eval_c}(e,\overline{\sigma})$ *does, and* $\mathsf{Eval_c}(e,\overline{\sigma}) \sqsubseteq V(\mathsf{Eval}(e,\sigma))$. *Moreover, they are derivable by isomorphic large-step proofs under the obvious correspondence between the large-step rules of both systems.*[3]

Proof. We proceed by induction on the proof tree under the large-step rules. The proof is similar to the proof of Lemma 2.4.6. We write $\xrightarrow[c]{*}$ for the derivability relation under the large-step rules (2.15)–(2.17) for closures to distinguish them from the corresponding large-step rules (2.8)–(2.10) for capsules, which we continue to denote by $\xrightarrow{*}$.

For the single-step rules (2.15), we have

$$\begin{aligned}
&\mathsf{Eval_c}(c,\overline{\sigma}) = c = V(\mathsf{Eval}(c,\sigma))\\
&\mathsf{Eval_c}(\lambda x.a,\overline{\sigma}) = \{\lambda x.a,\, \overline{\sigma}\} = V(\lambda x.a,\sigma) = V(\mathsf{Eval}(\lambda x.a,\sigma))\\
&\mathsf{Eval_c}(y,\overline{\sigma}) = \overline{\sigma}(y) = V(\sigma(y),\sigma) = V(\mathsf{Eval}(y,\sigma)).
\end{aligned}$$

The last line uses Lemma 2.4.4.

[3]For this purpose, the definition of V in (2.18) can be viewed as a pair of proof rules corresponding to the first two rules of (2.15).

Consider the corresponding rules (2.9) and (2.16). A conclusion $\langle d\ e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} f(c)$ is derivable by an application of (2.16) iff $\langle d,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} f$ and $\langle e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} c$ are derivable by smaller proofs. Similarly, $\langle d\ e,\, \sigma\rangle \overset{*}{\rightarrow} \langle f(c),\, \rho\rangle$ is derivable by an application of (2.9) iff $\langle d,\, \sigma\rangle \overset{*}{\rightarrow} \langle f,\, \sigma S\rangle$ and $\langle e,\, \sigma S\rangle \overset{*}{\rightarrow} \langle c,\, \sigma ST\rangle$ are derivable by smaller proofs.

By the induction hypothesis, $\langle d,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} f = V(f, \sigma S)$ and $\langle d,\, \sigma\rangle \overset{*}{\rightarrow} \langle f,\, \sigma S\rangle$ are derivable by isomorphic proofs. By Lemma 2.4.6, $\langle e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} c$ and $\langle e,\, \overline{\sigma S}\rangle \overset{*}{\underset{c}{\rightarrow}} c$ are derivable by the same proof. Again by the induction hypothesis, $\langle e,\, \overline{\sigma S}\rangle \overset{*}{\underset{c}{\rightarrow}} c$ and $\langle e,\, \sigma S\rangle \overset{*}{\rightarrow} \langle c,\, \sigma ST\rangle$ are derivable by isomorphic proofs, therefore so are $\langle e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} c = V(c, \sigma ST)$ and $\langle e,\, \sigma S\rangle \overset{*}{\rightarrow} \langle c,\, \sigma ST\rangle$.

Finally, consider the corresponding rules (2.10) and (2.17). A conclusion $\langle d\ e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} u$ is derivable by an application of (2.17) iff for some $\lambda x.a$, $\boldsymbol{\rho}$, and v,

$$\langle d,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} \{\lambda x.a,\, \boldsymbol{\rho}\} \qquad \langle e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} v \qquad \langle a[x/y],\, \boldsymbol{\rho}[y/v]\rangle \overset{*}{\underset{c}{\rightarrow}} u$$

are derivable by smaller proofs. Similarly, $\langle d\ e,\, \sigma\rangle \overset{*}{\rightarrow} \langle t,\, \tau\rangle$ is derivable by an application of (2.10) iff for some $\lambda z.b$, S, T, and w,

$$\langle d,\, \sigma\rangle \overset{*}{\rightarrow} \langle \lambda z.b,\, \sigma S\rangle \qquad \langle e,\, \sigma S\rangle \overset{*}{\rightarrow} \langle w,\, \sigma ST\rangle \qquad \langle b[z/y],\, \sigma ST[y/w]\rangle \overset{*}{\rightarrow} \langle t,\, \tau\rangle$$

are derivable by smaller proofs.

By the induction hypothesis, $\langle d,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} \{\lambda x.a,\, \boldsymbol{\rho}\}$ and $\langle d,\, \sigma\rangle \overset{*}{\rightarrow} \langle \lambda z.b,\, \sigma S\rangle$ are derivable by isomorphic proofs, and $\{\lambda x.a,\, \boldsymbol{\rho}\} \sqsubseteq V(\lambda z.b, \sigma S) = \{\lambda z.b,\, \overline{\sigma S}\}$, therefore $\lambda x.a = \lambda z.b$ and $\boldsymbol{\rho} \sqsubseteq \overline{\sigma S} \sqsubseteq \overline{\sigma ST}$.

By Lemmas 2.4.3 and 2.4.6, for some v', $\langle e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} v$ and $\langle e,\, \overline{\sigma S}\rangle \overset{*}{\underset{c}{\rightarrow}} v'$ are derivable by the same proof, and $v \sqsubseteq v'$. Again by the induction hypothesis, $\langle e,\, \overline{\sigma S}\rangle \overset{*}{\underset{c}{\rightarrow}} v'$ and $\langle e,\, \sigma S\rangle \overset{*}{\rightarrow} \langle w,\, \sigma ST\rangle$ are derivable by isomorphic proofs, and $v' \sqsubseteq V(w, \sigma ST)$. By transitivity, $\langle e,\, \overline{\sigma}\rangle \overset{*}{\underset{c}{\rightarrow}} v$ and $\langle e,\, \sigma S\rangle \overset{*}{\rightarrow} \langle w,\, \sigma ST\rangle$ are derivable by isomorphic

proofs, and $v \sqsubseteq V(w, \sigma ST)$. By Lemma 2.4.5,

$$\boldsymbol{\rho}[y/v] \;\sqsubseteq\; \overline{\sigma ST}[y/V(w, \sigma ST)] \;\sqsubseteq\; \overline{\sigma ST[y/w]}.$$

Again by Lemma 2.4.6, for some u', $\langle a[x/y],\, \boldsymbol{\rho}[y/v]\rangle \xrightarrow[c]{*} u$ and $\langle a[x/y],\, \overline{\sigma ST[y/w]}\rangle \xrightarrow[c]{*} u'$ are derivable by the same proof, and $u \sqsubseteq u'$; and again by the induction hypothesis, $\langle a[x/y],\, \overline{\sigma ST[y/w]}\rangle \xrightarrow[c]{*} u'$ and $\langle a[x/y],\, \sigma ST[y/w]\rangle \xrightarrow{*} \langle t,\, \tau\rangle$ are derivable by isomorphic proofs, and $u' \sqsubseteq V(t, \tau)$. By transitivity, $\langle a[x/y],\, \boldsymbol{\rho}[y/v]\rangle \xrightarrow[c]{*} u$ and $\langle a[x/y],\, \sigma ST[y/w]\rangle \xrightarrow[c]{*} \langle t,\, \tau\rangle$ are derivable by isomorphic proofs, and $u \sqsubseteq V(t, \tau)$.

□

2.5 A Functional/Imperative Language

In this section we give an operational semantics for a simply-typed higher-order functional and imperative language with mutable bindings. We thus fulfill the original desire to provide a unified semantics for functional and imperative languages. We might repeat some definitions as to have them all in a common place.

2.5.1 Expressions

Expressions $\mathsf{Exp} = \{d, e, a, b, \ldots\}$ contain both functional and imperative features. There is an unlimited supply of *variables* $x, y, \ldots$ of all (simple) types, as well as constants $f, c, \ldots$ for primitive values. $()$ is the only constant of type unit, and true and false are the only two constants of type bool. In the examples, $0, 1, 2, \ldots$ are predefined constants of type int. In addition, there are functional features

- λ-abstraction $\quad \lambda x.e$
- application $\quad (d\ e)$,

imperative features

- assignment $x := e$
- composition $d; e$
- conditional $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$
- while loop $\mathsf{while}\ b\ \mathsf{do}\ e$,

and syntactic sugars

- $\mathsf{let}\ x = d\ \mathsf{in}\ e$ $(\lambda x.e)\ d$
- $\mathsf{let\,rec}\ x = d\ \mathsf{in}\ e$ $\mathsf{let}\ x = a\ \mathsf{in}\ x := d; e$

where a is any expression of the appropriate type. The technique for formation of recursive functions in the last definition is known as *Landin's knot*.

Let Var be the set of variables, Const the set of constants, and λ-Abs the set of λ-abstractions. Given an expression e, let $\mathsf{FV}(e)$ denote the set of free variables of e. Given a partial function $h : \mathsf{Var} \rightharpoonup \mathsf{Var}$ such that $\mathsf{FV}(e) \subseteq \mathsf{dom}\, h$, let $h(e)$ be the expression e where every instance of a free variable $x \in \mathsf{FV}(e)$ has been replaced by the variable $h(x)$. $h : \mathsf{Exp} \rightharpoonup \mathsf{Exp}$ is the unique homomorphic extension of $h : \mathsf{Var} \rightharpoonup \mathsf{Var}$. Given two partial functions g and h, $g \circ h$ denotes their composition: $g \circ h(x) = g(h(x))$. Given a function h, we write $h[x/v]$ the function such that $h[x/v](y) = h(y)$ for $y \neq x$ and $h[x/v](x) = v$. Given an expression e, we write $e[x/y]$ the expression e with y substituted for all free occurrences of x.

2.5.2 Types

Types $\alpha, \beta, \ldots$ are ordinary simple types built inductively from an unspecified family of base types, including at least unit and bool, and the usual function type construc-

tor $\rightarrow$. All constants c of the language have a type $\mathsf{type}(c)$; by convention, we use c for a constant of a base type and f for a constant of a functional type. We follow [Win93] in assuming that each variable x is associated with a unique type $\mathsf{type}(x)$, that could for example be built into the variable name. Γ is a type environment, a partial function $\mathsf{Var} \rightharpoonup \mathsf{Type}$. As is standard, we write $\Gamma, x : \alpha$ for the typing environment Γ where x has been bound or rebound to α. The typing rules are standard:

$$\Gamma \vdash c : \alpha \text{ if } \mathsf{type}(c) = \alpha \qquad \Gamma, x : \alpha \vdash x : \alpha \qquad \frac{\mathsf{type}(x) = \alpha \quad \Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \rightarrow \beta}$$

$$\frac{\Gamma \vdash d : \alpha \rightarrow \beta \quad \Gamma \vdash e : \alpha}{\Gamma \vdash (d\ e) : \beta} \qquad \frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash x := e : \mathsf{unit}} \qquad \frac{\Gamma \vdash d : \mathsf{unit} \quad \Gamma \vdash e : \alpha}{\Gamma \vdash d; e : \alpha}$$

$$\frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash d : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e : \alpha} \qquad \frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \mathsf{while}\ b\ \mathsf{do}\ e : \mathsf{unit}}$$

Henceforth all the expressions we consider will be assumed to be well-typed with respect to these rules.

2.5.3 Small-Step Evaluation

Definitions

An expression is *irreducible* if it is either a constant or a λ-abstraction. Note that variables are not irreducible. Let $i, j, k, \ldots$ denote irreducible expressions, and $\mathsf{Irred} = \mathsf{Const} + \lambda\text{-}\mathsf{Abs}$ the set of irreducible expressions (These are often called *values* in the λ-calculus literature, but we avoid this terminology here because it is misleading, as they are not values in the intuitive sense.)

The *closure* of a set $A \subseteq \mathsf{dom}\,\gamma$ with respect to γ, denoted $\mathsf{cl}_\sigma(A)$, is the smallest set B containing A such that if $x \in B$ then $\mathsf{FV}(\gamma(x)) \subseteq B$. It is the domain of the

least-defined capsule environment whose domain contains A and that agrees with γ on its domain.

The term *α-conversion* refers to the renaming of bound variables. With a capsule $\langle e, \gamma\rangle$, this can happen in two ways. The traditional form maps a subterm $\lambda x.d$ to $\lambda y.d[x/y]$, provided y would not be captured in d. We call this *α-conversion of the first kind*. One can also rename a variable $x \in \mathsf{dom}\,\gamma$ and all free occurrences of x in e and $\gamma(z)$ for $z \in \mathsf{dom}\,\gamma$ to y, provided $y \notin \mathsf{dom}\,\gamma$ already and y would not be captured. We call this *α-conversion of the second kind*.

A capsule $\langle e, \gamma\rangle$ is *irreducible* if e is an irreducible expression. A *capsule value* is the equivalence class of an irreducible capsule modulo bisimilarity and α-conversion; equivalently, the λ-coterm represented by the capsule modulo α-conversion.

Small-step semantics

Capsules provide a particularly clean small-step semantics for the language studied here. A program determines a binary relation on capsules. The functional features are interpreted by the rules of §2.4.1, that we repeat here for completeness.

Let the operator $\to_{\mathsf{ca}}$ relate capsules. One rule of particular note is the assignment rule

$$\langle y := i,\, \gamma\rangle \to_{\mathsf{ca}} \langle (),\, \gamma[y/i]\rangle$$

The closure conditions on capsules ensure that y must already be bound in γ. The variable y is rebound to the irreducible expression i. The semantics of other features directly involving variables is given by:

$$\langle x,\, \gamma\rangle \to_{\mathsf{ca}} \langle \gamma(x),\, \gamma\rangle \qquad \langle (\lambda x.b)\ i,\, \gamma\rangle \to_{\mathsf{ca}} \langle b[x/y],\, \gamma[y/i]\rangle \quad (y \text{ fresh})$$

and the remaining semantics is:

$$\langle f\ c,\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle f(c),\ \gamma\rangle \qquad \langle (); e,\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle e,\ \gamma\rangle$$

$$\langle \mathsf{if\ true\ then}\ d\ \mathsf{else}\ e,\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle d,\ \gamma\rangle \qquad \langle \mathsf{if\ false\ then}\ d\ \mathsf{else}\ e,\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle e,\ \gamma\rangle$$

$$\langle \mathsf{while}\ b\ \mathsf{do}\ e,\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle \mathsf{if}\ b\ \mathsf{then}\ (e; \mathsf{while}\ b\ \mathsf{do}\ e)\ \mathsf{else}\ (),\ \gamma\rangle$$

Evaluation contexts C are defined by:

$$C ::= \quad [\,\cdot\,] \quad | \quad C\ e \quad | \quad i\ C \quad | \quad x := C \quad | \quad C; e \quad | \quad \mathsf{if}\ C\ \mathsf{then}\ d\ \mathsf{else}\ e$$

where each evaluation context $C[\,\cdot\,]$ generates a rule:

$$\frac{\langle d,\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle e,\ \delta\rangle}{\langle C[d],\ \gamma\rangle \rightarrow_{\mathsf{ca}} \langle C[e],\ \delta\rangle}$$

These context rules define a standard shallow applicative-order (leftmost innermost, call-by-value) evaluation strategy.

The reduction rules preserve types and cannot fail catastrophically. Thus every computation either continues forever or terminates with a well-typed final capsule $\langle i,\ \sigma\rangle$, where i is irreducible.

The relation $\xrightarrow{*}_{\mathsf{ca}}$ is the reflexive transitive closure of $\rightarrow_{\mathsf{ca}}$. If d and e are closed terms, we write $d \rightarrow_{\mathsf{ca}} e$ to mean $\langle d,\ [\,]\rangle \rightarrow_{\mathsf{ca}} \langle e,\ \gamma\rangle$ for some γ, and $d \xrightarrow{*}_{\mathsf{ca}} e$ to mean $\langle d,\ [\,]\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e,\ \gamma\rangle$ for some γ.

Examples

The following examples show that lexical scoping and recursion are handled correctly. The complete derivation of each example is given in Appendix A.

Example 2.5.1 $(\mathsf{let}\ x = 1\ \mathsf{in\ let}\ f = \lambda y.x\ \mathsf{in\ let}\ x = 2\ \mathsf{in}\ f\ 0) \xrightarrow{*}_{\mathsf{ca}} 1$ □

Example 2.5.2 $(\mathsf{let}\ x = 1\ \mathsf{in\ let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \xrightarrow{*}_{\mathsf{ca}} 2$ □

Example 2.5.3 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = (\mathsf{let}\ x = 2\ \mathsf{in}\ \lambda y.x)\ \mathsf{in}\ f\ 0) \xrightarrow{*}_{\mathsf{ca}} 2$ □

Example 2.5.4 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f := \lambda y.x; f\ 0) \xrightarrow{*}_{\mathsf{ca}} 2$ □

Example 2.5.5 $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3) \xrightarrow{*}_{\mathsf{ca}} 6$ □

2.5.4 Garbage Collection

During a computation, some variables can become superfluous, either because they are out of scope or because they are not called later. In this section we show how to garbage collect them.

A *monomorphism* $h : \langle d, \sigma \rangle \to \langle e, \tau \rangle$ is an injective map $h : \mathsf{dom}\,\sigma \to \mathsf{dom}\,\tau$ such that

- $\tau(h(x)) = h(\sigma(x))$ for all $x \in \mathsf{dom}\,\sigma$, where $h(e) = e[x/h(x)]$ (safe substitution); and
- $h(d) = e$.

The collection of monomorphic preimages of a given capsule contains an initial object that is unique up to α-conversion. This is the *garbage collected* version of the capsule.

There exists a simple garbage collection algorithm for a capsule $\langle d, \sigma \rangle$: starting from expression d, collect all free variables appearing in d and mark them as used in σ; also put those variables in a bag b. Then, for every variable x in the bag b, collect all free variables y appearing in $\sigma(x)$; if y is already marked as used in σ, do nothing, otherwise mark y as used in σ and add it to b. Finish when there are no more variables in b. Let σ' be σ where we keep only the marked variables; then $\langle d, \sigma' \rangle$ is the garbage collected version of $\langle d, \sigma \rangle$.

2.5.5 Big-Step Evaluation

A big-step semantics is easier to relate to a semantics of closures, as we will see in the next chapter. Let us thus define a big step semantics where the operator $\Downarrow_{\mathsf{ca}}$ relates capsules to irreducible capsules. The semantics of features directly involving variables is given by:

$$\langle x,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle\gamma(x),\,\gamma\rangle \qquad \langle\lambda x.e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle\lambda x.e,\,\gamma\rangle \qquad \frac{\langle e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle j,\,\zeta\rangle}{\langle x:=e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle(),\,\zeta[x/j]\rangle}$$

$$\frac{\langle d,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle\lambda x.a,\,\zeta\rangle \qquad \langle e,\,\zeta\rangle\Downarrow_{\mathsf{ca}}\langle j,\,\eta\rangle \qquad \langle a[x/y],\,\eta[y/j]\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}{\langle d\ e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}\ (y\text{ fresh})$$

and the remaining semantics is:

$$\langle c,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle c,\,\gamma\rangle \qquad \frac{\langle d,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle f,\,\zeta\rangle \qquad \langle e,\,\zeta\rangle\Downarrow_{\mathsf{ca}}\langle c,\,\delta\rangle}{\langle d\ e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle f(c),\,\delta\rangle}$$

$$\frac{\langle d,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle(),\,\zeta\rangle \qquad \langle e,\,\zeta\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}{\langle d;e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}$$

$$\frac{\langle b,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle\mathsf{true},\,\zeta\rangle \qquad \langle d,\,\zeta\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}{\langle\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle} \qquad \frac{\langle b,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle\mathsf{false},\,\zeta\rangle \qquad \langle e,\,\zeta\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}{\langle\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\delta\rangle}$$

$$\frac{\begin{array}{c}\langle b,\,\gamma_i\rangle\Downarrow_{\mathsf{ca}}\langle\mathsf{true},\,\delta_i\rangle \qquad \langle e,\,\delta_i\rangle\Downarrow_{\mathsf{ca}}\langle(),\,\gamma_{i+1}\rangle,\ 0\le i<n,\ n\ge 0\\ \langle b,\,\gamma_n\rangle\Downarrow_{\mathsf{ca}}\langle\mathsf{false},\,\delta_n\rangle\end{array}}{\langle\mathsf{while}\ b\ \mathsf{do}\ e,\,\gamma_0\rangle\Downarrow_{\mathsf{ca}}\langle(),\,\delta_n\rangle}$$

Property 2.5.6 *If $\langle d,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle e,\,\delta\rangle$ then $\langle e,\,\delta\rangle$ is an irreducible capsule.*

Proof. By trivial structural induction on the derivation of $\langle d,\,\gamma\rangle\Downarrow_{\mathsf{ca}}\langle e,\,\delta\rangle$. □

If d is a closed term and i is an irreducible close term, we write $d\Downarrow_{\mathsf{ca}}i$ to mean $\langle d,\,[\ ]\rangle\Downarrow_{\mathsf{ca}}\langle i,\,\gamma\rangle$ for some γ.

Examples

As expected, small-step and big-step semantics agree. Before proving it formally, let us show it on a few examples:

Example 2.5.7 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0)\Downarrow_{\mathsf{ca}} 1$ □

Example 2.5.8 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0)\Downarrow_{\mathsf{ca}} 2$ □

Example 2.5.9 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = (\mathsf{let}\ x = 2\ \mathsf{in}\ \lambda y.x)\ \mathsf{in}\ f\ 0)\Downarrow_{\mathsf{ca}} 2$ □

Example 2.5.10 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f := \lambda y.x; f\ 0)\Downarrow_{\mathsf{ca}} 2$ □

Example 2.5.11 $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3)\Downarrow_{\mathsf{ca}} 6$ □

Equivalence of big-step and small-step semantics

Theorem 2.5.12 *For a capsule* $\langle e,\, \gamma\rangle$ *and an irreducible capsule* $\langle i,\, \delta\rangle$, $\langle e,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle$ *if and only if* $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\, \delta\rangle$

Proof. The direct implication is proved by a simple structural induction on the big-step derivation of $\langle e,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle$. The converse if proved by recurrence on the number k of steps of the derivation of $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\, \delta\rangle$, with base case for $k = 1$; the case $k = 0$ is proved separately.

($\Rightarrow$) If e is irreducible (a λ-abstraction $\lambda x.d$ or a constant c), then $\langle e,\, \gamma\rangle = \langle i,\, \delta\rangle$ and $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\, \delta\rangle$ in zero step. If e is a variable x, $i = \gamma(x)$, $\gamma = \delta$, and $\langle e,\, \gamma\rangle \rightarrow_{\mathsf{ca}} \langle \gamma(x),\, \delta\rangle$, thus $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle \gamma(x),\, \delta\rangle$ in one step. The other cases are easily handled by structural induction on the big-step derivation of $\langle e,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle$. The case for the while loop involves a simple recurrence on the number of iterations n.

($\Leftarrow$) The proof is very similar. If e is irreducible, then $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\, \delta\rangle$ was performed in zero step, so $\langle e,\, \gamma\rangle = \langle i,\, \delta\rangle$ and $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\, \delta\rangle$. If e is a variable x, then $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle \gamma(x),\, \delta\rangle$ in one step and $\langle e,\, \gamma\rangle {\Downarrow_{\mathsf{ca}}} \langle \gamma(x),\, \delta\rangle$. The other cases are easily handled by case analysis on the first small-step of the derivation $\langle e,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\, \delta\rangle$. The case for the while loop involves a recurrence on the number of times the small-step rule for the while loop is applied.

□

Corollary 2.5.13 *If* $\langle d,\, \gamma\rangle {\Downarrow_{\mathsf{ca}}} \langle e,\, \delta\rangle$ *then* $\langle d,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e,\, \delta\rangle$

Proof. It is a direct consequence of property 2.5.6 and theorem 2.5.12. The converse is not true; a counterexample is given by any $\langle e,\, \delta\rangle$ not irreducible. □

2.6 Conclusion

Capsules provide an algebraic representation of state for higher-order functional and imperative programs. They are mathematically simpler than closures and correctly model static scope without auxiliary data constructs, even in the presence of recursion and mutable variables. Capsules form a natural coalgebraic extension of the λ-calculus, and we have shown how coalgebraic techniques can be brought to bear on arguments involving state. We have shown that capsule evaluation is faithful to β-reduction with safe substitution in the λ-calculus. We have shown how to closure-convert capsules, and we have proved soundness of the transformation in the absence of assignments. Finally, we have shown how capsules can be used to give a natural operational semantics to a higher-order functional and imperative language with mutable bindings.

Capsules have also been used to model objects [Koz12]. In the next chapter, we see how the relationship between capsules and closures established in Theorem 2.4.7 holds in the presence of assignment, after introducing appropriate extensions to the definition of closure to allow indirection. In chapter 4 we provide a semantics for separation logic using capsules.

Chapter 3

Capsules and Closures

This chapter precisely compares capsules and closures for a higher-order language with mutable bindings. It provides two comparisons: a comparison based on big-step semantics, which is simpler and thus allows us to see more easily how capsules and closures precisely relate; and another comparison based on small-step semantics, more complicated but ensuring soundness of capsules even for infinite computations.

3.1 Introduction

This chapter compares *Capsules* and *Closures*, including proofs of bisimilarity, in the semantics of a higher-order programming language with mutable variables. In proving soundness of one to the other, it gives a precise account of how capsule environments and closure environments relate to each other. It provides proofs both using big-step semantics and small-step semantics. Big-step semantics are simpler and thus allow to see more easily how capsules and closures precisely relate. However, while big-step semantics only allow to talk about final results of terminating computations, the use of small-step semantics allows to prove a stronger bisimi-

larity involving every step of the computation and thus also applicable to infinite computations.

The language used in this chapter was introduced in §2.5. It is both functional and imperative: it has higher-order functions, but every variable is mutable. This leads to interesting interactions and allows to go further than just enforcing lexical scoping. In particular, what do we expect the result of an expression like ($\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0$) to be? Scheme (using set! for :=) and OCaml (using references) answer 2. Capsules give a rigorous mathematical definition that agrees and conservatively extends the scoping rules of the λ-calculus. Our semantics of closures also agrees with this definition, but this requires introducing a level of indirection, with both a stack of environments and a store, à la ML. Finally, recursive definitions are often implemented using some sort of back-patching; we build this directly into the definition of the language by defining $\mathsf{let\ rec}\ x = d\ \mathsf{in}\ e$ as a syntactic sugar for $\mathsf{let}\ x = a\ \mathsf{in}\ x := d; e$, where a is any expression of the appropriate type.

This chapter is organized as follows. In §3.2, we describe a semantics based on closures for this language, which is an alternative to the capsule-based semantics presented in §2.5. In §3.3, we show a very strong correspondence (Theorem 3.3.5 in big-step and corollary 3.3.13 in small-step) between the two semantics, showing that every computation in the semantics of capsules is bisimilar to a computation in the semantics of closures, and vice-versa. In §3.4, we show (Propositions 3.4.1–3.4.4) that closure semantics retains some unnecessary information that capsule semantics omits, attesting of the simplicity of capsules. We finish with a discussion in §3.5.

3.2 Closure semantics

In this section we present a big-step semantics and a small-step semantics on *closures*. The semantics on closures is the semantics usually used and taught for functional languages. A level of indirection for variables has been added to support imperative features, *à la* ML. The big-step semantics is simpler, but does not say anything about nonterminating computation. The small-step semantics, while more complicated, allows to reason about infinite traces.

All the expressions we consider in this section are supposed well-typed with the rules of §2.5.2.

3.2.1 Definitions

Closures were introduced in the language Scheme [SS98]. We present a version of them using a level of indirection, allowing us to handle mutable variables. For clarity purposes, we repeat a few definitions here.

There is an unlimited number of locations $\ell, \ell_1, \ell_2 \ldots$; locations can be thought of as addresses in memory. An *environment* is a partial function from variables to locations. A *closure* is defined as a pair $\{\lambda x.e, \sigma\}$ such that $\mathsf{FV}(\lambda x.e) \subseteq \mathsf{dom}\,\sigma$, where $\lambda x.e$ is a λ-abstraction and σ is an environment that is used to interpret the free variables of $\lambda x.e$. A *value* is either a constant or a closure. Values for closures play the same role as irreducible terms for capsules. A *store* (or *memory*) is a partial function from locations to values.

Let $u, v, w, \ldots$ denote values, $\sigma, \tau, \ldots$ environments and $\mu, \nu, \xi, \chi, \ldots$ stores. Let Val be the set of values, Loc the set of locations and Cl the set of closures. Thus we

have:

$$\sigma : \mathsf{Var} \rightharpoonup \mathsf{Loc} \qquad \mu : \mathsf{Loc} \rightharpoonup \mathsf{Val} \qquad \mathsf{Val} = \mathsf{Const} + \mathsf{Cl}$$

3.2.2 Big-step

Semantics

A *state* is a triple $\langle e,\, \sigma,\, \mu\rangle$. A state is *valid* if and only if

$$\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma \qquad \mathsf{codom}\,\sigma \subseteq \mathsf{dom}\,\mu$$

$$\forall \{\lambda x.a,\, \tau\} \in \mathsf{codom}\,\mu,\, \mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\,\tau \wedge \mathsf{codom}\,\tau \subseteq \mathsf{dom}\,\mu$$

A *result* is a pair $(v,\, \mu)$. A result is *valid* if and only if either $v \in \mathsf{Const}$, or $v = \{\lambda x.a, \tau\} \in \mathsf{Cl}$ and the triple $\langle \lambda x.a,\, \tau,\, \mu\rangle$ is valid. We only consider valid states and results. Let us define a big step semantics where the operator $\Downarrow_{\mathsf{cl}}$ relates valid states to valid results. The semantics of features directly involving variables is given by:

$$\langle x,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\mu(\sigma(x)),\, \mu) \qquad \langle \lambda x.e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\{\lambda x.e,\, \sigma\},\, \mu)$$

$$\frac{\langle e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (v,\, \xi)}{\langle x := e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} ((),\, \xi[\sigma(x)/v])}$$

$$\frac{\langle d,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\{\lambda x.a,\, \tau\},\, \xi) \qquad \langle e,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (v,\, \chi) \qquad \langle a,\, \tau[x/\ell],\, \chi[\ell/v]\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}{\langle d\ e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}\ (\ell \text{ fresh})$$

and the remaining semantics is:

$$\langle c,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (c,\, \mu) \qquad \frac{\langle d,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (f,\, \xi) \qquad \langle e,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (c,\, \nu)}{\langle d\ e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (f(c),\, \nu)}$$

$$\frac{\langle d,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} ((),\, \xi) \qquad \langle e,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}{\langle d; e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}$$

$$\frac{\langle b,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\mathsf{true},\, \xi) \qquad \langle d,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}{\langle \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}$$

$$\frac{\langle b,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\mathsf{false},\, \xi) \qquad \langle e,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}{\langle \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)}$$

$$\frac{\begin{array}{c}\langle b,\, \sigma,\, \mu_i\rangle \Downarrow_{\mathsf{cl}} (\mathsf{true},\, \nu_i) \qquad \langle e,\, \sigma,\, \nu_i\rangle \Downarrow_{\mathsf{cl}} ((),\, \mu_{i+1}),\ 0 \le i < n,\ n \ge 0 \\ \langle b,\, \sigma,\, \mu_n\rangle \Downarrow_{\mathsf{cl}} (\mathsf{false},\, \nu_n)\end{array}}{\langle \mathsf{while}\ b\ \mathsf{do}\ e,\, \sigma,\, \mu_0\rangle \Downarrow_{\mathsf{cl}} ((),\, \nu_n)}$$

Examples

Example 3.2.1 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0) \Downarrow_{\mathsf{cl}} 1$ □

Example 3.2.2 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \Downarrow_{\mathsf{cl}} 2$ □

Example 3.2.3 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f := \lambda y.x; f\ 0) \Downarrow_{\mathsf{cl}} 2$ □

Example 3.2.4 $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \mathsf{in}\ f\ 3) \Downarrow_{\mathsf{cl}} 6$ □

3.2.3 Small-step

A first try

At first, it seems that a small-step semantics for closures should not be much more complicated than its big-step counterpart. However some issue arises on the rule for the application $(d\ e)$ when d has already been reduced to a λ-term and e to a value.

Using closures, we are trying to take the next small step in the state $\langle \{\lambda x.a,\, \tau\}\ v,\, \sigma,\, \mu\rangle$. We would like to write something like:

$$\langle \{\lambda x.a,\, \tau\}\ v,\, \sigma,\, \mu\rangle \to_{\mathsf{cl}} \langle a,\, \tau[x/\ell],\, \mu[\ell/v]\rangle \quad (\ell \text{ fresh})$$

This rule is wrong: it drops the environment σ, but when this evaluation is in context, σ has to come back once we finish evaluating a. One solution is to write a rule involving several small steps, which is really a big step rule. Another solution is to keep track of the whole stack of environments to come back to the previous environment each time we get out of a scope: this is what we do next.

Using capsules however, the following rule comes very naturally:

$$\langle(\lambda x.a)\ i,\ \gamma\rangle \to_{\mathsf{ca}} \langle a[x/y],\ \gamma[y/i]\rangle \qquad (y \text{ fresh})$$

Along with the other small-step rules, this shows that the capsule semantics is fully relational and does not need any stack or auxiliary data structure.

Environment stacks

The interaction of small-step semantics and closures leads to using stacks of environments: when entering the body of a function, the environment coming with its closure is pushed; and when leaving this body, it is popped. Let $\Sigma, \Pi, \ldots$ denote stacks of environments. Let us write $[\sigma]$ the stack of environments containing only the element σ, as to not confuse it with the single environment σ. $\sigma :: \tau$ represents the stack containing σ at the top of the stack and τ at its bottom. $\sigma :: \Sigma$ represents the stack Σ with σ added on top of it; and $\Sigma :: \sigma$ represents Σ with σ added at its bottom. We define $\mathsf{hd}(\Sigma) = \sigma$ and $\mathsf{tl}(\Sigma) = \Pi$ whenever $\Sigma = \sigma :: \Pi$.

State expressions

To define a small-step semantics of closures, we need to represent all the different shapes an expression can take throughout computation, until it becomes a value. State expressions $\mathsf{StExp} = \{s, t, \ldots\}$ allow to do this. Of course, expressions and

values are state expressions, but some state expressions are neither.

$$\mathsf{Exp} \subseteq \mathsf{StExp} \qquad \qquad \mathsf{Val} \subseteq \mathsf{StExp}$$

A *state expression* can be:

- an expression e; this includes constants c;
- a closure $\{\lambda x.a,\, \sigma\}$;
- a state expression followed by a pop indicator $\square$, as in $s\ \square$; when entering the body of a function, a new environment needs to be pushed on the stack of environments; this environment needs to be popped when leaving the body of the function; one way to know when this happens is to keep track of the end of the body with $\square$;
- a state expression applied to an expression, $s\ e$;
- a value applied to a state expression, $v\ s$;
- an assignment, $x := s$;
- a composition $s; e$;
- an if statement $\mathsf{if}\ s\ \mathsf{then}\ d\ \mathsf{else}\ e$.

We extend the notion of free variables to a state expression in a natural, syntactic way: for a well-formed closure $\{\lambda x.a,\, \sigma\}$ with $\mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\,\sigma$, $\mathsf{FV}(\{\lambda x.a,\, \sigma\})$ is the empty set; and $\mathsf{FV}(s\ \square) = \mathsf{FV}(s)$.

Stacks of environments, along with the introduction of closures and of the pop indicator $\square$, are a convenient way to model in which environment each variable should be looked up. Intuitively, any free variable in a λ-abstraction of a closure

should be interpreted in the environment coming with this closure. Because of the definition of state expressions, the pop indicators all are inside each other. The variables inside the deepest pop indicator are interpreted in the environment on top the stack; the variables inside the second deepest but outside the deepest pop indicator are interpreted in the second environment from the top of the stack, and so on. The variables outside of any pop indicator are interpreted in the environment at the bottom of the stack. A precise account of this idea will be given in §3.3.1 with the definition of $h \circ \Sigma$.

Semantics

A *state* is a triple $\langle s,\, \Sigma,\, \mu\rangle$.

$$\mathsf{FV}(s) \subseteq \mathsf{dom}\,(\mathsf{hd}\,\Sigma) \qquad\qquad \forall\sigma \in \Sigma, \mathsf{codom}\,\sigma \subseteq \mathsf{dom}\,\mu$$

$$\forall\{\lambda x.a,\, \tau\} \in \mathsf{codom}\,\mu,\, \mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\,\tau \wedge \mathsf{codom}\,\tau \subseteq \mathsf{dom}\,\mu$$

When evaluating an expression e, we start with the initial state $\langle e,\, [\sigma],\, \mu\rangle$ where σ and μ are both empty mappings. Let us define a small step semantics where the operator $\rightarrow_{\mathsf{cl}}$ relates valid states to valid results. The semantics of features directly involving variables is given by:

$$\langle x,\, [\sigma],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle \mu(\sigma(x)),\, [\sigma],\, \mu\rangle \qquad \langle \lambda x.a,\, [\sigma],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle \{\lambda x.a,\, \sigma\},\, [\sigma],\, \mu\rangle$$

$$\langle \{\lambda x.a, \sigma\}\ v,\, [\tau],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle a\,\square,\, \sigma[x/\ell] :: \tau,\, \mu[\ell/v]\rangle \quad (\ell \text{ fresh})$$

$$\langle v\,\square,\, \sigma :: \tau,\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle v,\, [\tau],\, \mu\rangle \qquad \langle x := v,\, [\sigma],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle (),\, [\sigma],\, \mu[\sigma(x)/v]\rangle$$

and the remaining semantics is:

$$\langle f\ c,\, [\sigma],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle f(c),\, [\sigma],\, \mu\rangle \qquad \langle (); e,\, [\sigma],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle e,\, [\sigma],\, \mu\rangle$$

$$\langle \mathsf{if\ true\ then}\ d\ \mathsf{else}\ e,\, [\sigma],\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle d,\, [\sigma],\, \mu\rangle$$

$$\langle \textsf{if false then } d \textsf{ else } e,\ [\sigma],\ \mu\rangle \rightarrow_{\textsf{cl}} \langle e,\ [\sigma],\ \mu\rangle$$

$$\langle \textsf{while } b \textsf{ do } e,\ [\sigma],\ \mu\rangle \rightarrow_{\textsf{cl}} \langle \textsf{if } b \textsf{ then } (e; \textsf{while } b \textsf{ do } e) \textsf{ else } (),\ [\sigma],\ \mu\rangle$$

Evaluation contexts C are defined by:

$$C ::= \quad [\,\cdot\,] \quad | \quad C\ e \quad | \quad v\ C \quad | \quad x := C \quad | \quad C; e \quad | \quad \textsf{if } C \textsf{ then } d \textsf{ else } e$$

where each evaluation context $C[\,\cdot\,]$ generates a rule:

$$\frac{\langle s,\ \Sigma,\ \mu\rangle \rightarrow_{\textsf{cl}} \langle t,\ \Pi,\ \nu\rangle}{\langle C[s],\ \Sigma,\ \mu\rangle \rightarrow_{\textsf{cl}} \langle C[t],\ \Pi,\ \nu\rangle}$$

One more rule is needed to be able to evaluate under a pop indicator $\Box$:

$$\frac{\langle s,\ \Sigma,\ \mu\rangle \rightarrow_{\textsf{cl}} \langle t,\ \Pi,\ \nu\rangle}{\langle s\ \Box,\ \Sigma :: \sigma,\ \mu\rangle \rightarrow_{\textsf{cl}} \langle t\ \Box,\ \Pi :: \sigma,\ \nu\rangle}$$

Note the similarity between the last two rules, including the definition of evaluation contexts, and the inductive definition of state environments. This is not by chance: the innermost state expression, if not a value, is always the one which will be evaluated next.

The final states, i.e., the states that cannot take a small step, are the (v, Σ, μ) for any value v.

As usual, we introduce $\xrightarrow{*}_{\textsf{cl}}$ as the reflexive transitive closure of $\rightarrow_{\textsf{cl}}$.

Properties

Some properties of this semantics can be easily proved:

- In a state expression s, all the pop indicators $\Box$ are inside each other; the deepest one is inside all the others, and so on.
- If starting from an initial state, the number of elements on the environment stack Σ is always one more than the number of pop indicators $\Box$ in s.
- if $\Sigma \rightarrow_{\textsf{cl}} \Pi$ then either $\Sigma = \Pi$ or $\Sigma = \sigma :: \Pi$ or $\sigma :: \Sigma = \Pi$ for some σ.

Examples

To illustrate the above semantics, we now show the evaluation of the examples of §2.5.3 using closures. The complete derivation of each example is given in Appendix A.

Example 3.2.5 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0) \xrightarrow{*}_{\mathsf{cl}} 1$ □

Example 3.2.6 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \xrightarrow{*}_{\mathsf{cl}} 2$ □

This final example is particularly interesting as it shows how nested □ allow to interpret the same variable in different scopes. In all the example e stands for $\{\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n, [f = \ell_1]\}$, and d stands for $\{\lambda n.n, [\ ]\}$, a dummy value used when creating the recursive function f.

Example 3.2.7 $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3) \xrightarrow{*}_{\mathsf{cl}} 6$ □

3.3 Equivalence of the semantics

3.3.1 Definitions

There is a very strong correspondence between the semantics of closures and capsules, both in small steps and in big steps. To give a precise account of this correspondence, we introduce an injective partial function $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ with which we define five relations. Each relation is between an element of the semantics of closures and an element of the semantics of capsules that play similar roles. Relations 1 to 4 below are relevant for comparing big-step semantics, while relations 1, 2 and 5 are relevant for comparing small-step semantics:

1. $v \xrightarrow{h} i$ between values and irreducible terms;

2. $\mu \xrightarrow{h} \gamma$ between stores and capsule environments;

3. $\langle d,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e,\, \gamma\rangle$ between big-step states and capsules;

4. $(v,\, \mu) \overset{h}{\sim} \langle i,\, \gamma\rangle$ between results and irreducible capsules;

5. $\langle s,\, \Sigma,\, \mu\rangle \overset{h}{\sim} \langle e,\, \gamma\rangle$ between small-step states and capsules.

One thing to notice is that nothing in the semantics of capsules plays the same role as the environment σ in the big-step semantics of closures, or the environment stack Σ in their small-step semantics: capsule environments γ relate to memories μ, and environments σ have been simplified. Let us now give precise definitions of those relations.

Definition 3.3.1 Given a value v and an irreducible term i, we say that h *transforms v into i*, where h is an injective map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$, and we write $v \xrightarrow{h} i$, if and only if:

- $v = i$ when $v \in \mathsf{Const}$, or
- $\mathsf{codom}\,\tau \subseteq \mathsf{dom}\,h$ and $(h \circ \tau)(\lambda x.a) = i$ when $v = \{\lambda x.a,\, \tau\} \in \mathsf{Cl}$

$\square$

Definition 3.3.2 Given a store μ and a capsule environment γ, we say that h *transforms μ into γ*, where h is an injective map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$, and we write $\mu \xrightarrow{h} \gamma$, if and only if:

$$\mathsf{dom}\,h = \mathsf{dom}\,\mu \qquad h(\mathsf{dom}\,\mu) = \mathsf{dom}\,\gamma$$

$$\forall \ell \in \mathsf{dom}\,\mu,\ \mu(\ell) \xrightarrow{h} \gamma(h(\ell))$$

$\square$

Throughout this section, we focus on the features directly involving variables: variable calls x, λ-abstractions $\lambda x.e$, applications $(d\ e)$ where d reduces to a λ-abstraction, and assignment $x := e$. Most differences between capsules and closures arise when using these features.

3.3.2 Big-step

Before proceeding to the equivalence of big-step semantics, we need a few more definitions that are specific to the big-step case.

Definition 3.3.3 Given a state $\langle d, \sigma, \mu\rangle$ and a capsule $\langle e, \gamma\rangle$, both valid, we say that they are *bisimilar under* h, where h is an injective map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$, and we write $\langle d, \sigma, \mu\rangle \stackrel{h}{\sim} \langle e, \gamma\rangle$, if and only if

$$(h \circ \sigma)(d) = e \qquad\qquad \mu \stackrel{h}{\to} \gamma$$

$\square$

Definition 3.3.4 Given a result (v, μ) and an irreducible capsule $\langle i, \gamma\rangle$, both valid, we say that they are *bisimilar under* h, where h is an injective map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$, and we write $(v, \mu) \stackrel{h}{\sim} \langle i, \gamma\rangle$ if and only if:

$$v \stackrel{h}{\to} i \qquad\qquad \mu \stackrel{h}{\to} \gamma$$

$\square$

Now that we know how to relate each element of both semantics, theorem 3.3.5 shows that any derivation using capsules mirrors a derivation using closures, and vice-versa:

Theorem 3.3.5 *If* $\langle d, \sigma, \mu\rangle \overset{h}{\sim} \langle e, \gamma\rangle$ *then* $\langle d, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (u, \nu)$ *for some* u, ν *if and only if* $\langle e, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i, \delta\rangle$ *for some* i, δ, *and in that case we have*

$$(u, \nu) \overset{g}{\sim} \langle i, \delta\rangle$$

where g *is an extension of* h, *i.e.,* $\mathsf{dom}\, h \subseteq \mathsf{dom}\, g$ *and* h *and* g *agree on* $\mathsf{dom}\, h$.

Proof. We show the direct implication by induction on the big-step derivation of $\langle d, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (u, \nu)$ and the converse by induction on the big-step derivation of $\langle e, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i, \delta\rangle$.

We show the most interesting cases of the induction first: variable x, λ-abstraction $\lambda x.e$, function application of a λ-abstraction $d\ e$ where d reduces to a λ-abstraction, and variable assignment $x := e$. In all these cases, both implications are very similar proofs. We then proceed to the other cases, constant c, function application of a constant function $d\ e$ where d reduces to a constant f, composition $d; e$, if conditional $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$ and while loop $\mathsf{while}\ b\ \mathsf{do}\ e$.

Variable If $d = x$ for some variable x then $e = (h \circ \sigma)(d) = y$ with y the variable such that $y = (h \circ \sigma)(x)$.

($\Rightarrow$) By definition of $\Downarrow_{\mathsf{cl}}$, $(u, \nu) = (\mu(\sigma(x)), \mu)$, and by definition of $\Downarrow_{\mathsf{ca}}$, $\langle e, \gamma\rangle = \langle y, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle \gamma(y), \gamma\rangle$. Moreover $\mu \overset{h}{\to} \gamma$, therefore by definition of $\overset{h}{\to}$, $\mu(\sigma(x)) \overset{h}{\to} \gamma(h(\sigma(x))) = \gamma(y)$. Therefore, with $g = h$, $(u, \nu) = (\mu(\sigma(x)), \mu) \overset{g}{\sim} \langle \gamma(y), \gamma\rangle$ which completes this case.

($\Leftarrow$) The converse is similar. By definition of $\Downarrow_{\mathsf{ca}}$, $\langle i, \delta\rangle = \langle \gamma(y), \gamma\rangle$, and by definition of $\Downarrow_{\mathsf{cl}}$, $\langle d, \sigma, \mu\rangle = \langle x, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (\mu(\sigma(x)), \mu)$. Moreover $\mu \overset{h}{\to} \gamma$, therefore by definition of $\overset{h}{\to}$, $\mu(\sigma(x)) \overset{h}{\to} \gamma(h(\sigma(x))) = \gamma(y)$. Therefore, with $g = h$, $(\mu(\sigma(x)), \mu) \overset{g}{\sim} \langle \gamma(y), \gamma\rangle = \langle i, \delta\rangle$ which completes this case.

λ-Abstraction If $d = \lambda x.a$, then $e = (h \circ \sigma)(\lambda x.a)$ which is a term α-equivalent to d, so $e = \lambda x.b$ for some b. Indeed, the variable x does not change from d to e since only the free variables of d are affected by $h \circ \sigma$.

($\Rightarrow$) By definition of $\Downarrow_{\mathsf{cl}}$, $(u, \nu) = (\{\lambda x.a, \sigma\}, \mu)$, and by definition of $\Downarrow_{\mathsf{ca}}$, $\langle e, \gamma\rangle = \langle \lambda x.b, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle \lambda x.b, \gamma\rangle$. But $\mathsf{codom}\,\sigma \subseteq \mathsf{dom}\,h$ and $\lambda x.b = (h \circ \sigma)(\lambda x.a)$, therefore $\{\lambda x.a, \sigma\} \stackrel{h}{\to} \lambda x.b$. Moreover we know $\mu \stackrel{h}{\to} \gamma$ and with $g = h$, we get $(\{\lambda x.a, \sigma\}, \mu) \stackrel{g}{\sim} \langle \lambda x.b, \gamma\rangle$ which completes this case.

($\Leftarrow$) The converse is similar. By definition of $\Downarrow_{\mathsf{ca}}$, $\langle i, \delta\rangle = \langle \lambda x.b, \gamma\rangle$, and by definition of $\Downarrow_{\mathsf{cl}}$, $\langle d, \sigma, \mu\rangle = \langle \lambda x.a, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (\{\lambda x.a, \sigma\}, \mu)$. But $\mathsf{codom}\,\sigma \subseteq \mathsf{dom}\,h$ and $\lambda x.b = (h \circ \sigma)(\lambda x.a)$, therefore $\{\lambda x.a, \sigma\} \stackrel{h}{\to} \lambda x.b$. Moreover we know $\mu \stackrel{h}{\to} \gamma$ and with $g = h$, we get $(\{\lambda x.a, \sigma\}, \mu) \stackrel{g}{\sim} \langle \lambda x.b, \gamma\rangle$ which completes this case.

Function application of a λ-abstraction If $d = d_1\ d_2$, then let $e_1 = (h \circ \sigma)(d_1)$ and $e_2 = (h \circ \sigma)(d_2)$. Since $e = (h \circ \sigma)(d)$ means that e is α-equivalent to d, $e = e_1\ e_2$, and we can easily check that $\langle d_1, \sigma, \mu\rangle \stackrel{h}{\sim} \langle e_1, \gamma\rangle$ and $\langle d_2, \sigma, \mu\rangle \stackrel{h}{\sim} \langle e_2, \gamma\rangle$.

($\Rightarrow$) If $\langle d_1\ d_2, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (u, \nu)$ because

$$\langle d_1, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (\{\lambda x.a, \tau\}, \xi) \qquad \langle d_2, \sigma, \xi\rangle \Downarrow_{\mathsf{cl}} (v, \chi) \qquad \langle a, \tau[x/\ell], \chi[\ell/v]\rangle \Downarrow_{\mathsf{cl}} (u, \nu)$$

with ℓ fresh, then by induction hypothesis on the derivation of d_1, there exist k, ζ and h_1 an extension of h such that

$$\langle e_1, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle k, \zeta\rangle \qquad\qquad (\{\lambda x.a, \tau\}, \xi) \stackrel{h_1}{\sim} \langle k, \zeta\rangle$$

The second condition implies that $k = \lambda x.b = (h_1 \circ \tau)(\lambda x.a)$ for some expression b, and that $\xi \stackrel{h_1}{\to} \zeta$. Moreover $d_2 \stackrel{h_1}{\to} e_2$ since $d_2 \stackrel{h}{\to} e_2$, therefore $\langle d_2, \sigma, \xi\rangle \stackrel{h_1}{\sim} \langle e_2, \zeta\rangle$. By induction hypothesis on the derivation of d_2, there exist j, η and h_2 an extension of

h_1 such that

$$\langle e_2,\, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle j,\, \eta\rangle \qquad\qquad (v,\, \chi) \overset{h_2}{\sim} \langle j,\, \eta\rangle$$

As ℓ is the fresh location chosen in the derivation of $\Downarrow_{\mathsf{cl}}$ for d, let y be a fresh variable for the derivation of $\Downarrow_{\mathsf{ca}}$ for e. Let $h_3 : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ such that:

$$h_3 : \mathsf{dom}\, h_2 \cup \{\ell\} \to \mathsf{codom}\, h_2 \cup \{y\}$$

$$\ell_2 \in \mathsf{dom}\, h_2 \mapsto h_2(\ell_2)$$

$$\ell \mapsto y$$

Lemma 3.3.6 *With the variables defined as above,*

$$\langle a,\, \tau[x/\ell],\, \chi[\ell/v]\rangle \overset{h_3}{\sim} (b[x/y],\, \eta[y/j])$$

Proof. First of all, $\lambda x.b = (h_1 \circ \tau)(\lambda x.a)$, h_3 is an extension of h_1 and $\mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\, h_1$, therefore $\lambda x.b = (h_3 \circ \tau)(\lambda x.a)$. Now $b[x/y] = ((h_3 \circ \tau)[x/y])(\lambda x.a) = (h_3 \circ \tau[x/\ell])(\lambda x.a)$ since $h_3(\ell) = y$.

We further need to argue that $\chi[\ell/v] \overset{h_3}{\to} \eta[y/j]$. We already know that $\mathsf{dom}\, h_3 = \mathsf{dom}\, h_2 \cup \{\ell\} = \mathsf{dom}\, \chi \cup \{\ell\} = \mathsf{dom}\, \chi[\ell/v]$, and $h_3(\mathsf{dom}\, \chi[\ell/v]) = \mathsf{codom}\, h_2 \cup \{y\} = \mathsf{dom}\, \eta[y/j]$. Let $\ell_3 \in \mathsf{dom}\, \chi[\ell/v]$. If $\ell_3 \in \mathsf{dom}\, \chi$, then $\chi[\ell/v](\ell_3) = \chi(\ell_3) \overset{h_2}{\to} \eta(h_3(\ell_3)) = \eta[y/j](h_3(\ell_3))$ by injectivity of h_3, therefore $\chi[\ell/v](\ell_3) \overset{h_3}{\to} \eta[y/j](h_3(\ell_3))$. Otherwise, $\ell_3 = \ell$ and then $\chi[\ell/v](\ell) = v \overset{h_2}{\to} j = \eta[y/j](y) = \eta[y/j](h_3(\ell))$, therefore since h_3 is an extension of h_2, $\chi[\ell/v](\ell) \overset{h_3}{\to} \eta[y/j](h_3(\ell))$. This completes the proof of the lemma. □

Using lemma 3.3.14 and by induction hypothesis on the derivation of a, there exist i, δ and g an extension of h_3 such that

$$\langle b[x/y],\, \eta[y/j]\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

Therefore, by definition of $\Downarrow_{\mathsf{cl}}$, $\langle e_1\ e_2,\ \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i,\ \delta\rangle$ and $(u,\ \nu) \overset{g}{\sim} \langle i,\ \delta\rangle$, which completes this case.

($\Leftarrow$) The converse is similar. If $\langle e_1\ e_2,\ \gamma\rangle \Downarrow_{\mathsf{cl}} \langle i,\ \delta\rangle$ because

$$\langle e_1,\ \gamma\rangle \Downarrow_{\mathsf{ca}} \langle \lambda x.b,\ \zeta\rangle \qquad \langle e_2,\ \zeta\rangle \Downarrow_{\mathsf{ca}} \langle j,\ \eta\rangle \qquad \langle b[x/y],\ \eta[y/j]\rangle \Downarrow_{\mathsf{ca}} \langle i,\ \delta\rangle$$

with y fresh, then by induction hypothesis on the derivation of e_1, there exist w, ξ and h_1 an extension of h such that

$$\langle d_1,\ \sigma,\ \mu\rangle \Downarrow_{\mathsf{ca}} (w,\ \xi) \qquad\qquad (w,\ \xi) \overset{h_1}{\sim} \langle \lambda x.b,\ \zeta\rangle$$

The second condition implies that $w = \{\lambda x.a,\ \tau\}$ for some a, τ such that $(h_1 \circ \tau)(\lambda x.a) = \lambda x.b$, and that $\xi \overset{h_1}{\to} \zeta$. Moreover $d_2 \overset{h_1}{\to} e_2$ since $d_2 \overset{h}{\to} e_2$, therefore $\langle d_2,\ \sigma,\ \xi\rangle \overset{h_1}{\sim} \langle e_2,\ \zeta\rangle$. By induction hypothesis on the derivation of e_2, there exist v, χ and h_2 an extension of h_1 such that

$$\langle d_2,\ \sigma,\ \xi\rangle \Downarrow_{\mathsf{ca}} (v,\ \chi) \qquad\qquad (j,\ \eta) \overset{h_2}{\sim} (v,\ \chi)$$

As y is the fresh variable chosen in the derivation of $\Downarrow_{\mathsf{ca}}$ for e, let ℓ be a fresh location for the derivation of $\Downarrow_{\mathsf{cl}}$ for d. Let $h_3 : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ such that:

$$h_3 : \mathsf{dom}\, h_2 \cup \{\ell\} \to \mathsf{codom}\, h_2 \cup \{y\}$$
$$\ell_2 \in \mathsf{dom}\, h_2 \mapsto h_2(\ell_2)$$
$$\ell \mapsto y$$

Lemma 3.3.7 *With the variables defined as above,*

$$\langle a,\ \tau[x/\ell],\ \chi[\ell/v]\rangle \overset{h_3}{\sim} (b[x/y],\ \eta[y/j])$$

Proof. This is the same as lemma 3.3.6, and the same proof holds. □

Using lemma 3.3.7 and by induction hypothesis on the derivation of $b[x/y]$, there exist u, ν and g an extension of h_3 such that

$$\langle a,\, \tau[x/\ell],\, \chi[\ell/v]\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu) \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

Therefore, by definition of $\Downarrow_{\mathsf{cl}}$,

$$\langle d_1\ d_2,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu) \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

which completes this case.

Variable assignment If $d = (x := d_1)$ for some variable x and expression d_1, then $e = (h \circ \sigma)(x := d_1) = (y := e_1)$ with y a variable such that $y = (h \circ \sigma)(x)$ and $e_1 = (h \circ \sigma)(d_1)$. Therefore $\langle d_1,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e_1,\, \gamma\rangle$.

($\Rightarrow$) The derivation of $\Downarrow_{\mathsf{cl}}$ for d shows that $(u,\, \nu) = ((),\, \xi[\sigma(x)/v])$ for some v, ξ such that

$$\langle e_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (v,\, \xi)$$

By induction hypothesis on the derivation of $\Downarrow_{\mathsf{cl}}$ for d_1, there exist j, ζ and g an extension of h such that

$$\langle e_1,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle j,\, \zeta\rangle \qquad\qquad (v,\, \xi) \overset{g}{\sim} \langle j,\, \zeta\rangle$$

Lemma 3.3.8 *With the variables defined as above,*

$$((),\, \xi[\sigma(x)/v]) \overset{g}{\sim} \langle (),\, \zeta[y/j]\rangle$$

Proof. The domain conditions are fulfilled since $(v,\, \xi) \overset{g}{\sim} \langle j,\, \zeta\rangle$, $\mathsf{dom}\,\xi = \mathsf{dom}\,\xi[\sigma(x)/v]$ and $\mathsf{dom}\,\zeta = \mathsf{dom}\,\zeta[y/j]$. Let $\ell \in \mathsf{dom}\,\xi[\sigma(x)/v] = \mathsf{dom}\,\xi$. If $\ell = \sigma(x)$ then

$\xi[\sigma(x)/v](\ell) = v \overset{g}{\sim} j = \zeta[y/j](y) = \zeta[y/j](g(\ell))$ since $g(\ell) = (g \circ \sigma)(x) = (h \circ \sigma)(x) = y$. Otherwise $\xi[\sigma(x)/v](\ell) = \xi(\ell) \overset{g}{\sim} \zeta(h(\ell)) = \zeta[y/j](g(\ell))$ using that h is injective and g is an extension of h. Finally $() \overset{g}{\to} ()$, which completes the proof of the lemma. □

Using lemma 3.3.15 and by definition of $\Downarrow_{\mathsf{ca}}$, $\langle x := e_1,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle (),\, \zeta[y/j]\rangle$ and $\langle u,\, \nu\rangle = ((),\, \xi[\sigma(x)/v]) \overset{g}{\sim} \langle (),\, \zeta[y/j]\rangle$, which completes this case.

($\Leftarrow$) The converse is similar. The derivation of $\Downarrow_{\mathsf{ca}}$ for e shows that $\langle i,\, \delta\rangle = \langle (),\, \zeta[x/j]\rangle$ for some j, ζ such that

$$\langle e_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (v,\, \xi)$$

By induction hypothesis on the derivation of $\Downarrow_{\mathsf{ca}}$ for e_1, there exists v, ξ and g an extension of h such that

$$\langle d_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{ca}} \langle v,\, \xi\rangle \qquad\qquad (v,\, \xi) \overset{g}{\sim} \langle j,\, \zeta\rangle$$

Lemma 3.3.9 *With the variables defined as above,*

$$((),\, \xi[\sigma(x)/v]) \overset{g}{\sim} \langle (),\, \zeta[y/j]\rangle$$

Proof. This is the same as lemma 3.3.8, and the same proof holds. □

Using lemma 3.3.9 and by definition of $\Downarrow_{\mathsf{ca}}$,

$$\langle x := d_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} ((),\, \xi[\sigma(x)/v]) \qquad ((),\, \xi[\sigma(x)/v]) \overset{g}{\sim} \langle (),\, \zeta[y/j]\rangle = \langle i,\, \delta\rangle$$

which completes this case.

The other cases are less interesting but we provide them here for completeness.

Constant If $d = c$ then $e = (h \circ \sigma)(d) = c$ as well.

($\Rightarrow$) The derivation of $\Downarrow_{\mathsf{cl}}$ shows that $(u, \nu) = (c, \mu)$, and the derivation of $\Downarrow_{\mathsf{ca}}$ shows that $\langle e, \gamma\rangle = \langle c, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle c, \gamma\rangle$. Moreover $\mu \xrightarrow{h} \gamma$, therefore with $g = h$, $(c, \mu) \overset{g}{\sim} \langle c, \gamma\rangle$ which completes this case.

($\Leftarrow$) The derivation of $\Downarrow_{\mathsf{ca}}$ shows that $\langle i, \delta\rangle = \langle c, \gamma\rangle$, and the derivation of $\Downarrow_{\mathsf{ca}}$ shows that $\langle d, \sigma, \mu\rangle = \langle c, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (c, \mu)$. Moreover $\mu \xrightarrow{h} \gamma$, therefore with $g = h$, $(c, \mu) \overset{g}{\sim} \langle c, \gamma\rangle$ which completes this case.

Function application of a constant function ($\Rightarrow$) If $\langle d_1\ d_2, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (u, \nu)$ because

$$\langle d_1, \sigma, \mu\rangle \Downarrow_{\mathsf{cl}} (f, \xi) \qquad \langle d_2, \sigma, \xi\rangle \Downarrow_{\mathsf{cl}} (c, \nu) \qquad u = f(c)$$

then, recalling that $\langle d_1, \sigma, \mu\rangle \overset{h}{\sim} (e_1, \gamma)$, by induction hypothesis on the derivation of d_1, there exist j, ζ and h_1 an extension of h such that

$$\langle e_1, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle j, \zeta\rangle \qquad (f, \xi) \overset{h_1}{\sim} \langle j, \zeta\rangle$$

The second condition implies $j = f$ and $\xi \xrightarrow{h_1} \zeta$. Moreover $d_2 \xrightarrow{h_1} e_2$ since $d_2 \xrightarrow{h} e_2$, therefore $\langle d_2, \sigma, \xi\rangle \overset{h_1}{\sim} \langle e_2, \zeta\rangle$. By induction hypothesis on the derivation of d_2, there exist k, δ and g an extension of h_1 such that

$$\langle e_2, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle k, \delta\rangle \qquad (c, \nu) \overset{g}{\sim} \langle k, \delta\rangle$$

The second condition implies $k = c$ and $\nu \xrightarrow{g} \delta$. Therefore, by definition of $\Downarrow_{\mathsf{ca}}$,

$$\langle e_1\ e_2, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle f(c), \delta\rangle \qquad (f(c), \nu) \overset{g}{\sim} \langle f(c), \delta\rangle$$

which completes this case.

($\Leftarrow$) If $\langle e_1\ e_2, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i, \delta\rangle$ because

$$\langle e_1, \gamma\rangle \Downarrow_{\mathsf{cl}} \langle f, \zeta\rangle \qquad \langle e_2, \zeta\rangle \Downarrow_{\mathsf{cl}} \langle c, \delta\rangle \qquad u = f(c)$$

then, recalling that $\langle d_1,\, \sigma,\, \mu\rangle \overset{h}{\sim} (e_1,\, \gamma)$, by induction hypothesis on the derivation of e_1, there exist v, ξ and h_1 an extension of h such that

$$\langle d_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (v,\, \xi) \qquad\qquad (v,\, \xi) \overset{h_1}{\sim} \langle f,\, \zeta\rangle$$

The second condition implies $v = f$ and $\xi \overset{h_1}{\to} \zeta$. Moreover $d_2 \overset{h_1}{\to} e_2$ since $d_2 \overset{h}{\to} e_2$, therefore $\langle d_2,\, \sigma,\, \xi\rangle \overset{h_1}{\sim} \langle e_2,\, \zeta\rangle$. By induction hypothesis on the derivation of e_2, there exist w, ν and g an extension of h_1 such that

$$\langle d_2,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{ca}} (w,\, \nu) \qquad\qquad (w,\, \nu) \overset{g}{\sim} \langle c,\, \delta\rangle$$

The second condition implies $w = c$ and $\nu \overset{g}{\to} \delta$. Therefore, by definition of $\Downarrow_{\mathsf{ca}}$,

$$\langle d_1\ d_2,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{ca}} \langle f(c),\, \delta\rangle \qquad\qquad (f(c),\, \nu) \overset{g}{\sim} \langle f(c),\, \delta\rangle$$

which completes this case.

Composition If $d = (d_1; d_2)$, then $e = (e_1; e_2)$ for $e_1 = (h \circ \sigma)(d_1)$ and $e_2 = (h \circ \sigma)(d_2)$, therefore $\langle d_1,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e_1,\, \gamma\rangle$ and $\langle d_2,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e_2,\, \gamma\rangle$.

($\Leftarrow$) The derivation of $\Downarrow_{\mathsf{cl}}$ for d shows that

$$\langle d_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} ((),\, \xi) \qquad\qquad \langle d_2,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)$$

for some ξ. By induction hypothesis on the derivation of d_1, there exist j, ζ and h_1 an extension of h such that

$$\langle e_1,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle j,\, \zeta\rangle \qquad\qquad ((),\, \xi) \overset{h_1}{\sim} \langle j,\, \zeta\rangle$$

The second condition implies $j = ()$ and $\xi \overset{h_1}{\to} \zeta$. Moreover $d_2 \overset{h_1}{\to} e_2$ since $d_2 \overset{h}{\to} e_2$, therefore $\langle d_2,\, \sigma,\, \xi\rangle \overset{h_1}{\sim} \langle e_2,\, \zeta\rangle$. By induction hypothesis on the derivation of d_2, there exist i, δ and g an extension of h_1 such that

$$\langle e_2,\, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

Therefore, by definition of $\Downarrow_{\mathsf{ca}}$,

$$\langle e_1; e_2,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

which completes this case.

(⇒) The derivation of $\Downarrow_{\mathsf{ca}}$ for e shows that

$$\langle e_1,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle (),\, \zeta\rangle \qquad\qquad \langle e_2,\, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle$$

for some ζ. By induction hypothesis on the derivation of e_1, there exist v, ξ and h_1 an extension of h such that

$$\langle d_1,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (v,\, \xi) \qquad\qquad (v,\, \xi) \overset{h_1}{\sim} \langle j,\, \zeta\rangle$$

The second condition implies $v = ()$ and $\xi \overset{h_1}{\to} \zeta$. Moreover $d_2 \overset{h_1}{\to} e_2$ since $d_2 \overset{h}{\to} e_2$, therefore $\langle d_2,\, \sigma,\, \xi\rangle \overset{h_1}{\sim} \langle e_2,\, \zeta\rangle$. By induction hypothesis on the derivation of e_2, there exist u, ν and g an extension of h_1 such that

$$\langle d_2,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu) \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

Therefore, by definition of $\Downarrow_{\mathsf{cl}}$,

$$\langle d_1; d_2,\, \sigma\rangle \mu \Downarrow_{\mathsf{ca}} (u,\, \nu) \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

which completes this case.

Conditional If $d = (\mathsf{if}\ a\ \mathsf{then}\ d_1\ \mathsf{else}\ d_2)$, then $e = (\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)$ for $b = (h \circ \sigma)(a)$, $e_1 = (h \circ \sigma)(d_1)$ and $e_2 = (h \circ \sigma)(d_2)$, therefore $\langle a,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle b,\, \gamma\rangle$, $\langle d_1,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e_1,\, \gamma\rangle$ and $\langle d_2,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e_2,\, \gamma\rangle$.

(⇐) The derivation of $\Downarrow_{\mathsf{cl}}$ for d shows that either

$$\langle a,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\mathsf{true},\, \xi) \qquad\qquad \langle d_1,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)$$

or

$$\langle a,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\mathsf{false},\, \xi) \qquad\qquad \langle d_2,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu)$$

For some ξ. Let us consider the case where $\langle a,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (\mathsf{true},\, \xi)$; the other case has a very similar proof. By induction hypothesis on the derivation of a, there exist j, ζ and h_1 an extension of h such that

$$\langle b,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle j,\, \zeta\rangle \qquad\qquad (\mathsf{true},\, \xi) \overset{h_1}{\sim} \langle j,\, \zeta\rangle$$

The second condition implies $j = \mathsf{true}$ and $\xi \overset{h_1}{\to} \zeta$. Moreover $d_1 \overset{h_1}{\to} e_1$ since $d_1 \overset{h}{\to} e_1$, therefore $\langle d_1,\, \sigma,\, \xi\rangle \overset{h_1}{\sim} \langle e_1,\, \zeta\rangle$. By induction hypothesis on the derivation of d_1, there exist i, δ and g an extension of h_1 such that

$$\langle e_1,\, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

Therefore, by definition of $\Downarrow_{\mathsf{ca}}$,

$$\langle \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

which completes this case.

($\Rightarrow$) The derivation of $\Downarrow_{\mathsf{ca}}$ for e shows that either

$$\langle b,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle \mathsf{true},\, \zeta\rangle \qquad\qquad \langle e_1,\, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle$$

or

$$\langle b,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle \mathsf{false},\, \zeta\rangle \qquad\qquad \langle e_2,\, \zeta\rangle \Downarrow_{\mathsf{ca}} \langle i,\, \delta\rangle$$

For some ζ. Let us consider the case where $\langle b,\, \gamma\rangle \Downarrow_{\mathsf{ca}} \langle \mathsf{true},\, \zeta\rangle$; the other case has a very similar proof. By induction hypothesis on the derivation of b, there exist v, ξ and h_1 an extension of h such that

$$\langle a,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (v,\, \xi) \qquad\qquad (v,\, \xi) \overset{h_1}{\sim} \langle j,\, \zeta\rangle$$

The second condition implies $v = \mathsf{true}$ and $\xi \xrightarrow{h_1} \zeta$. Moreover $d_1 \xrightarrow{h_1} e_1$ since $d_1 \xrightarrow{h} e_1$, therefore $\langle d_1,\, \sigma,\, \xi\rangle \overset{h_1}{\sim} \langle e_1,\, \zeta\rangle$. By induction hypothesis on the derivation of e_1, there exist u, ν and g an extension of h_1 such that

$$\langle d_1,\, \sigma,\, \xi\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu) \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

Therefore, by definition of $\Downarrow_{\mathsf{cl}}$,

$$\langle \mathsf{if}\ a\ \mathsf{then}\ d_1\ \mathsf{else}\ d_2,\, \sigma,\, \mu\rangle \Downarrow_{\mathsf{cl}} (u,\, \nu) \qquad\qquad (u,\, \nu) \overset{g}{\sim} \langle i,\, \delta\rangle$$

which completes this case.

Loop If $d = (\mathsf{while}\ a\ \mathsf{do}\ d_1)$, then $e = (\mathsf{while}\ b\ \mathsf{do}\ e_1)$ for $b = (h \circ \sigma)(a)$ and $e_1 = (h \circ \sigma)(d_1)$, therefore $\langle a,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle b,\, \gamma\rangle$ and $\langle d_1,\, \sigma,\, \mu\rangle \overset{h}{\sim} \langle e_1,\, \gamma\rangle$. Let $\mu_0 = \mu$, $\gamma_0 = \gamma$ and $h_0 = h$.

($\Rightarrow$) Let $\nu_n = \nu$. The derivation of $\Downarrow_{\mathsf{cl}}$ for d shows that

$$\langle a,\, \sigma,\, \mu_i\rangle \Downarrow_{\mathsf{cl}} (\mathsf{true},\, \nu_i) \qquad\qquad \langle d_1,\, \sigma,\, \nu_i\rangle \Downarrow_{\mathsf{cl}} ((),\, \mu_{i+1}),\ 0 \le i < n$$
$$\langle a,\, \sigma,\, \mu_n\rangle \Downarrow_{\mathsf{cl}} (\mathsf{false},\, \nu_n) \qquad\qquad u = ()$$

for some $n \ge 0, \mu_1, \ldots, \mu_n, \nu_0, \ldots, \nu_{n-1}$. Let us prove by recurrence on $0 \le i < n$ that there exists h_i, γ_i such that $\langle a,\, \sigma,\, \mu_i\rangle \overset{h_i}{\sim} \langle b,\, \gamma_i\rangle$ and $\langle d_1,\, \sigma,\, \mu_i\rangle \overset{h_i}{\sim} \langle e_1,\, \gamma_i\rangle$. The result is already true for $i = 0$, let us suppose it is true for $0 \le i < n$. By induction hypothesis on the derivation $\langle a,\, \sigma,\, \mu_i\rangle \Downarrow_{\mathsf{cl}} (\mathsf{true},\, \nu_i)$, there exist j_i, δ_i and g_i an extension of h_i such that

$$\langle b,\, \gamma_i\rangle \Downarrow_{\mathsf{ca}} \langle j_i,\, \delta_i\rangle \qquad\qquad (\mathsf{true},\, \nu_i) \overset{h_1}{\sim} \langle j_i,\, \delta_i\rangle$$

The second condition implies $j_i = \mathsf{true}$ and $\nu_i \xrightarrow{g_i} \delta_i$. Moreover $d_1 \xrightarrow{g_i} e_1$ since $d_1 \xrightarrow{h_i} e_1$, therefore $\langle d_1,\, \sigma,\, \nu_i\rangle \overset{g_i}{\sim} \langle e_1,\, \delta_i\rangle$. By induction hypothesis on the derivation

$\langle d_1,\, \sigma,\, \nu_i\rangle \Downarrow_{\mathsf{cl}} ((),\, \mu_{i+1})$, there exist k_i, γ_{i+1} and h_{i+1} an extension of g_i such that

$$\langle e_1,\, \delta_i\rangle \Downarrow_{\mathsf{ca}} \langle k_i,\, \gamma_{i+1}\rangle \qquad\qquad ((),\, \mu_{i+1}) \overset{h_{i+1}}{\sim} \langle k_i,\, \gamma_{i+1}\rangle$$

The second condition implies $k_i = ()$ and $\mu_{i+1} \overset{h_{i+1}}{\to} \gamma_{i+1}$. Moreover $a \overset{h_{i+1}}{\to} b$ since $a \overset{h_i}{\to} b$ and $d_1 \overset{h_{i+1}}{\to} e_1$ since $d_1 \overset{g_i}{\to} e_1$, therefore $\langle a,\, \sigma,\, \mu_{i+1}\rangle \overset{h_{i+1}}{\sim} \langle b,\, \gamma_{i+1}\rangle$ and $\langle d_1,\, \sigma,\, \mu_{i+1}\rangle \overset{h_{i+1}}{\sim} \langle e_1,\, \gamma_{i+1}\rangle$. This completes the recurrence. In particular, for $i = n-1$, $\langle a,\, \sigma,\, \mu_n\rangle \overset{h_n}{\sim} \langle b,\, \gamma_n\rangle$. By induction hypothesis on the derivation $\langle a,\, \sigma,\, \mu_n\rangle \Downarrow_{\mathsf{cl}} (\mathsf{false},\, \nu_n)$, there exist j_n, δ_n and g an extension of h_n such that

$$\langle b,\, \gamma_n\rangle \Downarrow_{\mathsf{ca}} \langle j_n,\, \delta_n\rangle \qquad\qquad (\mathsf{false},\, \nu_n) \overset{g}{\sim} \langle j_n,\, \delta_n\rangle$$

The second condition implies $j_n = \mathsf{false}$, therefore by definition of $\Downarrow_{\mathsf{ca}}$,

$$\langle \mathsf{while}\ b\ \mathsf{do}\ e_1,\, \gamma_0\rangle \Downarrow_{\mathsf{ca}} \langle (),\, \delta_n\rangle \qquad\qquad (u,\, \nu) = ((),\, \nu_n) \overset{g}{\sim} \langle (),\, \delta_n\rangle$$

which completes this case.

($\Leftarrow$) Let $\delta_n = \delta$. The derivation of $\Downarrow_{\mathsf{ca}}$ for e shows that

$$\langle b,\, \gamma_i\rangle \Downarrow_{\mathsf{ca}} \langle \mathsf{true},\, \delta_i\rangle \qquad\qquad \langle e_1,\, \delta_i\rangle \Downarrow_{\mathsf{ca}} \langle k_i,\, \gamma_{i+1}\rangle,\ 0 \le i < n$$

$$\langle b,\, \gamma_n\rangle \Downarrow_{\mathsf{ca}} \langle \mathsf{false},\, \delta_n\rangle \qquad\qquad i = ()$$

for some $n \ge 0, \gamma_1, \ldots, \gamma_n, \delta_0, \ldots, \delta_{n-1}$. Let us prove by recurrence on $0 \le i < n$ that there exists h_i, μ_i such that $\langle a,\, \sigma,\, \mu_i\rangle \overset{h_i}{\sim} \langle b,\, \gamma_i\rangle$ and $\langle d_1,\, \sigma\rangle\mu_i \overset{h_i}{\sim} \langle e_1,\, \gamma_i\rangle$. The result is already true for $i = 0$, let us suppose it is true for $0 \le i < n$. By induction hypothesis on the derivation $\langle b,\, \gamma_i\rangle \Downarrow_{\mathsf{ca}} \langle \mathsf{true},\, \delta_i\rangle$, there exist v_i, ν_i and g_i an extension of h_i such that

$$\langle a,\, \sigma,\, \mu_i\rangle \Downarrow_{\mathsf{cl}} (v_i,\, \nu_i) \qquad\qquad (v_i,\, \nu_i) \overset{h_1}{\sim} \langle \mathsf{true},\, \delta_i\rangle$$

The second condition implies $v_i = \mathsf{true}$ and $\nu_i \overset{g_i}{\to} \delta_i$. Moreover $d_1 \overset{g_i}{\to} e_1$ since $d_1 \overset{h_i}{\to} e_1$, therefore $\langle d_1,\, \sigma,\, \nu_i\rangle \overset{g_i}{\sim} \langle e_1,\, \delta_i\rangle$. By induction hypothesis on the derivation

$\langle e_1,\, \delta_i\rangle{\Downarrow_{\mathsf{cl}}}((),\, \gamma_{i+1})$, there exist w_i, μ_{i+1} and h_{i+1} an extension of g_i such that

$$\langle d_1,\, \sigma,\, \nu_i\rangle{\Downarrow_{\mathsf{cl}}}(w_i,\, \mu_{i+1}) \qquad\qquad (w_i,\, \mu_{i+1}) \overset{h_{i+1}}{\sim} \langle (),\, \gamma_{i+1}\rangle$$

The second condition implies $w_i = ()$ and $\mu_{i+1} \overset{h_{i+1}}{\to} \gamma_{i+1}$. Moreover $a \overset{h_{i+1}}{\to} b$ since $a \overset{h_i}{\to} b$ and $d_1 \overset{h_{i+1}}{\to} e_1$ since $d_1 \overset{g_i}{\to} e_1$, therefore $\langle a,\, \sigma,\, \mu_{i+1}\rangle \overset{h_{i+1}}{\sim} \langle b,\, \gamma_{i+1}\rangle$ and $\langle d_1,\, \sigma,\, \mu_{i+1}\rangle \overset{h_{i+1}}{\sim} \langle e_1,\, \gamma_{i+1}\rangle$. This completes the recurrence. In particular, for $i = n-1$, $\langle a,\, \sigma,\, \mu_n\rangle \overset{h_n}{\sim} \langle b,\, \gamma_n\rangle$. By induction hypothesis on the derivation $\langle b,\, \gamma_n\rangle{\Downarrow_{\mathsf{ca}}}(\mathsf{false},\, \nu_n)$, there exist v_n, δ_n and g an extension of h_n such that

$$\langle a,\, \sigma,\, \mu_n\rangle{\Downarrow_{\mathsf{cl}}}(v_n,\, \nu_n) \qquad\qquad (v_n,\, \nu_n) \overset{g}{\sim} \langle \mathsf{false},\, \delta_n\rangle$$

The second condition implies $v_n = \mathsf{false}$, therefore by definition of $\Downarrow_{\mathsf{cl}}$,

$$\langle \mathsf{while}\ a\ \mathsf{do}\ d_1,\, \sigma,\, \mu_0\rangle{\Downarrow_{\mathsf{ca}}}((),\, \nu_n) \qquad\qquad ((),\, \nu_n) \overset{g}{\sim} \langle (),\, \delta_n\rangle = \langle i,\, \delta\rangle$$

which completes this case and the proof. □

3.3.3 Small-step

The proof using small-step semantics is a little bit more complicated. We now give a precise account of the interpretation of variables in state environments, as described in §3.2.3. Given a map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ and a stack of environments Σ, let us inductively define the operator $h \circ \Sigma : \mathsf{StExp} \to \mathsf{Exp}$ as:

$$h \circ (\Sigma :: \sigma)(e) = h \circ \sigma(e)$$

$$h \circ \Sigma(\{\lambda x.a,\, \sigma\}) = \sigma(\lambda x.a)$$

$$h \circ (\Sigma :: \sigma)(s\ \square) = h \circ \Sigma(s)$$

$$h \circ \Sigma(s\ e) = (h \circ \Sigma(s))\ (h \circ \Sigma(e))$$

$$h \circ \Sigma(v\ s) = (h \circ \Sigma(v))\ (h \circ \Sigma(s))$$

$$h \circ (\Sigma :: \sigma)(x := s) = (h \circ \sigma(x)) := h \circ (\Sigma :: \sigma)(s)$$

$$h \circ \Sigma(s; e) = h \circ \Sigma(s); h \circ \Sigma(e)$$

$$h \circ \Sigma(\textsf{if } s \textsf{ then } d \textsf{ else } e) = \textsf{if } h \circ \Sigma(s) \textsf{ then } h \circ \Sigma(d) \textsf{ else } h \circ \Sigma(e)$$

Definition 3.3.10 Given a state $\langle s, \Sigma, \mu\rangle$ and a capsule $\langle e, \gamma\rangle$, both valid, we say that they are *bisimilar under* h, where h is an injective map $h : \textsf{Loc} \rightharpoonup \textsf{Var}$, and we write $\langle s, \Sigma, \mu\rangle \overset{h}{\sim} \langle e, \gamma\rangle$, if and only if

$$h \circ \Sigma(s) = e \qquad\qquad \mu \overset{h}{\to} \gamma$$

□

Now that we know how to relate each element of both semantics, Theorem 3.3.11 shows that any derivation using closures mirrors zero or more derivation steps using capsules, and Theorem 3.3.12 shows that any derivation step using capsules mirrors zero or more derivation steps using closures. Combined, they give rise to Corollary 3.3.13, which shows that any derivation using capsules is mirrored by a derivation using closures, and vice-versa.

Theorem 3.3.11 *If* $\langle s, \Sigma, \mu\rangle \overset{h}{\sim} \langle d, \gamma\rangle$ *and* $\langle s, \Sigma, \mu\rangle \to_{\textsf{cl}} \langle t, \Pi, \nu\rangle$*, then there exists* e, δ *such that*

$$\langle d, \gamma\rangle \overset{*}{\to}_{\textsf{ca}} \langle e, \delta\rangle \qquad\qquad \langle t, \Pi, \nu\rangle \overset{g}{\sim} \langle e, \delta\rangle$$

where g *is an extension of* h*, i.e.,* $\textsf{dom}\, h \subseteq \textsf{dom}\, g$ *and* h *and* g *agree on* $\textsf{dom}\, h$*.*

Theorem 3.3.12 *If* $\langle s, \Sigma, \mu\rangle \overset{h}{\sim} \langle d, \gamma\rangle$ *and* $\langle d, \gamma\rangle \to_{\textsf{ca}} \langle e, \delta\rangle$*, then there exists* t, Π, ν *such that*

$$\langle s, \Sigma, \mu\rangle \overset{*}{\to}_{\textsf{cl}} \langle t, \Pi, \nu\rangle \qquad\qquad \langle t, \Pi, \nu\rangle \overset{g}{\sim} \langle e, \delta\rangle$$

where g *is an extension of* h*, i.e.,* $\textsf{dom}\, h \subseteq \textsf{dom}\, g$ *and* h *and* g *agree on* $\textsf{dom}\, h$*.*

Corollary 3.3.13 *If* $\langle s,\, \Sigma,\, \mu\rangle \overset{h}{\sim} \langle d,\, \gamma\rangle$ *then*

- *if* $\langle s,\, \Sigma,\, \mu\rangle \xrightarrow{*}_{\mathsf{cl}} \langle t,\, \Pi,\, \nu\rangle$ *then there exists* e, δ *such that*

$$\langle d,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e,\, \delta\rangle \qquad\qquad \langle t,\, \Pi,\, \nu\rangle \overset{g}{\sim} \langle e,\, \delta\rangle$$

- *if* $\langle d,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e,\, \delta\rangle$ *then there exists* t, Π, ν *such that*

$$\langle s,\, \Sigma,\, \mu\rangle \xrightarrow{*}_{\mathsf{cl}} \langle t,\, \Pi,\, \nu\rangle \qquad\qquad \langle t,\, \Pi,\, \nu\rangle \overset{g}{\sim} \langle e,\, \delta\rangle$$

where g *is an extension of* h*, i.e.,* $\mathsf{dom}\, h \subseteq \mathsf{dom}\, g$ *and* h *and* g *agree on* $\mathsf{dom}\, h$*.*

Proof of Corollary 3.3.13. We use standard arguments on weak bisimilarity. The first part is proved by recurrence on the number of steps of the derivation of $\langle s,\, \Sigma,\, \mu\rangle \xrightarrow{*}_{\mathsf{cl}} \langle t,\, \Pi,\, \nu\rangle$ and application of Theorem 3.3.11. Similarly, the second part is by recurrence on the number of steps of $\langle d,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e,\, \delta\rangle$ and application of Theorem 3.3.12. $\square$

Proof of Theorem 3.3.11. We proceed by induction on the derivation of $\langle s,\, \Sigma,\, \mu\rangle \rightarrow_{\mathsf{cl}} \langle t,\, \Pi,\, \nu\rangle$. In the interest of space, we only show the most interesting cases of the induction in the main text: variable call x, λ-abstraction $\lambda x.e$, function application of a closure $\{\lambda x.a,\, \sigma\}\ v$, popping from the environment stack $v\ \square$, variable assignment $x := e$, contexts $C[s]$ and computing with the pop indicator $s\ \square$. The other cases, function application of a constant function $f\ c$, composition $d; e$, if conditional $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$ and while loop $\mathsf{while}\ b\ \mathsf{do}\ e$, are straightforward inductive arguments.

Variable If $s = x$ for some variable x and $\Sigma = [\sigma]$ then $d = (h \circ \sigma)(s) = y$ with y the variable such that $y = (h \circ \sigma)(x)$.

By definition of $\rightarrow_{\mathsf{cl}}$, $\langle t, \Pi, \nu\rangle = \langle \mu(\sigma(x)), [\sigma], \mu\rangle$, and by definition of $\rightarrow_{\mathsf{ca}}$, $\langle d, \gamma\rangle = \langle y, \gamma\rangle \rightarrow_{\mathsf{ca}} \langle \gamma(y), \gamma\rangle$. Moreover $\mu \xrightarrow{h} \gamma$, therefore by definition of $\xrightarrow{h}$, $\mu(\sigma(x)) \xrightarrow{h} \gamma(h(\sigma(x))) = \gamma(y)$. Therefore, with $g = h$, $\langle t, \Pi, \nu\rangle = \langle \mu(\sigma(x)), [\sigma], \mu\rangle \overset{g}{\sim} \langle \gamma(y), \gamma\rangle$ which completes this case.

λ-Abstraction If $s = \lambda x.a$ and $\Sigma = [\sigma]$, then $d = (h \circ \sigma)(\lambda x.a)$ which is a term α-equivalent to s, so $d = \lambda x.b$ for some b. Indeed, the variable x does not change from s to d since only the free variables of s are affected by $h \circ \sigma$.

By definition of $\rightarrow_{\mathsf{cl}}$, $\langle t, \Pi, \nu\rangle = \langle \{\lambda x.a, \sigma\}, [\sigma], \mu\rangle$, and by reflexivity of $\xrightarrow{*}_{\mathsf{ca}}$, $\langle d, \gamma\rangle = \langle \lambda x.b, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle \lambda x.b, \gamma\rangle$. But $\mathsf{codom}\,\sigma \subseteq \mathsf{dom}\,h$ and $\lambda x.b = (h \circ \sigma)(\lambda x.a)$, therefore $\{\lambda x.a, \sigma\} \xrightarrow{h} \lambda x.b$. Moreover we know $\mu \xrightarrow{h} \gamma$ and with $g = h$, we get $\langle \{\lambda x.a, \sigma\}, [\sigma], \mu\rangle \overset{g}{\sim} \langle \lambda x.b, \gamma\rangle$ which completes this case.

Function application of a closure If $s = \{\lambda x.a, \sigma\}\ v$ and $\Sigma = [\tau]$, then $(h \circ \Sigma)(\{\lambda x.a, \sigma\}) = (h \circ \sigma)(\lambda x.a)$ is a $\lambda x.b$ for some expression b, and $(h \circ \Sigma)(v)$ is some irreducible term i. Since $d = (h \circ \Sigma)(s)$, $d = (\lambda x.b)\ i$.

By definition of $\rightarrow_{\mathsf{cl}}$, $\langle t, \Pi, \nu\rangle = \langle a\ \square, \sigma[x/\ell] :: \tau, \mu[\ell/v]\rangle$ with ℓ fresh, and by definition of $\rightarrow_{\mathsf{ca}}$, $\langle d, \gamma\rangle \rightarrow_{\mathsf{ca}} \langle b[x/y], \gamma[y/i]\rangle$, with y fresh. Let $g : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ such that:

$$g : \mathsf{dom}\,h \cup \{\ell\} \rightarrow \mathsf{codom}\,g \cup \{y\}$$

$$\ell_h \in \mathsf{dom}\,h \mapsto h(\ell_h)$$

$$\ell \mapsto y$$

Lemma 3.3.14 *With the variables defined as above,*

$$\langle a, [\sigma[x/\ell]], \mu[\ell/v]\rangle \overset{g}{\sim} \langle b[x/y], \gamma[y/i]\rangle$$

Proof. First of all, $\lambda x.b = (h \circ \sigma)(\lambda x.a)$, g is an extension of h and $\mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\, h$, therefore $\lambda x.b = (g \circ \sigma)(\lambda x.a)$. Now $b[x/y] = ((g \circ \sigma)[x/y])(a) = (g \circ \sigma[x/\ell])(\lambda x.a)$ since $g(\ell) = y$.

We further need to argue that $\mu[\ell/v] \xrightarrow{g} \gamma[y/i]$. We already know that $\mathsf{dom}\, g = \mathsf{dom}\, h \cup \{\ell\} = \mathsf{dom}\, \mu \cup \{\ell\} = \mathsf{dom}\, \mu[\ell/v]$, and $g(\mathsf{dom}\, \mu[\ell/v]) = \mathsf{codom}\, h \cup \{y\} = \mathsf{dom}\, \gamma[y/i]$. Let $\ell' \in \mathsf{dom}\, \mu[\ell/v]$. If $\ell' \in \mathsf{dom}\, \mu$, then $\mu[\ell/v](\ell') = \mu(\ell') \xrightarrow{h} \gamma(g(\ell')) = \gamma[y/i](g(\ell'))$ by injectivity of g, therefore $\mu[\ell/v](\ell') \xrightarrow{g} \gamma[y/i](g(\ell'))$. Otherwise, $\ell' = \ell$ and then $\mu[\ell/v](\ell) = v \xrightarrow{h} i = \gamma[y/i](y) = \gamma[y/i](g(\ell))$, therefore since g is an extension of h, $\mu[\ell/v](\ell) \xrightarrow{g} \gamma[y/i](g(\ell))$. This completes the proof of the lemma. $\square$

Using lemma 3.3.14, we get that $(g \circ [\sigma[x/\ell]])(a) = b[x/y]$ and $\mu[\ell/v] \xrightarrow{g} \gamma[y/i]$. But $g \circ (\sigma[x/\ell] :: \tau)(a\ \square) = (g \circ [\sigma[x/\ell]])(a)$, therefore $\langle a\ \square,\ \sigma[x/\ell] :: \tau,\ \mu[\ell/v]\rangle \xrightarrow{g} \gamma[y/i]$, which completes this case.

Popping from the environment stack If $s = v\ \square$ for some value v and $\Sigma = \sigma :: \tau$, then $d = (h \circ \Sigma)(v\ \square) = (h \circ \Sigma)(v)$, which is an irreducible term i such that $v \xrightarrow{h} i$, since:

- if v is a constant c, $i = (h \circ \Sigma)(c) = c$;
- if v is a closure $\{\lambda x.a,\ \sigma'\}$, $i = (h \circ \Sigma)(\{\lambda x.a,\ \sigma'\}) = (h \circ \sigma')(\lambda x.a)$ and $\mathsf{codom}\, \sigma' \subseteq \mathsf{dom}\, h$.

By definition of $\to_{\mathsf{cl}}$, $\langle t,\ \Pi,\ \nu\rangle = \langle v,\ [\tau],\ \mu\rangle$, and by reflexivity of $\xrightarrow{*}_{\mathsf{ca}}$, $\langle d,\ \gamma\rangle = \langle i,\ \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle i,\ \gamma\rangle$. But $i = (h \circ \Sigma(v))$ does not depend on Σ, therefore $i = (h \circ [\tau])(v)$. Moreover we know $\mu \xrightarrow{h} \gamma$ and with $g = h$, we get $\langle v,\ [\tau],\ \mu\rangle \overset{g}{\sim} \langle i,\ \gamma\rangle$ which completes this case.

Variable assignment If $s = (x := v)$ for some variable x and value v and $\Sigma = [\sigma]$, then $d = (h \circ \Sigma)(x := v) = (y := i)$ with y a variable such that $y = (h \circ \sigma)(x)$ and $i = (h \circ \Sigma)(v)$. Therefore $\langle v, \sigma, \mu \rangle \overset{h}{\sim} \langle i, \gamma \rangle$.

By definition of $\rightarrow_{\mathsf{cl}}$, $\langle s, \Pi, \nu \rangle = \langle (), [\sigma], \mu[\sigma(x)/v] \rangle$, and by definition of $\rightarrow_{\mathsf{ca}}$, $\langle d, \gamma \rangle = \langle y := i, \gamma \rangle = \langle (), \gamma[y/i] \rangle$. The following lemma completes this case.

Lemma 3.3.15 *With the variables defined as above,*

$$\langle (), \sigma, \mu[\sigma(x)/v] \rangle \overset{h}{\sim} \langle (), \gamma[y/i] \rangle$$

Proof. The domain conditions are fulfilled since $\langle v, \sigma, \mu \rangle \overset{h}{\sim} \langle i, \gamma \rangle$, $\mathsf{dom}\,\mu = \mathsf{dom}\,\mu[\sigma(x)/v]$ and $\mathsf{dom}\,\gamma = \mathsf{dom}\,\gamma[y/i]$. Let $\ell \in \mathsf{dom}\,\mu[\sigma(x)/v] = \mathsf{dom}\,\mu$. If $\ell = \sigma(x)$ then $\mu[\sigma(x)/v](\ell) = v \overset{h}{\sim} i = \gamma[y/i](y) = \gamma[y/i](h(\ell))$ since $h(\ell) = (h \circ \sigma)(x) = (h \circ \sigma)(x) = y$. Otherwise $\mu[\sigma(x)/v](\ell) = \mu(\ell) \overset{h}{\sim} \gamma(h(\ell)) = \gamma[y/i](h(\ell))$ using that h is injective and h is an extension of h. Finally $() \overset{h}{\rightarrow} ()$, which completes the proof of the lemma. $\square$

Contexts If $s = C[s_1]$ for some context C such that $\langle s_1, \Sigma, \mu \rangle \rightarrow_{\mathsf{cl}} \langle t_1, \Pi, \nu \rangle$, then by definition of $\rightarrow_{\mathsf{cl}}$, $t = C[t_1]$. By definition of $\overset{h}{\sim}$ there exists d_1 such that $d = C[d_1]$. By induction hypothesis there exists e_1, δ such that $\langle d_1, \gamma \rangle \overset{*}{\rightarrow}_{\mathsf{ca}} \langle e_1, \delta \rangle$ and $\langle t_1, \Pi, \nu \rangle \overset{g}{\sim} \langle e_1, \delta \rangle$ for some g extension of h. By definition of $\rightarrow_{\mathsf{ca}}$, $\langle C[d_1], \gamma \rangle \overset{*}{\rightarrow}_{\mathsf{ca}} \langle C[e_1], \delta \rangle$. By induction on the structure of C, and using the fact that the context C cannot contain any $\square$, we can then prove that $\langle C[t_1], \Pi, \nu \rangle \overset{g}{\sim} \langle C[e_1], \delta \rangle$, which completes this case.

Computing under the pop indicator $\square$ If $s = s_1 \, \square$ for s_1 not a value, such that $\langle s_1, \Sigma, \mu \rangle \rightarrow_{\mathsf{cl}} \langle t_1, \Pi, \nu \rangle$, and $\Sigma = \Sigma' :: \sigma$, then by definition of $\rightarrow_{\mathsf{cl}}$, $t = t_1 \, \square$ and $\Pi = \Pi' :: \sigma$ for some Π'. $\langle s_1 \, \square, \Sigma' :: \sigma, \mu \rangle \overset{h}{\sim} \langle d, \gamma \rangle$, therefore $\langle s_1, \Sigma', \mu \rangle \overset{h}{\sim} \langle d, \gamma \rangle$.

By induction hypothesis there exists e, δ such that $\langle d,\, \gamma\rangle \xrightarrow{*}_{\mathsf{ca}} \langle e,\, \delta\rangle$ and $\langle t_1,\, \Pi,\, \nu\rangle \overset{h}{\sim}$ $\langle e,\, \delta\rangle$. Now this proves that $\langle t_1\, \square,\, \Pi' :: \sigma,\, \nu\rangle \overset{h}{\sim} \langle e,\, \delta\rangle$, which completes this case.

□

Proof of Theorem 3.3.12. We proceed similarly as for the proof of Theorem 3.3.11, by induction on the derivation of $\langle d,\, \gamma\rangle \to_{\mathsf{ca}} \langle e,\, \delta\rangle$. We do not detail any case here. The cases for variable call, function application of a λ-term, variable assignment, and contexts are symmetric to the ones seen in the proof of Theorem 3.3.11. The case for function application of a constant function, composition, if conditional and while loop are straightforward inductive arguments. Finally, this Theorem does not need cases for λ-abstractions, popping from the environment stack or computing with the pop indicator, as no rule in $\to_{\mathsf{ca}}$ applies to those. □

3.4 Capsules encode less information

When evaluating an expression using capsules, less information is kept than when evaluating the same expression using closures. Intuitively, when using closures, the state of the computation keeps track of exactly what variables of a λ-abstraction are in scope, even if those variables do not appear in the λ-abstraction itself and will therefore never be used. When using capsules however, the capsule only keeps track of the variables that are both in scope and appear in the λ-abstraction.

For example, let us evaluate the expressions $d = (\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ y = \lambda y.0\ \mathsf{in}\ y)$ and $e = (\mathsf{let}\ y = \lambda y.0\ \mathsf{in}\ \mathsf{let}\ x = 1\ \mathsf{in}\ y)$. Using the definitions of $\Downarrow_{\mathsf{cl}}$ and $\Downarrow_{\mathsf{ca}}$, we can prove that:

$$d \Downarrow_{\mathsf{cl}} (\{\lambda y.0,\, [x = \ell_1]\},\, [\ell_1 = 1, \ell_2 = \{\lambda y.0,\, [x = 1]\}])$$

$$e \Downarrow_{\mathsf{cl}} (\{\lambda y.0,\, [\,]\},\, [\ell_1 = 1, \ell_2 = \{\lambda y.0,\, [\,]\}])$$

$$d \Downarrow_{\mathsf{ca}} \langle \lambda y.0,\ [x' = 1, y' = \lambda y.0] \rangle$$

$$e \Downarrow_{\mathsf{ca}} \langle \lambda y.0,\ [x' = 1, y' = \lambda y.0] \rangle$$

On this example, the result of evaluating d and e with $\Downarrow_{\mathsf{cl}}$ keeps track of whether x is in scope or not, but evaluating d and e with $\Downarrow_{\mathsf{ca}}$ does not. This information is completely superfluous for the rest of the computation and suppressing it with capsules avoids some overhead. Propositions 3.4.1 to 3.4.4 give a more precise account of what is happening.

Proposition 3.4.1 *If $v \xrightarrow{h} i$ then given h, i can be uniquely determined from v; the converse is not true.*

Proof. If $v \xrightarrow{h} i_1$ and $v \xrightarrow{h} i_2$ then either:

- $v \in \mathsf{Const}$ and then $v = i_1$ and $v = i_2$ thus $i_1 = i_2$;
- $v = \{\lambda x.a,\, \tau\} \in \mathsf{Cl}$ and then $i_1 = (h \circ \tau)(\lambda x.a)$ and $i_2 = (h \circ \tau)(\lambda x.a)$ thus $i_1 = i_2$.

However, $\{\lambda y.0,\ [\,]\} \xrightarrow{h} (\lambda y.0)$ and $\{\lambda y.0,\ [x = \ell]\} \xrightarrow{h} (\lambda y.0)$. □

Proposition 3.4.2 *If $\mu \xrightarrow{h} \gamma$ then given h, γ can be uniquely determined from μ; the converse is not true.*

Proof. If $\mu \xrightarrow{h} \gamma_1$ and $\mu \xrightarrow{h} \gamma_2$ then $\mathsf{dom}\,\gamma_1 = h(\mathsf{dom}\,\mu) = \mathsf{dom}\,\gamma_2$. Moreover, for all $\ell \in \mathsf{dom}\, mu$, $\mu(\ell) \xrightarrow{h} \gamma_1(h(\ell))$ and $\mu(\ell) \xrightarrow{h} \gamma_2(h(\ell))$ therefore using proposition 3.4.1, $\gamma_1(h(\ell)) = \gamma_2(h(\ell))$. This covers all the domain of γ_1 and γ_2 since $\mathsf{dom}\,\gamma_1 = \mathsf{dom}\,\gamma_2 = h(\mathsf{dom}\,\mu)$.

However, with h transforming ℓ in z, $[\ell = \{\lambda y.0,\ [\,]\}] \xrightarrow{h} [z = \lambda y.0]$ and $[\ell = \{\lambda y.0,\ [x = \ell]\}] \xrightarrow{h} [z = \lambda y.0]$ □

Proposition 3.4.3 *If* $\langle d, \sigma, \mu\rangle \overset{h}{\sim} \langle e, \gamma\rangle$ *then given* h, $\langle e, \gamma\rangle$ *can be uniquely determined from* $\langle d, \sigma, \mu\rangle$*; the converse is not true.*

Proof. If $\langle d, \sigma, \mu\rangle \overset{h}{\sim} \langle e_1, \gamma_1\rangle$ and $\langle d, \sigma, \mu\rangle \overset{h}{\sim} \langle e_2, \gamma_2\rangle$, then $(h \circ \sigma(d)) = e_1$ and $(h \circ \sigma(d)) = e_2$ therefore $e_1 = e_2$. Moreover $\mu \overset{h}{\rightarrow} \gamma_1$ and $\mu \overset{h}{\rightarrow} \gamma_2$ therefore using proposition 3.4.2, $\gamma_1 = \gamma_2$.

However, with h transforming ℓ in z,

$$\langle x, [x = \ell], [\ell = \{\lambda y.0, [\,]\}]\rangle \overset{h}{\sim} \langle z, [z = \lambda y.0]\rangle$$

$$\langle x, [x = \ell], [\ell = \{\lambda y.0, [x = \ell]\}]\rangle \overset{h}{\sim} \langle z, [z = \lambda y.0]\rangle$$

□

Proposition 3.4.4 *If* $(v, \mu) \overset{h}{\sim} \langle i, \gamma\rangle$ *then given* h, $\langle i, \gamma\rangle$ *can be uniquely determined from* (v, μ)*; the converse is not true.*

Proof. The unicity of $\langle i, \gamma\rangle$ is a direct consequence of propositions 3.4.1 and 3.4.2. However,

$$(\{\lambda y.0, [\,]\}, [\,]) \overset{h}{\sim} \langle \lambda y.0, [\,]\rangle$$

$$(\{\lambda y.0, [x = \ell]\}, [\ell = 1]) \overset{h}{\sim} \langle \lambda y.0, [\,]\rangle$$

□

The idea behind those propositions is that for every capsule, there are several bisimilar states corresponding to different computations, and each keeping track of a different set of superfluous information. Similarly, for every irreducible capsules, there are several bisimilar results keeping track of superfluous information. Capsules thus offer a much cleaner representation of the state of computation.

3.5 Discussion

3.5.1 Capsules and Closures: a strong correspondence

Theorem 2.4.1 and corollary 3.3.13 show that capsules and closures are very strongly related. Not only is there a derivation based on capsules for every derivation based on closures, but these two derivations mirror each other. In big steps, this is because each rule of the definition of $\Downarrow_{\mathsf{ca}}$ mirrors a rule of the definition of $\Downarrow_{\mathsf{cl}}$, and because the proof of the theorem is a direct structural induction on the definitions of $\Downarrow_{\mathsf{cl}}$ and $\Downarrow_{\mathsf{ca}}$. In small steps, rules for capsules and closures do not mirror each other as perfectly, but the same idea holds. Thus the computations are completely bisimilar, even though definining computations for capsules is simpler.

3.5.2 Suppression of the environment σ or the stack Σ

When using closures, a state is a triple $\langle d, \sigma, \mu\rangle$ in big-step (resp. a triple $\langle s, \Sigma, \mu\rangle$ in small-step), whereas when using capsules, it is just a capsule $\langle e, \gamma\rangle$ in both cases. It they are bisimilar under h, it means that $(h \circ \sigma)(d) = e$ (resp. $(h \circ \Sigma)(d) = e$) and $\mu \xrightarrow{h} \gamma$. Capsules eliminate the need for the environment σ (resp. the stack of environments Σ) and thus suppress the indirection in closures that was needed to handle imperative features. Their small-step semantics also does not need any stack of environments of any sort, making the state of computation much simpler. Finally, we originally created the capsule environment γ to replace the (closure) environment σ (resp. the stack of environments Σ). However, it is remarkable that γ is much closer to the store μ, while at the same time eliminating the need for σ (resp. Σ).

Chapter 4

Capsules and Separation

In this chapter, we study a formulation of separation logic using capsules, a representation of the state of a computation in higher-order programming languages with mutable variables. We prove soundness of the frame rule in this context and investigate alternative formulations with weaker side conditions.

4.1 Introduction

Separation logic is a logic for the study of locality and shared data. Introduced by Reynolds in a series of lectures in the late 1990s, based on an earlier idea of Burstall, separation logic has been widely studied in the last decade [Rey00, IO01, ORY01, Rey02, BBTS07, PB08]. The difficulties of reasoning in the presence of heaps, stores, stacks, and pointers are no more apparent than in this literature. Several papers [YO02, BCY05, PBC06, COY07] cite notoriously thorny issues from dangling pointers to arcane side conditions needed for soundness. Reynolds himself acknowledged that his original formulation of separation logic was flawed [Rey02]. Chief among the difficulties is the issue of catastrophic failure due to the dereferencing of unbound

variables or dangling pointers. There seems to be an unspoken belief that this is an unavoidable aspect that must be confronted in any realistic model of computation.

On the contrary, we believe that the essential structure of separation is independent of these encumbrances. It is our thesis that freedom from catastrophic failure is the responsibility of the programming language, not the logic. Capsule semantics provides this assurance, even in the presence of higher-order constructs and mutable variables. This is because all free variables appearing in a capsule must be bound in the environment of the capsule, by definition.

In this chapter, we propose a semantics for separation logic based on capsules. The formulation works for higher-order programs with mutable variables. In §4.4 we give the semantics and prove the soundness of the frame rule in this context. In §4.4.4, we study the motivation behind the nonstandard definition of partial correctness preferred in much of the literature of separation logic [Rey02, COY07] and investigate alternatives. It is here that the advantages of capsule semantics in the study of separation can best be seen.

4.2 Assertions

Assertions $P, Q, \ldots$ are statements in some logical system, possibly with free variables in Var. We write $\mathsf{FV}(P)$ for the set of free variables of P. These variables are subject to interpretation provided by a capsule environment σ.

The exact nature of the underlying logic is unimportant—it could be propositional, first order, second order or higher order—but we do require a few basic properties common to standard logical systems. There must be a well-defined satisfaction relation $\models$ such that $\sigma \models P$ has a truth value when the free variables of P are interpreted by the capsule environment σ. Use of the metaexpression $\sigma \models P$

carries the tacit assumption that $\mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma$. We will augment the logic with the separation logic operators $*$ and $-\!\!*$ (defined later in §4.4). Finally, we require the following (natural) property to hold: if $\sigma \models P$, and $z \in \mathsf{dom}\,\sigma - \mathsf{FV}(P)$, then z can be renamed via α-conversion of the second kind without affecting the truth of P. In examples, we will use first order logic with $*$ and $-\!\!*$, and equality on base types.

4.3 Partial Correctness

The traditional definition of partial correctness and the definition used in the literature on separation logic diverge in a subtle and interesting way. The difference hinges on whether the precondition is required to assert the absence of catastrophic failure due to dangling pointers or lookup of unbound variables; this is the **abort** condition of Reynolds [Rey02] or the **fault** condition of Calcagno, O'Hearn, and Yang [COY07]. Our view, however, is that avoidance of catastrophic failure is the responsibility of the programming language semantics, not the program logic, and capsules do just that. Can this condition then be eliminated? In this section we shed some light on this question.

Let P, Q be assertions and e a program. At issue is the meaning of the partial correctness assertion $\{P\}\,e\,\{Q\}$. Consider the following three metastatements, each parameterized by a closed environment σ:

(A_σ) $\sigma \models P$

(B_σ) $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma$

(C_σ) if $\langle e,\, \sigma\rangle \xrightarrow{*} \langle v,\, \tau\rangle$ and v is irreducible, then $\tau \models Q$.

Statement (A_σ) entails $\mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma$, because the definition of $\models$ does not make sense without it. More strongly, $\mathsf{cl}_\sigma(\mathsf{FV}(P)) \subseteq \mathsf{dom}\,\sigma$, since σ is closed. Statement (B_σ) is equivalent to the assertion that $\langle e,\, \sigma\rangle$ is a valid capsule. Reynolds's definition [Rey02] uses a slightly different formulation

(B'_σ) $\neg(\langle e,\, \sigma\rangle \xrightarrow{*} \mathbf{abort})$

in place of (B_σ). Here $\langle e,\, \sigma\rangle$ need not be a valid capsule. The semantics of capsule evaluation already precludes **abort**, thus (B'_σ) is always true if $\langle e,\, \sigma\rangle$ is a capsule; that is, (B_σ) implies (B'_σ).

Now consider the following potential interpretations of $\{P\}\, e\, \{Q\}$.

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; \forall\sigma\ (\mathrm{A}_\sigma) \wedge (\mathrm{B}_\sigma) \Rightarrow (\mathrm{C}_\sigma) \tag{4.1}$$

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; \forall\sigma\ (\mathrm{A}_\sigma) \Rightarrow (\mathrm{B}_\sigma) \wedge (\mathrm{C}_\sigma) \tag{4.2}$$

Definition (4.1) says that if the precondition P holds of the input state σ and the evaluation of $\langle e,\, \sigma\rangle$ terminates normally, then the output state τ satisfies the postcondition Q. This is the naive interpretation used in traditional forms of Hoare logic. Alternatively, the version preferred in the literature on separation logic would be (4.2), the difference being that the precondition P must ensure that the evaluation of $\langle e,\, \sigma\rangle$ cannot terminate abnormally.

Reynolds's version [Rey02] is actually slightly weaker, using (B'_σ) instead of (B_σ):

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; \forall\sigma\ (\mathrm{A}_\sigma) \Rightarrow (\mathrm{B}'_\sigma) \wedge (\mathrm{C}_\sigma) \tag{4.3}$$

However, the difference is inconsequential: if $\{P\}\, e\, \{Q\}$ holds in the sense of (4.3) but not (4.2), then there exists a variable $x \in \mathsf{FV}(e) - \mathsf{dom}\,\sigma$ for some σ satisfying P, and consequently $x \in \mathsf{FV}(e) - \mathsf{cl}_\sigma(\mathsf{FV}(P))$; but by (B'_σ), x can never be referenced or assigned in the evaluation of $\langle e,\, \sigma\rangle$. Thus the presence or absence of x in the domain of σ affects neither the truth of P nor the evaluation of $\langle e,\, \sigma\rangle$.

But there is a much more important benefit to (4.2) over (4.3). Consider the metastatement

(B) $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$.

A consequence of (4.2) is that (A_σ) implies (B_σ) for all σ. If P is satisfiable at all, say by some σ, then (B) must hold, since variables in $\mathsf{dom}\,\sigma$ not occurring free in P can be renamed (by an α-conversion of the second kind—see §2.5.3) without affecting the truth of P. Thus (4.2) holds with (B) in place of (B_σ). Moreover, since (B) is independent of σ, assuming P is satisfiable at all, (4.2) is equivalent to the definition

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; (\mathrm{B}) \wedge (\forall\sigma\ (\mathrm{A}_\sigma) \Rightarrow (\mathrm{C}_\sigma)) \tag{4.4}$$

Note that, unlike (B_σ) and (B'_σ), the condition (B) is syntactically checkable, thus suitable as a side condition in a rule of inference. If we like, we may remove the condition (B) in the definition of $\{P\}\, e\, \{Q\}$ and instead introduce it as a side condition in the frame rule. However, can it be eliminated entirely? That is, is the formulation (4.1) sound? We show in §4.4.4 that it is not. In fact, even only slightly weaker forms of the side condition (B) do not suffice for soundness.

4.4 Capsules and Separation Logic

4.4.1 Definitions

Here is our semantics for separation logic in terms of capsules. Call closed environments σ and τ *independent* and write $\sigma \perp \tau$ if their domains are disjoint. Define

$\sigma + \tau$ to be the join of σ and τ, provided they are independent. That is,

$$(\sigma+\tau)(x) = \begin{cases} \sigma(x), & \text{if } x \in \mathsf{dom}\,\sigma, \\ \tau(x), & \text{if } x \in \mathsf{dom}\,\tau, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Define *separating conjunction* by

$$\sigma \models P * Q$$

if there exist σ_1 and σ_2 such that $\sigma = \sigma_1 + \sigma_2$, $\sigma_1 \models P$, and $\sigma_2 \models Q$. Define *separating implication* by

$$\sigma \models P \mathrel{-\!\!*} Q$$

if $\sigma + \tau \models Q$ whenever $\tau \models P$ and $\sigma + \tau$ exists. It is easily seen that capsule environments form a *separation algebra* in the sense of [COY07] under these definitions. That is, the structure

$$(\{\text{capsule environments}\}, +, \varnothing)$$

is a cancellative partial commutative monoid. This means that $+$ is a commutative and associative partial binary operation with identity $\varnothing$ satisfying the *cancellative property*: the partial function $+$ is injective in each variable. The relation $\sigma \perp \tau$ holds if and only if $\sigma + \tau$ is defined.

It follows from results of [COY07] that separating conjunction $*$ and separating implication $\mathrel{-\!\!*}$ satisfy the usual intuitionistic relationship: For all closed σ such that $\mathsf{FV}(P) \cup \mathsf{FV}(Q) \cup \mathsf{FV}(R) \subseteq \mathsf{dom}\,\sigma$,

$$\sigma \models (P * Q) \mathrel{-\!\!*} R \;\Leftrightarrow\; \sigma \models P \mathrel{-\!\!*} (Q \mathrel{-\!\!*} R).$$

Other axioms of separation logic mentioned in [Rey02] are also easily checked:

$$(P \vee Q) * R \Leftrightarrow (P * R) \vee (Q * R)$$

$$(P \wedge Q) * R \Rightarrow (P * R) \wedge (Q * R)$$

$$(\exists x\ P) * Q \Leftrightarrow \exists x\ (P * Q) \quad (x \notin \mathsf{FV}(Q))$$

$$(\forall x\ P) * Q \Rightarrow \forall x\ (P * Q) \quad (x \notin \mathsf{FV}(Q)).$$

4.4.2 The Frame Rule

The soundness of the frame rule was first proved in [YO02] for the heap model of computation. Our proof is essentially the same as the one given in [Rey02], but somewhat shorter due to the simplifications afforded by capsule semantics.

Lemma 4.4.1 *If*

$$\langle e,\, \sigma_1 + \sigma_2 \rangle \xrightarrow{*} \langle e,\, \tau \rangle$$

and $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma_1$ *(that is,* $\langle e,\, \sigma_1 \rangle$ *is a capsule), then for some* τ_1*,* $\langle e,\, \sigma_1 \rangle \xrightarrow{*} \langle e,\, \tau_1 \rangle$ *and* $\tau = \tau_1 + \sigma_2$*.*

Proof. By induction on the derivation. None of the small-step evaluation rules listed in §3.2 access any variable outside the domain of σ_1 except for fresh variables introduced in the application rule. In particular, the environment σ_2 is not touched during the evaluation. □

Theorem 4.4.2 *Under capsule semantics, the frame rule*

$$\frac{\{P\}\, e\, \{Q\}}{\{P * R\}\, e\, \{Q * R\}}$$

is sound with respect to definition (4.2) *or* (4.4) *of partial correctness assertions. Equivalently, the frame rule is sound with respect to definition* (4.1) *of partial correctness assertions in the presence of the side condition* $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$*.*

Proof. As argued in §4.3, in all cases we can assume $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$. Suppose $\{P\}\ e\ \{Q\}$. Let $\sigma \models P * R$. Then $\sigma = \sigma_1 + \sigma_2$ with $\sigma_1 \models P$ and $\sigma_2 \models R$. Then $\mathsf{FV}(R) \subseteq \mathsf{dom}\,\sigma_2$ and $\mathsf{FV}(e) \subseteq \mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma_1$, therefore $\langle e,\, \sigma_1\rangle$ is a valid capsule. Since $\langle e,\, \sigma\rangle \xrightarrow{*} \langle v,\, \tau\rangle$, by Lemma 4.4.1 there exists τ_1 such that $\langle e,\, \sigma_1\rangle \xrightarrow{*} \langle v,\, \tau_1\rangle$ and $\tau = \tau_1 + \sigma_2$, and $\tau_1 \models Q$ by the premise of the rule. Thus $\tau \models Q * R$. □

4.4.3 Discussion

Calcagno, O'Hearn, and Yang [COY07] argue that the soundness of the frame rule for a given evaluation semantics is equivalent to the following two properties.

Safety Monotonicity If $\langle e,\, \sigma_0\rangle$ is safe and $\sigma = \sigma_0 + \sigma_1$, then $\langle e,\, \sigma\rangle$ is safe.

Frame Property If $\langle e,\, \sigma_0\rangle$ is safe, $\sigma = \sigma_0 + \sigma_1$, and $\langle e,\, \sigma\rangle \xrightarrow{*} \langle e,\, \sigma'\rangle$, then there exists σ_0' such that $\sigma' = \sigma_0' + \sigma_1$ and $\langle e,\, \sigma_0\rangle \xrightarrow{*} \langle e,\, \sigma_0'\rangle$.

(Here we are allowing $\langle e,\, \sigma\rangle$ to violate the closure conditions in the definition of capsules, and *safe* means that (B'_σ) holds.) In their words, "The first condition says that if a state has enough resources for safe execution of a command, then so do superstates. The second condition says that if a state has enough resources for the command to execute safely, then execution on any bigger state can be tracked back to the small state."

With capsules, the safety monotonicity property is vacuously true, and the frame property reduces to Lemma 4.4.1.

4.4.4 Alternative Conditions

Recall from §4.3 the side condition

(B) $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$,

for which the frame rule with semantics (4.1) for partial correctness assertions is sound. One might ask whether there is a weaker side condition that suffices for soundness. In this section we show that there is not much hope.

The frame rule as stated by Reynolds has a side condition, which says that "no variable occurring free in R is modified by e" [Rey02]. A literal formulation of the side condition in terms of capsules is

(C) $\mathsf{AV}(e) \cap \mathsf{FV}(R) = \varnothing$,

where $\mathsf{AV}(e)$, the *assigned variables* of e, is the set of $x \in \mathsf{FV}(e)$ having a free occurrence on the left-hand side of an assignment :=. This is a bit confusing, because (C) seems to serve no purpose:

Theorem 4.4.3

(a) *Under semantics* (4.2) *of partial correctness assertions, the side condition (C) in the frame rule is redundant.*

(b) *Under semantics* (4.1) *of partial correctness assertions, the frame rule with side condition (C) is not sound.*

Proof. First (a). As argued in §4.3, semantics (4.2) is equivalent to semantics (4.1) with side condition (B), provided P is satisfiable at all. We show that in all such nontrivial instances, (C) is subsumed by (B).

Suppose $\sigma \models P * R$. Then

$$\sigma = \sigma_1 + \sigma_2 \qquad \sigma_1 \models P \qquad \sigma_2 \models R.$$

By (B), we have

$$\mathsf{AV}(e) \subseteq \mathsf{FV}(e) \subseteq \mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma_1$$

and also

$$\mathsf{FV}(R) \subseteq \mathsf{dom}\,\sigma_2 \qquad\qquad \mathsf{dom}\,\sigma_1 \cap \mathsf{dom}\,\sigma_2 = \varnothing,$$

therefore (C) holds.

For (b), we give a counterexample to soundness. Let σ consist of the bindings

$$\sigma(f) = \lambda().x \qquad\qquad \sigma(x) = 2.$$

Let $R = R(f)$ be the assertion $f() = 2$. Let e be the program $x := 3$. Let $P = Q = \mathsf{true}$. The corresponding instance of the frame rule is

$$\frac{\{\mathsf{true}\}\; x := 3\; \{\mathsf{true}\}}{\{\mathsf{true} * f() = 2\}\; x := 3\; \{\mathsf{true} * f() = 2\}}$$

The premise $\{\mathsf{true}\}\; x := 3\; \{\mathsf{true}\}$ holds, but the conclusion does not. We have

$$\sigma \models \mathsf{true} * f() = 2,$$

where $\sigma = \varnothing + \sigma$, $\varnothing \models \mathsf{true}$, and $\sigma \models f() = 2$ and $\varnothing$ is the empty environment. The program e does not assign to f, the only variable free in R, yet it indirectly alters the value of f by assigning a new value to x, making R false. $\square$

We remark that Theorem 4.4.3(b) holds not just for capsules, but for any programming language with records, arrays, objects, pointers, or any form of aliasing whatsoever.

The problem at first seems to be that it is not enough to say that no variable in $\mathsf{FV}(R)$ may be modified by e; we must ensure that no variable in the closure of $\mathsf{FV}(R)$ may be modified by e, so that e cannot even indirectly alter R. This is the condition

(B_1) $\forall\sigma\ \ \sigma \models P * R \Rightarrow \mathsf{AV}(e) \cap \mathsf{cl}_\sigma(\mathsf{FV}(R)) = \varnothing$

which is not expressible by any syntactic property of e, P, Q, and R.

But even this is not enough for soundness. Condition (B_1) is implied by the very strong syntactic property

(B_2) $\mathsf{AV}(e) \subseteq \mathsf{FV}(P)$

which is only slightly weaker than (B). It asserts that all free variables assigned by e are mentioned by P. Nevertheless, even (B_2) is not enough for soundness. At first this may seem quite counterintuitive, because (B_2) implies that starting in any state satisfying $P * R$, e cannot change any variable in the closure of $\mathsf{FV}(R)$, therefore cannot affect the truth of R. We state it as a theorem.

Theorem 4.4.4 *The frame rule under semantics* (4.1) *for partial correctness assertions with side condition* (B_2) *is not sound.*

Proof. Let σ consist of the bindings

$$\sigma(g) = \lambda x.2 \qquad\qquad \sigma(f) = \lambda().3.$$

Let $R = R(f)$ be the assertion $f() = 3$. Let e be the program

$$g := \lambda x.\mathsf{if}\ x = 1\ \mathsf{then}\ f()\ \mathsf{else}\ 2.$$

Let P and Q both be the assertion $g(0) = 2$. The corresponding instance of the frame rule is

$$\frac{\{g(0) = 2\}\ e\ \{g(0) = 2\}}{\{g(0) = 2 * f() = 3\}\ e\ \{g(0) = 2 * f() = 3\}}$$

The premise $\{g(0) = 2\}\ e\ \{g(0) = 2\}$ holds, as does the side condition (B_2), since

$$\mathsf{AV}(e) = \{g\} = \mathsf{FV}(P).$$

However, the conclusion does not. We have

$$\sigma \models g(0) = 2 * f() = 3,$$

where $\sigma = \sigma_1 + \sigma_2$, $\mathsf{dom}\,\sigma_1 = \{g\}$, $\mathsf{dom}\,\sigma_2 = \{f\}$, $\sigma_1 \models g(0) = 2$, and $\sigma_2 \models f() = 3$. However, after execution of the program e, the resulting environment binds g to a term containing a free occurrence of f, so g and f cannot be separated. □

4.5 Conclusion and Future Work

We were motivated to undertake this study in response to an anonymous review of chapter 2 claiming that capsules "contradict the insights of separation logic which has been extensively researched for the last decade." We hope that we have convinced the reader that there is no contradiction whatsoever—in fact quite the opposite! Capsules provide a novel perspective on separation logic, because they capture the same locality and persistence structure as traditional heap models, but in a simpler, more mathematically tractable framework. We feel that this has great potential for enhancing the understanding of separation by allowing research to focus on the essentials.

We have only begun to scratch the surface in this work. We would like to investigate other structures that have arisen in the study of separation logic to see whether capsules can contribute there as well. The preliminary results of this chapter leave us optimistic.

In particular, higher-order separation logic [BBTS07] proposes to use the much more powerful higher-order logic in predicates. Nested Hoare triples [SBRY09] are a neat idea to specify code stored in the heap. The anti-frame rule [Pot08, SYB+10] presents a very interesting way of modeling hidden state. Finally, we would like to

study the concurrency rule [O'H07] in the context of capsules.

Part II

Non-Well-Founded Computation

Chapter 5

Well-Founded Coalgebras, Revisited

Theoretical models of recursion schemes have been well studied under the names well-founded coalgebras, recursive coalgebras, corecursive algebras, and Elgot algebras. Much of this work focuses on conditions ensuring unique or canonical solutions, e.g. when the coalgebra is well-founded.

If the coalgebra is not well-founded, then there can be multiple solutions. The standard semantics of recursive programs gives a particular solution, namely the least solution in a flat Scott domain, which may not be the desired one. In chapters 6 and 7, we propose programming language constructs to allow the specification of alternative solutions and methods to compute them, and we implemente these new constructs as an extension of OCaml.

In this chapter, we prove some theoretical results characterizing well-founded coalgebras that slightly extend results of Adámek, Lücke, and Milius (2007), along with several examples for which this extension is useful. We also give several examples that are not well-founded but still have a desired solution. In each case, the

function would diverge under the standard semantics of recursion, but can be specified and computed with the programming language constructs we have proposed.

5.1 Introduction

Infinite coinductive datatypes and functions on them offer interesting challenges in the design of programming languages. While most programmers feel comfortable with inductive datatypes, coinductive datatypes are often considered difficult to handle. Many programming languages do not even provide constructs to define them. OCaml offers the possibility of defining coinductive datatypes, but the means to define recursive functions on them are limited. Often the obvious definitions do not halt or provide the wrong solution.

Theoretical models of recursion schemes have been well studied under the names well-founded coalgebras, recursive coalgebras [ALM07], corecursive algebras [CUV09], and Elgot algebras [AMV06]. Much of this work focuses on conditions ensuring unique or canonical solutions, e.g. when the coalgebra is well-founded.

A prototypical example of a function that fits the well-founded scheme is mergesort. Given a list, we can sort it by dividing it into identical pieces, sorting the smaller lists, then merging the resulting sorted lists. The base case is the empty list or the list containing a single element. As with most recursive functions, the scheme of definition is: given an argument, check if it is the base case; if not, prepare the arguments for the recursive calls, recursively apply the function, then combine the results of the recursive calls into the final result. For mergesort, this scheme is illustrated in the following diagram:

$$\begin{array}{ccc} A^* & \xrightarrow{\mathsf{mergesort}} & A^* \\ \gamma \downarrow & & \uparrow \alpha \\ A^* + A^* \times A^* & \xrightarrow[\mathsf{id}_{A^*} + \mathsf{mergesort} \times \mathsf{mergesort}]{} & A^* + A^* \times A^* \end{array}$$

The function γ checks whether the list is empty or a singleton, otherwise divides it in two lists of roughly equal size.

$$\gamma(\ell) = \iota_1(\ell), \quad \ell = [\,] \text{ or } \ell = [a]$$

$$\gamma([a_1; \ldots; a_n]) = \iota_2([a_1; \ldots; a_{\lfloor n/2 \rfloor}], [a_{\lfloor n/2 \rfloor + 1}; \ldots; a_n]), \quad n \geq 2.$$

Here ι_1 and ι_2 are the injections into the coproduct. After the function is applied recursively, the results of the recursive calls are combined by α, which merges the two sorted lists.

$$\alpha(\iota_1(\ell)) = \ell \qquad\qquad \alpha(\iota_2(\ell_1, \ell_2)) = \mathsf{merge}(\ell_1, \ell_2)$$

The merge function obeys a similar scheme:

$$\begin{array}{ccc} A^* \times A^* & \xrightarrow{\mathsf{merge}} & A^* \\ \gamma \downarrow & & \uparrow \alpha \\ A^* + A \times A^* \times A^* & \xrightarrow[\mathsf{id}_{A^*} + \mathsf{id}_A \times \mathsf{merge}]{} & A^* + A \times A^* \end{array}$$

where

$$\gamma([\,], \ell) = \gamma(\ell, [\,]) = \iota_1(\ell) \qquad\qquad \alpha(\iota_1(\ell)) = \ell$$

$$\gamma(a_1 :: \ell_1, a_2 :: \ell_2) = \begin{cases} \iota_2(a_1, \ell_1, a_2 :: \ell_2) & \text{if } a_1 \leq a_2 \\ \iota_2(a_2, a_1 :: \ell_1, \ell_2) & \text{if } a_1 > a_2 \end{cases} \qquad \alpha(\iota_2(a, \ell)) = a :: \ell.$$

The fact that these functions are well-defined and unique follows from the theory of recursive coalgebras [ALM07].

Abstractly, these definitional schemes are of the form

$$\begin{array}{ccc} C & \xrightarrow{h} & A \\ \gamma \downarrow & & \uparrow \alpha \\ FC & \xrightarrow[Fh]{} & FA \end{array} \tag{5.1}$$

where F is usually a polynomial functor on Set and (C, γ) and (A, α) are a coalgebra and an algebra, respectively, for the functor F. The function h being defined is called an *F-coalgebra-algebra morphism*. Diagram (5.1) plays a key role in this chapter, as well as in chapters 6 and 7.

The standard semantics of recursion, as provided by all modern programming languages, provides a means of expressing and computing the unique solution of (5.1), provided the coalgebra C is well-founded; that is, provided there is a basis to the recursion. However, the diagram (5.1) can act as a valid definitional scheme even when C is not well-founded. This observation was the starting point of our work on new program constructs for functions defined by such definitional schemes when C is not well-founded, described in chapter 6 and 7.

In the course of our study, we also proved some theoretical results that clarify and slightly generalize some results of [ALM07]. In this chapter, we present those results, and provide some examples where our extension is useful. Although the results of [ALM07] apply to a large class of recursive function definitions, there appear to be cases that are not covered, at least not in any straightforward way. The simplest example is the case of mutually recursive definitions. For example, consider the even and odd predicates on natural numbers. In an ML-style language, we would write:

```
let rec even n = if n = 0 then true else odd (n-1)
and odd n = if n = 0 then false else even (n-1)
```

Our results extend the results of [ALM07] to several patterns of function definitions, including this one. Mutually recursive functions are treated in [AMV06], but our treatment is more symmetric.

The main result of this chapter is a theoretical result that clarifies and mildly generalizes a result of [ALM07]. We show:

- Every F-coalgebra C contains a maximal well-founded subcoalgebra $\mathsf{wf}\, C$.
- If R is a final F-coalgebra, then $\mathsf{wf}\, R$ is the initial F-algebra.
- Let C be an F-coalgebra. The following are equivalent:
  - C is well-founded; that is, $C = \mathsf{wf}\, C$.
  - There is a valid induction principle for C (defined precisely in §5.3.2).
  - There is a unique coalgebra morphism $C \to \mathsf{wf}\, R$.
  - There is a unique coalgebra-algebra morphism from C to any F-algebra.

Our constructions are based on the concept of *realizations*, a concrete representation of final coalgebras for a wide class of multisorted type signatures [Koz11]. Realizations go beyond ordinary polynomial functors on Set in that they handle infinite (countable or uncountable) product and sum as well as total and partial functions. They also handle multi-sorted signatures in a more symmetric way, without relying on any Cartesian structure or parameterization as in [AMV06].

Our second contribution is a variety of well-founded and non-well-founded examples that illustrate the power and limitations of the theory.

The chapter is organized as follows. In §5.2 we review the results of [Koz11] on realizations of coinductive types, which are essential to the understanding of our main theoretical results in §5.3. In §5.3 we give a new characterization of well-founded coalgebras in terms of realizations. In §5.4 we present several examples of well-founded applications. Some of these are already covered by the results of [ALM07], but others, such as mutually recursive functions even/odd and the Ackermann function, are not. However, each of these exhibits some interesting or surprising characteristic that attests to the wide applicability of the theory. In §5.5 we present several non-well-founded examples, including an example of Capretta [Cap07] involving descending sequences of natural numbers and the semantics of alternating Turing machines and IND programs [HK84]. These examples illustrate the usefulness of (5.1) as a definitional scheme even in the non-well-founded case. We conclude in §5.6 with a discussion of related theoretical and practical results.

5.2 Realization of Coinductive Types

In the proof of Theorem 5.3.3, we make use of an explicit construction of final coalgebras from [Koz11]. To make this chapter self-contained, this section recalls the main definitions and results.

5.2.1 Directed Multigraphs

A *directed multigraph* is a structure $G = (V, E, \mathsf{src}, \mathsf{tgt})$ with nodes V, edges E, and two maps $\mathsf{src}, \mathsf{tgt} : E \to V$ giving the source and target of each edge, respectively. We write $e : s \to t$ if $s = \mathsf{src}\, e$ and $t = \mathsf{tgt}\, e$. When specifying multigraphs, we will sometimes use the notation $s \xrightarrow{n} t$ for the metastatement, "There are exactly n

edges from s to t."

A *path* is a finite alternating sequence of nodes and edges

$$s_0\, e_0\, s_1\, e_1\, s_2\, \cdots\, s_{n-1}\, e_{n-1}\, s_n,$$

$n \geq 0$, such that $e_i : s_i \to s_{i+1}$, $0 \leq i \leq n-1$. These are the arrows of the free category generated by G. The *length* of a path is the number of edges. A path of length 0 is just a single node. The first and last nodes of a path p are denoted $\mathsf{src}\, p$ and $\mathsf{tgt}\, p$, respectively. As with edges, we write $p : s \to t$ if $s = \mathsf{src}\, p$ and $t = \mathsf{tgt}\, p$.

A *multigraph homomorphism* $\ell : G_1 \to G_2$ is a map $\ell : V_1 \to V_2$, $\ell : E_1 \to E_2$ such that if $e : s \to t$ then $\ell(e) : \ell(s) \to \ell(t)$. This lifts to a functor on the free categories generated by G_1 and G_2.

5.2.2 Type Signatures

A *type signature* is a directed multigraph F along with a designation of each node of F as either *existential* or *universal*. The existential and universal nodes correspond respectively to coproduct and product constructors. The directed edges of the graph represent the corresponding destructors.

For example, consider an algebraic signature consisting of a binary function symbol f, a unary function symbol g, and a constant c. This would ordinarily be represented by the polynomial endofunctor $FX = X^2 + X + \mathbb{1}$, or in OCaml by

```
type t = F of t * t | G of t | C
```

We would represent this signature by a directed multigraph consisting of four nodes $\{t, f, g, c\}$, of which t is existential and f, g, c are universal, along with edges

$$t \xrightarrow[f]{1} \qquad t \xrightarrow[g]{1} \qquad t \xrightarrow[c]{1} \qquad f \xrightarrow[t]{2} \qquad g \xrightarrow[t]{1}\ .$$

The multigraph is illustrated in Fig. 5.1.

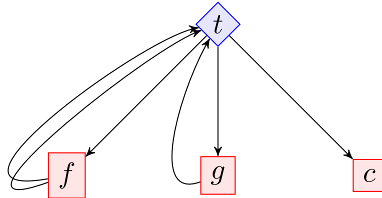


Figure 5.1: A multigraph representing a single-sorted algebraic signature. Blue diamonds represent existential nodes and red squares universal nodes.

5.2.3 Coalgebras and Realizations

Let F be a type signature with nodes V. An F-*coalgebra* is a V-indexed collection of pairs (C_s, γ_s), where the C_s are sets and the γ_s are set functions

$$\gamma_s : C_s \to \begin{cases} \sum_{\mathsf{src}\, e = s} C_{\mathsf{tgt}\, e}, & \text{if } s \text{ is existential,} \\ \prod_{\mathsf{src}\, e = s} C_{\mathsf{tgt}\, e}, & \text{if } s \text{ is universal.} \end{cases}$$

A morphism of F-coalgebras is a V-indexed collection of set maps h_s that commute with the γ_s in the usual way. Similarly, an F-*algebra* is a V-indexed collection of pairs (A_s, α_s), where the A_s are sets and the α_s are set functions

$$\alpha_s : \left\{ \begin{array}{ll} \sum_{\mathsf{src}\, e = s} A_{\mathsf{tgt}\, e}, & \text{if } s \text{ is existential,} \\ \prod_{\mathsf{src}\, e = s} A_{\mathsf{tgt}\, e}, & \text{if } s \text{ is universal} \end{array} \right\} \to A_s.$$

A morphism of F-algebras is a V-indexed collection of set maps h_s that commute with the α_s in the usual way. These definitions correspond to the traditional definition of F-coalgebras and F-algebras for an endofunctor F on Set^V.

Coalgebras are equivalent to *realizations*. An F-*realization* is a directed multigraph G along with a multigraph homomorphism $\ell : G \to F$, called a *typing*, with the following properties.

- If $\ell(u)$ is existential, then there is exactly one edge of G with source u.

- If $\ell(u)$ is universal, then ℓ is a bijection between the edges of G with source u and the edges of F with source $\ell(u)$.

A homomorphism of F-realizations is a multigraph homomorphism that commutes with the typings.

Theorem 5.2.1 ([Koz11]) *The categories of F-coalgebras and F-realizations are equivalent* (*in the sense of* [ML71, §IV.4]).

5.2.4 Final Coalgebras

Realizations allow a concrete construction of final coalgebras that is reminiscent of the Brzozowski derivative on sets of strings. Here, instead of strings, the derivative acts on certain sets of paths of the type signature.

Let F be a type signature. Construct a realization R, ℓ as follows. A node of R is a set A of finite paths in F such that

- A is nonempty and prefix-closed;
- all paths in A have the same first node, which we define to be $\ell(A)$;
- if p is a path in A of length n and $\mathsf{tgt}\, p$ is existential, then there is exactly one path of length $n + 1$ in A extending p;
- if p is a path in A of length n and $\mathsf{tgt}\, p$ is universal, then all paths of length $n + 1$ extending p are in A.

The edges of R are defined as follows. Let A be a set of paths in F and e an edge of F. Define the *Brzozowski derivative* of A with respect to e to be

$$D_e(A) = \{p \mid (\mathsf{src}\, e)\, e\, p \in A\},$$

the set of paths obtained by removing the initial edge e from paths in A that start with that edge. If A is a node of R and $D_e(A)$ is nonempty, we include exactly one edge

$$(A,\, e) : A \to D_e(A)$$

in R and take $\ell((A,\, e)) = e$. It is readily verified that $\mathsf{tgt}\,(A,\, e) = D_e(A)$ satisfies properties (i)–(iv) and that $\ell(D_e(A)) = \mathsf{tgt}\, e$, so ℓ is a typing.

Theorem 5.2.2 ([Koz11]) *The realization R, ℓ is final in the category of F-realizations. The corresponding F-coalgebra as constructed in Theorem 5.2.1 is final in the category of F-coalgebras.*

5.3 Characterization of Well-Founded Coalgebras

Well-foundedness of coalgebras has a precise characterization in terms of their corresponding realizations: a coalgebra is *well-founded* if and only if its corresponding realization is well-founded as a graph; that is, if it has no infinite directed paths. The main theorem of [ALM07] characterizes halting in terms of finiteness instead of well-foundedness, which by König's lemma is equivalent for the finitary functors considered in [ALM07], but it is really well-foundedness and not finiteness that is the essential property. In the following, we consider coalgebras for a wider class of functors, namely multi-sorted polynomial functors on Set^V, where V is a set of sorts, with infinite (countable and uncountable) product and sum, as well as total and partial functions. This is the same class of functors considered in [Koz11]. Let F be such a functor.

When a recursive function is called on a well-founded argument, the solution is unique and the standard semantics will terminate. Theorem 5.3.3, which generalizes

[ALM07] to the non-finitary case, characterizes the conditions under which this occurs.

The proof of Theorem 5.3.3 relies on some extra interesting facts which we also prove, namely that every F-coalgebra C contains a unique maximal well-founded subcoalgebra $\mathsf{wf}\, C$ and that if R is the final F-coalgebra, then $\mathsf{wf}\, R$ is the initial F-algebra.

5.3.1 Well-Founded Coalgebras

An *F-coalgebra-algebra morphism* is a set function $h : C \to A$, where (C, γ) is an F-coalgebra and (A, α) is an F-algebra, such that the diagram (5.1) commutes.

An F-realization $G = (V,\, E,\, \mathsf{src},\, \mathsf{tgt},\, \ell)$ is *well-founded* if all directed E-paths are finite. An F-coalgebra is *well-founded* if its corresponding F-realization is.

Lemma 5.3.1 *Every F-coalgebra contains a unique maximal well-founded subcoalgebra.*

Proof. Equivalently, every F-realization $G = (V,\, E,\, \mathsf{src},\, \mathsf{tgt},\, \ell)$ contains a unique maximal well-founded F-subrealization $\mathsf{wf}\, G$. The nodes $\mathsf{wf}\, V$ are the nodes of G from which there are no infinite directed E-paths. The graph $\mathsf{wf}\, G$ is the induced subgraph on $\mathsf{wf}\, V$. Equivalently, the set of nodes of $\mathsf{wf}\, G$ is the smallest set of nodes A of G satisfying the closure condition: if all E-successors of s are in A, then $s \in A$. □

Lemma 5.3.2 *Let $R = (V,\, E,\, \mathsf{src},\, \mathsf{tgt},\, \ell)$ be the final F-realization. Then $\mathsf{wf}\, R$ is an F-algebra.*

Proof. By Lambek's lemma [Lam68], the structure map $(\gamma_s \mid s \in V)$ of the final F-coalgebra corresponding to R is invertible, thus forms an F-algebra. Translating

back to the realization R, this means that

- for every edge $e \in E$ such that $\mathsf{src}\, e$ is existential and every node v of R with $\ell(v) = \mathsf{tgt}\, e$, there exists a unique node u and edge d of R such that $\mathsf{src}\, d = u$, $\mathsf{tgt}\, d = v$, and $\ell(d) = e$; and
- for every universal node $s \in V$ and tuple $(v_e \mid \mathsf{src}\, e = s)$ of nodes of R such that $\ell(v_e) = \mathsf{tgt}\, e$, there exist a unique node u and tuple of edges $(d_e \mid \mathsf{src}\, e = s)$ of R such that $\mathsf{src}\, d_e = u$, $\mathsf{tgt}\, d_e = v_e$, and $\ell(d_e) = e$.

The existence and uniqueness of u in the above two cases assert the closure of R under the algebraic operations. The subrealization $\mathsf{wf}\, R$ is closed under these operations, because any node all of whose immediate E-successors are in $\mathsf{wf}\, R$ is also in $\mathsf{wf}\, R$, therefore $\mathsf{wf}\, R$ is a subalgebra of R. □

We will show in Corollary 5.3.4 that $\mathsf{wf}\, R$ is in fact the initial F-algebra (up to isomorphism). To show initiality, we need to show that there is a unique F-algebra morphism to any other F-algebra. This will follow as a special case of Theorem 5.3.3(iv) below.

5.3.2 Induction Principle

The well-founded part of a realization G can be expressed in the modal μ-calculus as $\mathsf{wf}\, G = \mu X.\Box X$, where the modality $\Box$ is interpreted in G by the E-successor relation $E(x) = \{\mathsf{tgt}\, e \mid e \in E,\ \mathsf{src}\, e = x\}$; that is, the modal formula $\Box P$ holds of x if P holds of all E-successors of x. Thus G is well-founded if $\mu X.\Box X$ is universally valid in G.

The *induction principle* for a well-founded realization $G = (V,\, E,\, \mathsf{src},\, \mathsf{tgt},\, \ell)$ is:

$$\frac{\forall x\ (\forall y \in E(x)\ P(y)) \to P(x)}{\forall x\ P(x)}, \tag{5.2}$$

or more concisely,

$$\frac{\Box P \to P}{P}.$$

As we argue in Theorem 5.3.3, this rule is sound if and only if G is well-founded.

5.3.3 Main Theorem

We are now ready to state and prove our main theorem. We include point (v) to align with [ALM07, Theorem 3.8], although it is not really needed for our work.

Theorem 5.3.3 *Let (C, γ) be an F-coalgebra and let R be the final F-coalgebra. The following are equivalent:*

(i) *C is well-founded; that is, $C = \mathsf{wf}\, C$.*

(ii) *The induction principle* (5.2) *is valid for C.*

(iii) *There is a unique coalgebra morphism $C \to \mathsf{wf}\, R$.*

(iv) *There is a unique coalgebra-algebra morphism from C to any F-algebra.*

(v) *There is a unique parameterized coalgebra-algebra morphism from C to any F-algebra.*

Proof. The equivalence of (i) and (ii) is a fundamental property of relational algebra. The implication (i) $\Rightarrow$ (ii) requires the axiom of dependent choice.

Assuming (i) and (ii), (iv) can be proved by defining a coalgebra-algebra morphism by induction, using (5.2). Let (A_s, α_s) be an arbitrary F-algebra. Assume the coalgebra C is given in the form of an F-realization $G = (V, E, \mathsf{src}, \mathsf{tgt}, \ell)$. We must define maps $h_s : \ell^{-1}(s) \to A_s$ for $s \in V$ satisfying condition (5.1). This is equivalent to the following two conditions. Let $s \in V$ and $u \in V$ such that $\ell(u) = s$.

- If s is existential, let d be the unique edge with $\mathsf{src}\, d = u$, let $v = \mathsf{tgt}\, d$, and let $e = \ell(d)$. Then

$$h_s(u) = \alpha_s(\mathsf{in}_e(h_{\mathsf{tgt}\, e}(v))) \in A_s.$$

- If s is universal, for each e such that $\mathsf{src}\, e = s$, let d_e be the unique edge with $u = \mathsf{src}\, e$ and $\ell(d_e) = e$, and let $v_e = \mathsf{tgt}\, d_e$. Then

$$h_s(u) = \alpha_s(h_{\mathsf{tgt}\, e}(v_e) \mid \mathsf{src}\, e = s) \in A_s.$$

The maps h_s are uniquely defined by these equations due to the well-foundedness of the E-successor relation on G.

By Lemma 5.3.2, $\mathsf{wf}\, R$ is an F-algebra, thus (iii) follows as a special case of (iv).

To argue that (iii) implies (i), we observe that under any morphism of F-realizations $C \to \mathsf{wf}\, R$, an infinite path in C would map to an infinite path in $\mathsf{wf}\, R$, which cannot exist by definition, since $\mathsf{wf}\, R$ is well-founded. Thus C must be well-founded as well.

For (v) $\Rightarrow$ (iv), suppose that there is a unique parameterized coalgebra-algebra morphism from C to any F-algebra. That is, for any $\alpha' \colon FA \times C \to A$ there is a unique h which makes the following diagram commute:

$$\begin{array}{ccc} C & \xrightarrow{h} & A \\ {\scriptstyle (\gamma,\, \mathsf{id})}\downarrow & & \uparrow{\scriptstyle \alpha'} \\ FC \times C & \xrightarrow[Fh \times \mathsf{id}]{} & FA \times C \end{array} \tag{5.3}$$

We want to show that that there is a unique coalgebra-algebra morphism from C to any F-algebra.

Take an arbitrary F-algebra $\alpha \colon A \to FA$ and consider $\alpha' = \alpha \circ \pi_1 \colon FA \times C \to A$. Using the diagram (5.3), we know that there exists a unique $h \colon C \to A$ such that

$h = \alpha \circ \pi_1 \circ (Fh \times \mathsf{id}) \circ (\gamma,\, \mathsf{id})$. We show that h is a coalgebra-algebra morphism from C to A and that it is unique.

$$\begin{aligned} h &= \alpha \circ \pi_1 \circ (Fh \times \mathsf{id}) \circ (\gamma,\, \mathsf{id}) && \text{diagram (5.3)} \\ &= \alpha \circ Fh \circ \pi_1 \circ (\gamma,\, \mathsf{id}) && \pi_1 \text{ is a natural transformation} \\ &= \alpha \circ Fh \circ \gamma && \pi_1 \circ (f,\, g) = f. \end{aligned}$$

For uniqueness, note that any other coalgebra-algebra morphism $g : C \to A$ also makes diagram (5.3) commute, for $\alpha' = \alpha \circ \pi_1$:

$$\begin{aligned} g &= \alpha \circ Fg \circ \gamma && \text{definition of coalgebra-algebra morphism} \\ &= \alpha \circ Fg \circ \pi_1 \circ (\gamma,\, \mathsf{id}) && \pi_1 \circ \langle f, g \rangle = f \\ &= \alpha \circ \pi_1 \circ (Fg \times \mathsf{id}) \circ \gamma && \pi_1 \text{ is a natural transformation.} \end{aligned}$$

Hence $g = h$.

For (iv) $\Rightarrow$ (v), we need the following fact. Let $\gamma\colon C \to FC$ be an F-coalgebra. Define $G(X) = C \times FX$. If (C, γ) is a well-founded F-coalgebra, then $(C, (\gamma,\, \mathsf{id}))$ is a well-founded G-coalgebra. If (i) holds for F, then it also holds for G, therefore (iv) holds for G, and (v) follows trivially for F since the diagram (5.3) for F is a coalgebra-algebra morphism diagram for G. □

Corollary 5.3.4 *The F-coalgebra $\mathsf{wf}\, R$ is (up to isomorphism) the initial F-algebra.*

Proof. The structure $\mathsf{wf}\, R$ is an F-algebra by Lemma 5.3.2. But it is also a well-founded F-coalgebra by definition. By the equivalence of Theorem 5.3.3(i) and (iv), there is a unique F-algebra morphism from $\mathsf{wf}\, R$ to any F-algebra, thus $\mathsf{wf}\, R$ is initial. □

5.3.4 Non-Well-Founded Coalgebras

In many interesting non-well-founded cases, h is not unique and depends on the choice of solution method in the codomain A. However, for a large class of ordered codomains, one is interested in a canonical solution, namely the least fixpoint of a monotone map specified by the function definition. This situation was studied in [AMV06], in which it was shown that under certain conditions on the codomain, a function defined on a non-well-founded coalgebra can be considered a function on the final coalgebra and is independent of the input representation. This covers many examples in which the intended solution is a least fixpoint. The following result is a minor adaptation of [AMV06, Proposition 3.5] to our framework and the proof is similar.

Theorem 5.3.5 *Let (A, α) be an ordered F-algebra such that A is a chain-complete and α order-continuous. The construction of the least fixpoint of the map $h \mapsto \alpha \circ Fh \circ \gamma$ is natural in S; that is, if $f\colon S \to S'$ is an F-coalgebra morphism, then $h_S = h_{S'} \circ f$.*

Although Theorem 5.3.5 covers many interesting non-well-founded situations, there are some that it does not cover. For instance, to define substitution on infinitary λ-terms, the codomain is a coalgebra of infinitary terms, which is not ordered in any natural way. In this case, the solution is unique for other reasons.

5.4 Well-Founded Examples

In this section, we present examples of recursive functions which are well-founded. The first two, the greatest common divisor of two integers and the towers of Hanoi,

already fit the framework of [ALM07]. The other are guaranteed to have a unique solution using the multi-sorted extension to their framework that we have proposed.

5.4.1 Integer GCD

For integers $m, n \geq 0$ but not both 0, we would like to compute a triple (g, s, t) such that g is the greatest common divisor (gcd) of m and n and $sm + tn = g$. A recursive definition is

```
let rec gcd m n =
  if n = 0 then (m,1,0) else
  let (q,r) = (m/n, m mod n) in
  let (g,s,t) = gcd n r in
  (g,t,s-q)
```

This gives the following instantiation of (5.1):

$$\begin{array}{ccc} \mathbb{N} \times \mathbb{N} & \xrightarrow{h} & \mathbb{N} \times \mathbb{Z} \times \mathbb{Z} \\ \gamma \downarrow & & \uparrow \alpha \\ F(\mathbb{N} \times \mathbb{N}) & \xrightarrow[Fh]{} & F(\mathbb{N} \times \mathbb{Z} \times \mathbb{Z}) \end{array}$$

Here $FX = \mathbb{N} + X \times \mathbb{N}$ and

$$\gamma(m, n) = \begin{cases} \iota_0(m) & \text{if } n = 0 \qquad\qquad \alpha(\iota_0(g)) = (g, 1, 0) \\ \iota_1(n, m \bmod n, m/n) & \text{if } n \neq 0 \qquad\qquad \alpha(\iota_1(g, s, t, q)) = (g, t, s - q). \end{cases}$$

The theory of recursive coalgebras [ALM07] guarantees the existence of a unique function satisfying the diagram.

5.4.2 Towers of Hanoi

Another classic example of a recursive function is the towers of Hanoi. This mathematical game consists of three rods A, B and C and a number of disks of different sizes that can slide on any rod. At the beginning of the game, all disks are on rod A in order of size, smallest on top. The goal of the game is to find a procedure to move all disks to rod B while respecting the following rules:

- only one disk at a time can be moved
- a move consists of removing the upper disk from one of the rods and sliding in onto another rod, on top of other disks that might already be on that rod;
- no disk may be placed on top of a smaller disk.

For n disks, a recursive solution consists in recursively moving $n-1$ disks from the origin rod A to the third rod C, then moving the biggest disk from the origin rod A to the destination rod B, and finally recursively moving $n-1$ disks from the third rod C to the destination rod B. It is given by the following OCaml implementation, where o, d and t are the origin, destination and third rod, respectively:

```
let rec hanoi n o d t =
  if n = 0 then [ ] else
    (hanoi (n-1) o t d) @ [(o,d)] @ (hanoi (n-1) t d o)
```

Let R be the set of rods $\{A, B, C\}$. A move can be represented as an element of R^2 consisting of the origin and the destination of the move. This gives the following instantiation of (5.1):

$$\begin{array}{ccc}
\mathbb{N} \times R^3 & \xrightarrow{\quad h \quad} & (R^2)^* \\
\gamma \downarrow & & \uparrow \alpha \\
\mathbb{1} + R^2 \times (\mathbb{N} \times R^3) \times (\mathbb{N} \times R^3) & \xrightarrow[\quad Fh \quad]{} & \mathbb{1} + R^2 \times (R^2)^* \times (R^2)^*
\end{array}$$

Here $FX = \mathbb{1} + R^2 \times X$ and

$$\gamma(n, o, d, t) = \begin{cases} \iota_0(), & \text{if } n = 0, \\ \iota_1(o, d, (n-1, o, t, d), (n-1, t, d, o)), & \text{if } n \neq 0 \end{cases}$$

$$\alpha(\iota_0()) = \varepsilon \qquad \alpha(\iota_1(o, d, b, e)) = b \cdot (o, d) \cdot e.$$

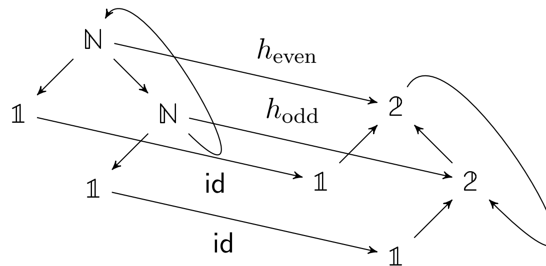
The theory of recursive coalgebras [ALM07] guarantees the existence of a unique function satisfying the diagram.

5.4.3 Mutually Recursive Functions: `even-odd`

This subsection illustrates how our generalization to multi-sorted signatures handles mutually recursive functions in a symmetric way. A very simple example is the definition of the even and odd predicates on natural numbers.

```
let rec even n = if n = 0 then true else odd (n-1)
and odd n = if n = 0 then false else even (n-1)
```

We can depict the recursion graphically with the following diagram:



This can be viewed as an endofunctor $F : \mathsf{Set}^V \to \mathsf{Set}^V$, where $V = \{\mathsf{even}, \mathsf{odd}\}$. The functor is defined by: $F(A, B) = (\mathbb{1} + B,\ \mathbb{1} + A)$ and if $g : A \to A'$ and $h : B \to B'$, then $F(g, h) = (\mathsf{id} + h, \mathsf{id} + g) : F(A, B) \to F(A', B')$.

An F-coalgebra is a pair $((C, D),\ \gamma)$, where $\gamma : (C, D) \to F(C, D)$ is a morphism in the underlying category Set^V; that is,

$$\gamma = (\gamma_{\mathsf{even}}, \gamma_{\mathsf{odd}}) : (C, D) \to (\mathbb{1} + D, \mathbb{1} + C),$$

where $\gamma_{\mathsf{even}} : C \to \mathbb{1} + D$ and $\gamma_{\mathsf{odd}} : D \to \mathbb{1} + C$. Similarly, an F-algebra is a pair $((A, B),\ \alpha)$, where $\alpha : F(A, B) \to (A, B)$ is a morphism in Set^V; that is,

$$\alpha = (\alpha_{\mathsf{even}}, \alpha_{\mathsf{odd}}) : (\mathbb{1} + B, \mathbb{1} + A) \to (A, B),$$

where $\alpha_{\mathsf{even}} : \mathbb{1} + B \to A$ and $\alpha_{\mathsf{odd}} : \mathbb{1} + A \to B$.

An F-algebra-coalgebra morphism $h\colon ((C, D), \gamma) \to ((A, B), \alpha)$ is a map $h = (h_{\mathsf{even}}, h_{\mathsf{odd}}) : (C, D) \to (A, B)$ such that the following diagram commutes:

$$\begin{array}{ccc} (C, D) & \xrightarrow{(h_{\mathsf{even}}, h_{\mathsf{odd}})} & (A, B) \\ {\scriptstyle (\gamma_{\mathsf{even}}, \gamma_{\mathsf{odd}})}\downarrow & & \uparrow{\scriptstyle (\alpha_{\mathsf{even}}, \alpha_{\mathsf{odd}})} \\ (\mathbb{1} + D, \mathbb{1} + C) & \xrightarrow[(\mathsf{id} + h_{\mathsf{odd}}, \mathsf{id} + h_{\mathsf{even}})]{} & (\mathbb{1} + B, \mathbb{1} + A) \end{array}$$

In our application, we have $A = B = \mathbb{2}$ and $C = D = \mathbb{N}$, with

$$\gamma_{\mathsf{even}}(n) = \gamma_{\mathsf{odd}}(n) = \begin{cases} \iota_0() & \text{if } n = 0 \\ \iota_1(n-1) & \text{if } n > 0 \end{cases}$$

$$\alpha_{\mathsf{even}}(\iota_0()) = \mathsf{true}$$

$$\alpha_{\mathsf{odd}}(\iota_0()) = \mathsf{false}$$

$$\alpha_{\mathsf{even}}(\iota_1(b)) = \alpha_{\mathsf{odd}}(\iota_1(b)) = b.$$

5.4.4 Ackermann Function

The Ackermann function

$$A(0, n) = n + 1 \tag{5.4}$$

$$A(m + 1, 0) = A(m, 1) \tag{5.5}$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n)) \tag{5.6}$$

is a notoriously fast-growing function that also fits into our general scheme (although one should not try to compute it!). This example is quite interesting, because at first glance it seems not to fit into the general scheme (5.1) because of the nested recursive call in the third clause. However, a key insight comes from the termination proof, which is done by induction on the well-founded lexicographic order on $\mathbb{N} \times \mathbb{N}$ with m as the more significant parameter. We see that we can break the definition into two stages, both higher-order.

Rewriting $A(m, n)$ as $A_m(n)$, we have that (5.4)–(5.6) is equivalent to

$$A_0 = \lambda n.n + 1$$

$$A_{m+1} = \lambda n.A_m^{n+1}(1),$$

where f^n denotes the n-fold composition of f with itself:

$$f^0 = \lambda n.n$$

$$f^{n+1} = f \circ f^n.$$

The outermost stage computes $m \mapsto A_m$. The diagram is

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{A} & \mathbb{N}^{\mathbb{N}} \\ \gamma \downarrow & & \uparrow \alpha \\ \mathbb{1} + \mathbb{N} & \xrightarrow[\mathsf{id}_{\mathbb{1}} + A]{} & \mathbb{1} + \mathbb{N}^{\mathbb{N}} \end{array}$$

where

$$\gamma(0) = \iota_0()$$

$$\gamma(m+1) = \iota_1(m)$$

$$\alpha(\iota_0()) = \lambda n.n+1$$

$$\alpha(\iota_1(f)) = \lambda n.f^{n+1}(1).$$

In turn, the function α is defined in terms the n-fold composition function $(n, f) \mapsto f^n$:

$$\begin{array}{ccc} \mathbb{N} \times D^D & \xrightarrow{\mathsf{comp}} & D^D \\ \gamma \downarrow & & \uparrow \alpha \\ F(\mathbb{N} \times D^D) & \xrightarrow[F(\mathsf{comp})]{} & F(D^D) \end{array}$$

where $FX = \mathbb{1} + D^D \times X$ and

$$\gamma(0, f) = \iota_0()$$

$$\gamma(n+1, f) = \iota_1(f, n, f)$$

$$\alpha(\iota_0()) = \mathsf{id}_D$$

$$\alpha(\iota_1(f, g)) = f \circ g.$$

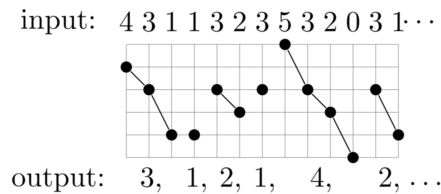
5.5 Non-Well-Founded Examples

Chapters 6 and 7 present many examples of non-well-founded functions, including probabilistic protocols, p-adic numbers, and a fairly substantial example involving abstract interpretation. To give a taste of those non-well-founded examples, we present here a few others.

5.5.1 Descending Sequences

As the simplest nontrivial coinductive datatype, *streams* offer the ideal playground to test new theories. We present an example on streams of natural numbers $\mathbb{N}^\omega$. The following example, taken from a talk by Capretta [Cap07], has a unique solution, but does not fit the existing theory of well-founded coalgebras [ALM07] or our generalization presented here, nor does it fit the theory of core corecursive algebras [CUV09].

The goal is to produce from a given stream of natural numbers another stream of natural numbers containing the lengths of the maximal strictly descending subsequences of the input stream. An example is shown in the following figure, where the input stream is depicted in a grid to easily picture the order of elements.



Here is a simple recursive definition of the function in CoCaml (see chapter 7), where the `constructor` solver builds a new stream:

```
let descending arg =
  let corec[constructor] descending_aux (n, i :: j :: t) =
    if i > j then descending_aux (n+1, j :: t)
             else n :: descending_aux (1, j :: t) in
  descending_aux (1, arg)
```

This definition corresponds to the following instantiation of (5.1):

$$\begin{array}{ccc}
\mathbb{N} \times \mathbb{N}^\omega & \xrightarrow{\quad h \quad} & \mathbb{N}^\omega \\
\gamma \downarrow & & \uparrow \alpha \\
\mathbb{N} \times \mathbb{N}^\omega + \mathbb{N} \times (\mathbb{N} \times \mathbb{N}^\omega) & \xrightarrow[h + \mathsf{id}_{\mathbb{N}} \times h]{} & \mathbb{N}^\omega + \mathbb{N} \times \mathbb{N}^\omega
\end{array}$$

where $FX = X + \mathbb{N} \times X$ and

$$\gamma(n, i :: j :: t) = \begin{cases} \iota_0(n+1, j :: t) & \text{if } i > j \\ \iota_1(n, (1, j :: t)) & \text{otherwise} \end{cases} \qquad \alpha(\iota_0(s)) = s \quad \alpha(\iota_1(n, s)) = n :: s.$$

5.5.2 Alternating Turing Machines and IND Programs

The semantics of alternating Turing machines is described in terms of an inductive labeling of machine configurations C with either false (rejecting), true (accepting), or $\bot$ (undetermined). In the present framework, the function γ would give the set of successor configurations and the labeling of the state as either existential or universal, and α would tell how to label configurations false, true, or $\bot$ inductively up the computation tree. Formally, α gives the infimum for universal configurations and supremum for existential configurations in 3-valued Kleene logic $\mathbb{3} = \{\mathsf{false}, \bot, \mathsf{true}\}$ with ordering $\mathsf{false} \le \bot \le \mathsf{true}$.

$$\begin{array}{ccc}
C & \xrightarrow{\quad h \quad} & \mathbb{3} \\
\gamma \downarrow & & \uparrow \alpha \\
\mathbb{2} \times \mathcal{P}_{\text{fin}}(C) & \xrightarrow[\mathsf{id}_{\mathbb{2}} + \mathcal{P}_{\text{fin}}(h)]{} & \mathbb{2} \times \mathcal{P}_{\text{fin}}(\mathbb{3})
\end{array}$$

The canonical solution is defined to be the least fixpoint with respect to a different order, namely the flat Scott order $\bot \sqsubseteq \mathsf{false}$, $\bot \sqsubseteq \mathsf{true}$. This example is interesting, because it is a case in which α is not strict; for example, a universal configuration

can be labeled false as soon as one of its successors is known to be labeled false, regardless of the labels of the other successors.

A similar model is the IND programming language for the inductive sets [HK84]. An IND program consists of a sequence of labeled statements of three kinds: universal and existential assignment ($x := \forall$ and $x := \exists$, respectively), conditional test (if $s = t$ then ℓ_1 else ℓ_2), and halting (accept, reject). IND programs accept exactly the inductively definable sets, which over $\mathbb{N}$ are exactly the Π^1_1 sets. The semantics is identical to alternating Turing machines, except that the branching degree is equal to the cardinality of the domain of computation, thus the finite powerset functor must be replaced by the unrestricted powerset functor.

5.6 Discussion

In this chapter, we have presented the origins of our work on bringing coinduction to a functional language in the form of effective language constructs.

The work in the present chapter and chapters 6 and 7 was inspired by work on recursive coalgebras [ALM07] and Elgot algebras [AMV06]. We have extended and clarified the results in [ALM07] by providing a different proof that works on a larger class of functors. Our generalization handles multi-sorted signatures and mutually recursive functions in a symmetric way and is not restricted to finitary functors. We have also provided several examples of functions defined using this scheme, as well as non-well-founded examples that do not have a unique solution but still have a canonical solution. Finally, we have briefly described our work on programming language constructs to allow the programmer to choose alternative solution methods when the standard semantics of recursion would not halt.

Chapter 6

Language Constructs for Non-Well-Founded Computation

Recursive functions defined on a coalgebraic datatype C may not converge if there are cycles in the input, that is, if the input object is not well-founded. Even so, there is often a useful solution. Unfortunately, current functional programming languages provide no support for specifying alternative solution methods. In this chapter we give numerous examples in which it would be useful to do so: free variables, α-conversion, and substitution in infinitary λ-terms; halting probabilities and expected running times of probabilistic protocols; abstract interpretation; and constructions involving finite automata. In each case the function would diverge under the standard semantics of recursion. We propose programming language constructs that would allow the specification of alternative solutions and methods to compute them.

6.1 Introduction

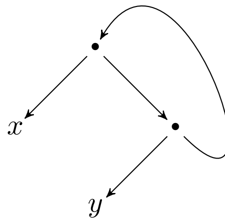
Coalgebraic datatypes have become popular in recent years in the study of infinite behaviors and non-terminating computation. One would like to define functions on coinductive datatypes by structural recursion, but such functions may not converge if there are cycles in the input; that is, if the input object is not well-founded. Even so, there is often a useful solution that we would like to compute.

For example, consider the problem of computing the set of free variables of a λ-term. In pseudo-ML, we might write

```
type term =                      let rec fv = function
  | Var of string                  | Var v -> {v}
  | App of term * term             | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam of string * term           | Lam (x,t) -> (fv t) − {x}
```

and this works provided the argument is an ordinary (well-founded) λ-term. However, if we call the function on an infinitary term (λ-coterm), say

```
let rec t = App (Var "x", App (Var "y", t))
```



(6.1)

then the function will diverge, even though it is clear the answer should be $\{x, y\}$. Note that this is not a corecursive definition: we are not asking for a greatest solution or a unique solution in a final coalgebra, but rather a least solution in a different ordered domain from the one provided by the standard semantics of recursive functions. The standard semantics gives us the least solution in the flat Scott domain $(\mathcal{P}(\mathbf{string})_\bot, \sqsubseteq)$ with bottom element $\bot$ representing nontermination,

whereas we would like the least solution in a different CPO, namely $(\mathcal{P}(\mathbf{string}), \subseteq)$ with bottom element $\varnothing$.

The coinductive elements we consider are always *regular*, that is, they have a finite but possibly cyclic representation. This is different from a setting in which infinite elements are represented lazily. A few of our examples, like substitution, could be computed by lazy evaluation, but most of them, for example free variables, could not.

Theoretically, the situation is similar to the one is chapter 5, and is governed by diagrams of the form (5.1):

$$\begin{array}{ccc} C & \xrightarrow{h} & A \\ {\scriptstyle\gamma}\downarrow & & \uparrow{\scriptstyle\alpha} \\ FC & \xrightarrow[Fh]{} & FA \end{array}$$

describing a recursive definition of a function $h : C \to A$. Here F is a functor describing the structure of the recursion. To apply h to an input x, the function $\gamma : C \to FC$ identifies the base cases, and in the recursive case prepares the arguments for the recursive calls; the function $Fh : FC \to FA$ performs the recursive calls; and the function $\alpha : FA \to A$ assembles the return values from the recursive calls into final value $h(x)$.

A canonical example is the usual factorial function

```
let rec factorial = function
  | 0 -> 1
  | n -> n * factorial (n-1)
```

Here the abstract diagram (5.1) becomes

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{h} & \mathbb{N} \\ \gamma \downarrow & & \uparrow \alpha \\ \mathbb{1} + \mathbb{N} \times \mathbb{N} & \xrightarrow[\mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{N}} \times h]{} & \mathbb{1} + \mathbb{N} \times \mathbb{N} \end{array} \tag{6.2}$$

where the functor is $FX = \mathbb{1} + \mathbb{N} \times X$ and γ and α are given by:

$$\gamma(0) = \iota_0() \qquad \alpha(\iota_0()) = 1$$

$$\gamma(n+1) = \iota_1(n+1, n) \qquad \alpha(\iota_1(c,d)) = cd$$

where ι_0 and ι_1 are injectors into the coproduct. The fact that there is one recursive call is reflected in the functor by the single X occurring on the right-hand side. The function γ determines whether the argument is the base case 0 or the inductive case $n+1$, and in the latter case prepares the recursive call. The function α combines the result of the recursive call with the input value by multiplication. In this case we have a unique solution, which is precisely the factorial function.

Ordinary recursion over inductive datatypes corresponds to the case in which C is well-founded. In this case, the solution h exists and is unique: it is the least solution in the standard flat Scott domain. For example, the factorial function is uniquely defined by (6.2) in this sense. If C is not well-founded, there can be multiple solutions, and the one provided by the standard semantics of recursion is typically not be the one we want. Nevertheless, the diagram (5.1) can still serve as a valid definitional scheme, provided we are allowed to specify a desired solution. In the free variables example, the codomain of the function (sets of variables) is indeed a complete CPO under the usual set inclusion order, and the constructor α is continuous, thus the desired solution can be obtained by a least fixpoint computation.

The example (6.1) involving free variables of a λ-coterm fits this scheme with the diagram

$$\begin{array}{ccc} \mathtt{Term} & \xrightarrow{\mathtt{fv}} & \mathcal{P}(\mathtt{Var}) \\ \gamma \downarrow & & \uparrow \alpha \\ F(\mathtt{Term}) & \xrightarrow[\mathsf{id}_{\mathtt{Var}} + \mathtt{fv}^2 + \mathsf{id}_{\mathtt{Var}} \times \mathtt{fv}]{} & F(\mathcal{P}(\mathtt{Var})) \end{array}$$

where $FX = \mathtt{Var} + X^2 + \mathtt{Var} \times X$ and

$$\begin{aligned} \gamma(\mathtt{Var}\ x) &= \iota_0(x) & \alpha(\iota_0(x)) &= \{x\} \\ \gamma(\mathtt{App}\ (t_1, t_2)) &= \iota_1(t_1, t_2) & \alpha(\iota_1(u, v)) &= u \cup v \\ \gamma(\mathtt{Lam}\ (x, t)) &= \iota_2(x, t) & \alpha(\iota_2(x, v)) &= v \setminus \{x\}. \end{aligned}$$

Here the domain of $\mathtt{fv}$ (regular λ-coterms) is not well-founded and the codomain (sets of variables) is not a final coalgebra, but the codomain is a complete CPO under the usual set inclusion order with bottom element $\varnothing$, and the desired solution is the least solution in this order; it is just not the one that would be computed by the standard semantics of recursive functions.

Unfortunately, current programming languages provide little support for specifying alternative solutions. One must be able to specify a canonical method for solving systems of equations over an F-algebra (the codomain) obtained from the function definition and the input. We will demonstrate through several examples that such a feature would be extremely useful in a programming language and would bring coinduction and coinductive datatypes to a new level of usability in accordance with the elegance already present for algebraic datatypes. Our examples include free variables, α-conversion, and substitution in infinitary terms; halting probabilities, expected running times, and outcome functions of probabilistic protocols; and ab-

stract interpretation. In each case, the function would diverge under the standard semantics of recursion.

In this chapter we propose programming language constructs that would allow the specification of alternative solutions and methods to compute them. These examples require different solution methods: iterative least fixpoint computation, Gaussian elimination, structural coinduction. We describe how this feature might be implemented in a functional language and give mock-up implementations of all our examples. In our implementation, we show how the function definition specifies a system of equations and indicate how that system of equations might be extracted automatically and then passed to an equation solver. In many cases, we suspect that the process can be largely automated, requiring little extra work on the part of the programmer.

Current functional languages are not particularly well suited to the manipulation of coinductive datatypes. For example, in OCaml one can form coinductive objects with **let rec** as in (6.1), but due to the absence of mutable variables, such objects can only be created and not dynamically manipulated, which severely limits their usefulness. One can simulate them with references, but this negates the elegance of algebraic manipulation of inductively defined datatypes, for which the ML family of languages is so well known. It would be of benefit to be able to treat coinductive types the same way.

Our mock-up implementation with all examples and solvers is available from [CoC12].

6.2 Motivating Examples

In this section we present a number of motivating examples that illustrate the usefulness of the problem. Several examples of well-founded definitions that fit the scheme (5.1) can be found in the cited literature, including the Fibonacci function and various divide-and-conquer algorithms such as quicksort and mergesort, so we focus on non-well-founded examples: free variables and substitution in λ-coterms, probabilistic protocols, and abstract interpretation.

6.2.1 Substitution

We now describe another function on infinitary λ-terms: substitution. A typical implementation for well-founded terms would be

```
let rec subst t y = function
  | Var x -> if x = y then t else Var x
  | App (t1,t2) -> App (subst t y t1, subst t y t2)
  | Lam (x,s) -> if x = y then Lam (x,s)
                 else if x ∈ fv t then
                   let w = fresh ()
                   in Lam (w, subst t y (rename w x s))
                 else Lam (x, subst t y s)
```

where `fv` is the free variable function defined above and `rename w x s` is a function that substitutes a fresh variable `w` for `x` in a term `s`.

```
let rec rename w x = function
  | Var z -> Var (if z = x then w else z)
  | App (t1,t2) -> App (rename w x t1, rename w x t2)
  | Lam (z,s) -> if z = x then Lam (z,s)
```

```
else Lam (z, rename w x s)
```

Applied to a λ-coterm with a cycle, for example attempting to substitute a term for y in (6.1), the computation would never finish. Nevertheless, this computation fits the scheme (5.1) with $C = A = \texttt{term}$ (the set of λ-coterms), functor

$$FX = \texttt{term} + X^2 + \texttt{string} \times X \qquad Fh = \mathsf{id}_{\texttt{term}} + h^2 + \mathsf{id}_{\texttt{string}} \times h$$

and γ and α defined by

$$\gamma(\texttt{Var}\ x) = \begin{cases} \iota_0(t) & \text{if } x = y \\ \iota_0(\texttt{Var}\ x) & \text{otherwise} \end{cases}$$

$$\gamma(\texttt{App}\ (t_1, t_2)) = \iota_1(t_1, t_2)$$

$$\gamma(\texttt{Lam}\ (x, s)) = \begin{cases} \iota_0(\texttt{Lam}\ (x, s)) & \text{if } x = y \\ \iota_2(w, \texttt{rename}\ w\ x\ s) & \text{if } x \neq y \text{ and } x \in \texttt{fv}\ t\text{, where } w \text{ is fresh} \\ \iota_2(x, s) & \text{otherwise} \end{cases}$$

$$\alpha(\iota_0(s)) = s$$

$$\alpha(\iota_1(s_1, s_2)) = \texttt{App}\ (s_1, s_2)$$

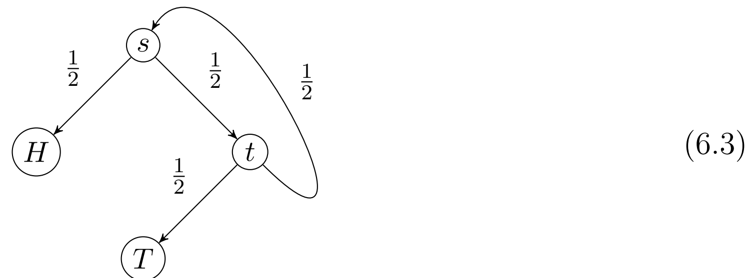
$$\alpha(\iota_2(x, s)) = \texttt{Lam}\ (x, s)$$

In this case, even though the domain is not well-founded, the solution nevertheless exists and is unique up to observational equivalence. This is because the definition of the function is corecursive and takes values in a final coalgebra.

6.2.2 Probabilistic Protocols

In this section, we present a few examples in the realm of probabilistic protocols. Imagine one wants to simulate a biased coin, say a coin with probability 2/3 of

heads, with a fair coin. Here is a possible solution: flip the fair coin. If it comes up heads, output heads, otherwise flip again. If the second flip is tails, output tails, otherwise repeat from the start. This protocol can be represented succinctly by the following probabilistic automaton:



(6.3)

Operationally, starting from states s and t, the protocol generates series that converge to 2/3 and 1/3, respectively.

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{8} + \tfrac{1}{32} + \tfrac{1}{128} + \cdots = \tfrac{2}{3}$$
$$\mathsf{Pr}_H(t) = \tfrac{1}{4} + \tfrac{1}{16} + \tfrac{1}{64} + \tfrac{1}{256} + \cdots = \tfrac{1}{3}.$$

However, these values can also be seen to satisfy a pair of mutually recursive equations:

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{Pr}_H(t) \qquad\qquad \mathsf{Pr}_H(t) = \tfrac{1}{2} \cdot \mathsf{Pr}_H(s).$$

This gives rise to a contractive map on the unit interval, which has a unique solution. It is also monotone and continuous with respect to the natural order on the unit interval, therefore has a unique least solution.

One would like to define the probabilistic automaton (6.3) by

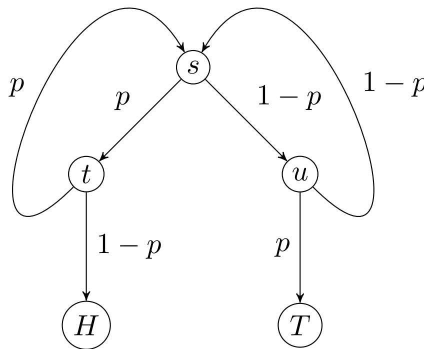
```
type pa = H | T | Flip of float * pa * pa
let rec s = Flip (0.5,H,t) and t = Flip (0.5,T,s)
```

and write a recursive program, say something like

```
let rec pr_heads = function
  | H -> 1.
  | T -> 0.
  | Flip (p,u,v) -> p *. (pr_heads u) +. (1 -. p) *. (pr_heads v)
```

and specify that the extracted equations should be solved exactly by Gaussian elimination, or by iteration until achieving a fixpoint to within a sufficiently small error tolerance ε. We give implementations using both methods.

The *von Neumann trick* for simulating a fair coin with a coin of arbitrary bias is a similar example. In this protocol, we flip the coin twice. If the outcome is HT, we output heads. If the outcome is TH, we output tails. These outcomes occur with equal probability. If the outcome is HH or TT, we repeat.



Here we would define

```
let rec s = Flip (p,t,u) and t = Flip (p,s,H) and u = Flip (p,T,s)
```

but the typing and recursive function `pr_heads` are the same. Markov chains and Markov decision processes can be modeled the same way.

Other functions on probabilistic automata can be computed as well. The expected number of steps starting from state s is the least solution of the equation

$$E(s) = \begin{cases} 0 & \text{if } s \in \{\mathtt{H},\mathtt{T}\} \\ 1 + p \cdot E(u) + (1-p) \cdot E(v) & \text{if } s = \mathtt{Flip}(p,u,v). \end{cases}$$

We would like to write simply

```
let rec ex = function
  | H -> 0.
  | T -> 0.
  | Flip (p,u,v) -> 1. +. p *. (ex u) +. (1 -. p) *. (ex v)
```

and specify that the extracted equations should be solved by Gaussian elimination or least fixpoint iteration from 0.

The coinflip protocols we have discussed all fit the abstract definitional scheme (5.1) in the form

$$\begin{array}{ccc} S & \xrightarrow{h} & \mathbb{R} \\ {\scriptstyle\gamma}\downarrow & & \uparrow{\scriptstyle\alpha} \\ FS & \xrightarrow[Fh]{} & F\mathbb{R} \end{array}$$

where S is the set of states (a state can be either `H`, `T`, or a triple (p, u, v), where $p \in \mathbb{R}$ and $u, v \in S$, the last indicating that it flips a p-biased coin and moves to state u with probability p and v with probability $1 - p$), F is the functor

$$FX = \mathbb{1} + \mathbb{1} + \mathbb{R} \times X^2 \qquad\qquad Fh = \mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{R}} \times h^2.$$

For both the probability of heads and expected running times examples, we can take

$$\gamma(s) = \begin{cases} \iota_0() & \text{if } s = \mathtt{H} \\ \iota_1() & \text{if } s = \mathtt{T} \\ \iota_2(p, u, v) & \text{if } s = (p, u, v). \end{cases}$$

For the probability of heads, we can take

$$\alpha(\iota_0()) = 1 \qquad \alpha(\iota_1()) = 0 \qquad \alpha(\iota_2(p, a, b)) = pa + (1 - p)b.$$

For the expected running time, we can take

$$\alpha(\iota_0()) = \alpha(\iota_1()) = 0 \qquad\qquad \alpha(\iota_2(p, a, b)) = 1 + pa + (1-p)b.$$

The desired solution in all cases is a least fixpoint in an appropriate ordered domain.

6.2.3 Abstract Interpretation

In this section we present our most involved example: abstract interpretation of a simple imperative language. Our example follows Cousot and Cousot [CC77] as inspired by lecture notes of Stephen Chong [Cho10].

Consider a simple imperative language of while programs with integer expressions a and commands c. Let Var be a countable set of variables.

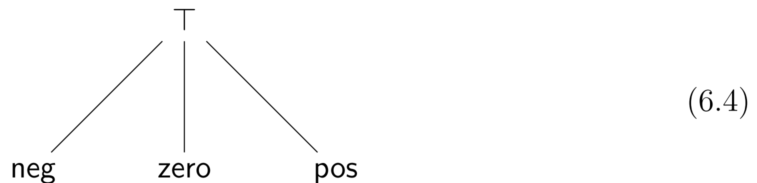
$$a ::= n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2$$

$$c ::= \mathbf{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \mathsf{if}\ a\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ a\ \mathsf{do}\ c$$

For the purpose of tests in the conditional and while loop, an integer is considered true if and only if it is nonzero. Otherwise, the operational semantics is standard, in the style of [Win93]. A store is a partial function from variables to integers, an arithmetic expression is interpreted relative to a store and returns an integer, and a command is interpreted relative to a store and returns an updated store.

Abstract interpretation defines an abstract domain that approximates the values manipulated by the program. We define an abstract domain for integers that abstracts an integer by its sign. The set of abstract values is $\mathsf{AbsInt} = \{\mathsf{neg}, \mathsf{zero}, \mathsf{pos}, \top\}$, where neg, zero, and pos represent negative, zero, and positive integers, repectively, and $\top$ represents an integer of unknown sign. The abstract values form a join

semilattice with $\sqcup$ defined by the following diagram:



(6.4)

The abstract interpretation of an arithmetic expression is defined relative to an abstract store $\sigma : \mathsf{Var} \rightharpoonup \mathsf{AbsInt}$, used to interpret the abstract values of variables. We write $\mathsf{AS} = \mathsf{Var} \rightharpoonup \mathsf{AbsInt}$ for the set of abstract stores. The abstract interpretation of arithmetic expressions is given by:

$$\mathcal{A}[\![n]\!]\sigma = \begin{cases} \mathsf{pos} & \text{if } n > 0 \\ \mathsf{zero} & \text{if } n = 0 \\ \mathsf{neg} & \text{if } n < 0 \end{cases}$$

$$\mathcal{A}[\![x]\!]\sigma = \sigma(x)$$

$$\mathcal{A}[\![a_1 + a_2]\!] = \begin{cases} \mathcal{A}[\![a_1]\!]\sigma & \text{if } \mathcal{A}[\![a_2]\!]\sigma = \mathsf{zero} \\ \mathcal{A}[\![a_2]\!]\sigma & \text{if } \mathcal{A}[\![a_1]\!]\sigma = \mathsf{zero} \\ \mathcal{A}[\![a_1]\!]\sigma \sqcup \mathcal{A}[\![a_2]\!]\sigma & \text{otherwise.} \end{cases}$$

The abstract interpretation of commands returns an abstract store, which is an abstraction of the concrete store returned by the commands. Abstract stores form a join semilattice, where the join of two abstract stores just takes the join of each variable: $(\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup \sigma_2(x)$. Commands other than the while loop are interpreted as follows:

$$\mathcal{C}[\![\mathbf{skip}]\!]\sigma = \sigma \qquad \mathcal{C}[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \qquad \mathcal{C}[\![c_1 \,;\, c_2]\!]\sigma = \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$$

$$\mathcal{C}[\![\textsf{if } a \textsf{ then } c_1 \textsf{ else } c_2]\!]\sigma = \begin{cases} \mathcal{C}[\![c_1]\!]\sigma & \text{if } \mathcal{A}[\![a]\!]\sigma \in \{\textsf{pos}, \textsf{neg}\} \\ \mathcal{C}[\![c_2]\!]\sigma & \text{if } \mathcal{A}[\![a]\!]\sigma = \textsf{zero} \\ \mathcal{C}[\![c_1]\!]\sigma \sqcup \mathcal{C}[\![c_2]\!]\sigma & \text{otherwise.} \end{cases}$$

We would ideally like to define

$$\mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!]\sigma = \begin{cases} \sigma & \text{if } \mathcal{A}[\![a]\!]\sigma = \textsf{zero} \\ \sigma \sqcup \mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!](\mathcal{C}[\![c]\!]\sigma) & \text{otherwise.} \end{cases}$$

Unfortunately, when $\mathcal{A}[\![a]\!]\sigma \neq \textsf{zero}$, the definition is not well-founded, because it is possible for σ and $\mathcal{C}[\![c]\!]\sigma$ to be equal. However, it is a correct definition of $\mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!]$ as a least fixpoint in the join semilattice of abstract stores. The existence of the least fixpoint can be obtained in a finite time by iteration because the join semilattice of abstract stores satisfies the ascending chain condition (ACC), that is, it does not contain any infinite ascending chains.

Given $\mathcal{A}[\![a]\!]$ and $\mathcal{C}[\![c]\!]$ previously defined, $\mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!]$ satisfies the following instantiation of (5.1):

$$\begin{array}{ccc} \textsf{AS} & \xrightarrow{\mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!]} & \textsf{AS} \\ \gamma \downarrow & & \uparrow \alpha \\ \textsf{AS} + \textsf{AS} \times \textsf{AS} & \xrightarrow[\textsf{id}_{\textsf{AS}} + \textsf{id}_{\textsf{AS}} \times \mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!]]{} & \textsf{AS} + \textsf{AS} \times \textsf{AS} \end{array}$$

where the functor is $FX = \textsf{AS} + \textsf{AS} \times X$ and

$$\gamma(\sigma) = \begin{cases} \iota_1(\sigma) & \text{if } \mathcal{A}[\![a]\!]\sigma = \textsf{zero} \\ \iota_2(\sigma, \mathcal{C}[\![c]\!]\sigma) & \text{otherwise} \end{cases} \qquad \begin{array}{rcl} \alpha(\iota_1(\sigma)) & = & \sigma \\ \alpha(\iota_2(\sigma, \tau)) & = & \sigma \sqcup \tau \end{array}$$

The function $\mathcal{C}[\![\textsf{while } a \textsf{ do } c]\!]$ is the least function in the pointwise order that makes the above diagram commute.

This technique allows us to define $\mathcal{C}[\![c]\!]$ inductively on the structure of c. An inductive definition can be used here because the set of abstract syntax trees is well-founded.

The literature on abstract interpretation explains how to compute the least fixpoint, and much research has been done on techniques for accelerating convergence to the least fixpoint. This body of research can inform compiler optimization techniques for computation with coalgebraic types.

6.2.4 Finite Automata

We conclude this section with a brief example involving finite automata. Suppose we want to construct a deterministic finite automaton (DFA) over a two-letter alphabet accepting the intersection of two regular sets given by two other DFAs over the same alphabet. We might define states coalgebraically by

```
type state = State of bool * state * state
```

where the first component specifies whether the state is an accepting state and the last two components give the states to move to under the two input symbols. The standard product construction is defined coalgebraically simply by

```
let rec product (s : state) (t : state) : state =
   match s, t with
     | State (b1,s1,t1), State (b2,s2,t2) ->
         State (b1 && b2, product s1 t1, product s2 t2)
```

and we can compute it, provided we can solve the generated equations.

6.3 A Framework for Non-Well-Founded Computation

In this section we discuss our proposed framework for incorporating language constructs to support non-well-founded computation. At a high level, we wish to specify a function h uniquely using a finite set E of structural recursive equations. The function is defined in much the same way as an ordinary recursive function on an inductive datatype. However, the value $h(x)$ of the function on a particular input x is computed not by calling the function in the usual sense, but by generating a system of equations from the function definition and then passing the equations to a specified equation solver to find a solution. The equation solver is either a standard library function or programmed by the user according to an explicit interface.

The process is partitioned into several tasks as follows.

1. The left-hand sides of the clauses in the function definition determine syntactic terms representing equation schemes. These schemes are extracted by the compiler from the abstract syntax tree of the left-hand side expressions. This determines (more or less, subject to optimizations) the function γ in the diagram (5.1).

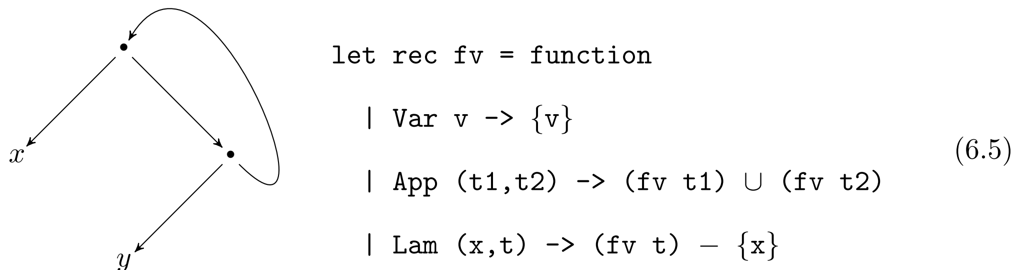
2. The right-hand sides of the clauses in the function definition determine the function α in the diagram (5.1) (again, more or less, subject to optimizations). These expressions essentially tell how to evaluate terms extracted in step 1 in the codomain. As in 1, these are determined by the compiler from the abstract syntax trees of the right-hand sides.

3. At runtime, when the function is called with a coalgebraic element c, a finite

system of equations is generated from the schemes extracted in steps 1 and 2, one equation for each element of the coalgebra reachable from c. In fact, we can take the elements reachable from c as the variables in our equations. Each such element matches exactly one clause of the function body, and this determines the right-hand side of the equation that is generated.

4. The equations are passed to a solver that is specified by the user. This will presumably be a module that is programmed separately according to a fixed interface and available as a library function. There should be a simple syntactic mechanism for specifying an alternative solution method (although we do not specify here what that should look like).

Let us illustrate this using our initial example of the free variables. Recall the infinitary λ-term below and the definition of the free variables function from the introduction:




```
let rec fv = function
  | Var v -> {v}
  | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam (x,t) -> (fv t) − {x}
```

(6.5)

Steps 1 and 2 would analyze the left-and right-hand sides of the three clauses in the body at compile time to determine the equation schemes. Then at runtime, if the function were called on the coalgebraic element pictured, the runtime system would generate four equations, one for each node reachable from the top node:

```
fv t = (fv x) ∪ (fv u)    fv u = (fv y) ∪ (fv t)    fv x = {x}    fv y = {y}
```

where `t` and `u` are the unlabeled top and right nodes of the term above.

As noted, these equations have many solutions. In fact, any set containing the variables `x` and `y` will be a solution. However, we are interested in the least solution in the ordered domain $(\mathcal{P}(\mathsf{Var}), \subseteq)$ with bottom element $\varnothing$. In this case, the least solution would assign $\{\mathtt{x}\}$ to the leftmost node, $\{\mathtt{y}\}$ to the lowest node, and $\{\mathtt{x,y}\}$ to the other two nodes.

With this in mind, we would pass the generated equations to an iterative equation solver, which would produce the desired solution. In many cases, such as this example, the codomain is a complete partial order and we have default solvers to compute least fixpoints, leaving to the programmer the simple task of indicating that this is the desired solution method. That would be an ideal situation: the defining equations of (6.5) plus a simple tag would be enough to obtain the desired solution.

6.3.1 Generating Equations

The equations are generated from the recursive function definition and the input c, a coalgebraic element, in accordance with the abstract definitional scheme (5.1). The variables can be taken to be the elements of the coalgebraic object reachable from c. There are finitely many of these, as no infinite object can ever exist in a running program. More accurately stated, the objects of the final coalgebra represented by coalgebraic elements during program execution are all *regular* in the sense that they have a finite representation. These elements are first collected into a data structure (in our implementation, simply a list) and the right-hand sides of the equations are determined by the structure of the object using pattern matching. The object matches exactly one of the terms extracted in step 1.

6.4 A First Implementation

The examples of §6.2 show the need for new program constructs that would allow the user to manipulate corecursive types with the same ease and elegance as we are used to for algebraic datatypes. It is the goal of this section to provide language constructs that allow us to provide the intended semantics to the examples above in a functional language like OCaml.

The general idea behind the implementation is as follows. We want to keep the overhead for the programmer to a minimum. We want the programmer to specify the function in the usual way, then at runtime, when the function is evaluated on a given argument, a set of equations is generated and passed on to a solver, which will find a solution according to the specification. In an ideal situation, the programmer only has to specify the solver. For the examples where a CPO structure is present in the codomain, such as the free variables example, or when we have a complete metric space and a contractive map, we provide the typical solution methods (least and unique fixpoint) and the programmer only needs to tag the codomain with the intended solver. In other cases, the programmer needs to implement the solver.

6.4.1 Equations and Solvers

Our mock-up implementation aims to allow the programmer to encode a particular instantiation of the general diagram (5.1) as an OCaml module. This module can then be passed to an OCaml functor, `Corecursive`, that builds the desired function. We discuss the structure of `Corecursive` later in this section.

The functor F is represented by a parameterized type `'b f`. The structures (C, γ) and (A, α), which form a coalgebra and an algebra, respectively, for the functor F, are defined by types `coalgebra` and `algebra`, respectively. This allows

us to specify γ naturally as a function from `coalgebra` to `coalgebra f` and α as a function from `algebra f` to `algebra`. In the free variables example, if `VarSet` is a module implementing sets of strings, this is done as:

```
type 'b f = I1 of string | I2 of 'b * 'b | I3 of string * 'b
type coalgebra = Var of string
               | App of coalgebra * coalgebra
               | Lam of string * coalgebra
type algebra = VarSet.t

let gamma (c:coalgebra) : coalgebra f =
  match c with
    | Var v -> I1 v
    | App(c1, c2) -> I2(c1, c2)
    | Lam(x, c) -> I3(x, c)
let alpha (s:algebra f) : algebra =
  match s with
    | I1 v -> VarSet.singleton v
    | I2(s1, s2) -> VarSet.union s1 s2
    | I3(x, s) -> VarSet.remove x s
```

Variables are represented by strings and fresh variables are generated with a counter. Equations are of the form `variable = t`, where the variables on the left-hand side are elements of the domain and the terms on the right side are built up from the constructors of the datatype, constants and variables.

In the `fv` example, the domain was specified by the following datatype:

```
type term =
  | Var of string
```

```
| App of term * term
| Lam of string * term
```

Recall the four equations above defining the free variables of the λ-term (6.1) from the introduction:

$$\texttt{fv t = (fv x)} \cup \texttt{(fv u)} \quad \texttt{fv u = (fv y)} \cup \texttt{(fv t)} \quad \texttt{fv x = } \{\texttt{x}\} \quad \texttt{fv y = } \{\texttt{y}\}$$

A variable name is generated for each element of the coalgebra encountered. For example, here we write `v1` for the unknown corresponding to the value of `fv t`; `v2` for `x`; `v3` for `u`; and `v4` for `y`. An equation is represented as a pair of a variable and an element of type `f variable`. The intuitive meaning of a pair $(\texttt{v}, \texttt{w})$ is the equation $\texttt{v} = \alpha(\texttt{w})$. In the example above, we would have

`("v1", I2("v2", "v3"))`	representing	`v1 = v2` $\cup$ `v3`
`("v2", I1("x"))`	representing	`v2 =` $\{$`x`$\}$
`("v3", I2("v4", "v1"))`	representing	`v3 = v4` $\cup$ `v1`
`("v4", I1("y"))`	representing	`v4 =` $\{$`y`$\}$

The function `solve` can now be described. Its arguments are a variable `v` for which we want a solution and a system of equations in which `v` appears. It returns a value for `v` that satisfies the equations. In most cases the solution is not unique, and the `solve` method determines which solution is returned.

For technical reasons, two more functions need to be provided. The function `equal` provides an equality test on the coalgebra, which allows the equation generator to know when it has encountered a loop. In most cases, this equality is just the OCaml physical equality `==`; this is necessary because the OCaml equality `=` on coinductive objects does not terminate. In some other cases the function `equal` is an equality function built from both `=` and `==`. This is necessary when the argument is a pair that is destructed and rebuilt at each recursive call.

The function `fh` can be seen either as an iterator on the functor `f` in the style of folding and mapping on lists or as a monadic operator on the functor `f`. It allows the lifting of a function from `'c` (typically `coalgebra`) to `'a` (typically `algebra`) to a function from `'c f` to `'a f`, while folding on an element of type `'e`. It works by destructing the element of type `'c f` to get some number (perhaps zero) elements of type `'c`, successively applying the function on each of them while passing through the element of type `'e`, and reconstructing an element of type `'a f` with the same constructor used in `'c f`, returned with the final value of the element of type `'e`. In the example on free variables, the function `fh` is defined as:

```
let fh (h: 'c * 'e -> 'a * 'e) : 'c f * 'e -> 'a f * 'e = function
  | I1 v, e -> I1 v, e
  | I2(c1, c2), e -> let a1, e1 = h (c1, e) in
                     let a2, e2 = h (c2, e1) in
                     I2(a1, a2), e2
  | I3(x, c), e -> let a, e1 = h (c, e) in
                   I3(x, a), e1
```

If we had access to an abstract representation of the functor `f`, analyzing it allows to automatically generate the function `fh`. This is what we do in §6.5.

All this is summarized in the signature of a type `SOLVER`, used to specify one of those functions:

```
module type SOLVER = sig
  type 'b f
  type coalgebra
  type algebra

  val gamma : coalgebra -> coalgebra f
```

```
    val alpha : algebra f -> algebra

    type variable = string
    type equation = variable * (variable f)

    val solve : variable -> equation list -> algebra

    val equal : coalgebra -> coalgebra -> bool
    val fh : ('c * 'e -> 'a * 'e) -> 'c f * 'e -> 'a f * 'e
  end
```

Let us now define the OCaml functor `Corecursive`. From a specification of a function as a module `S` of type `SOLVER`, it generates the equations to be solved and sends them to `S.solve`. Here is how it generates the equations: starting from an element `c` of the coalgebra, it gathers all the elements of the coalgebra that are reachable from `c`, recursively descending with `gamma` and `fh`, and stopping when reaching an element that is equal—in the sense of the function `equal`—to an element that has already been seen. For each of those elements, it generates an associated fresh variable and an associated equation based on applying `gamma` to that element.

From an element `c`, generating the equations and solving them with `solve` returns an element `a` in the coalgebra, the result of applying the function we defined to `c`.

```
module Corecursive :
  functor (S: SOLVER) -> sig
    val main : S.coalgebra -> S.algebra
  end
```

We will now explain the default solvers we have implemented and which are available

for the programmer to use. These solvers cover the examples we have shown before: a least fixpoint solver, a solver that generates coinductive elements and is used for substitution, and a Gaussian elimination solver.

6.4.2 Least Fixpoints

If the algebra A is a CPO, then every monotone function f on A has a least fixpoint, by the Knaster–Tarski theorem. Moreover, if the CPO satisfies the *ascending chain condition* (ACC), that is, if there does not exist an infinite ascending chain, then this least fixpoint can be computed in finite time by iteration, starting from $\bot_A$. Even if the ACC is not satisfied, an approximate least fixpoint may suffice.

In the free variables example, the codomain $(\mathcal{P}(\mathsf{Var}), \subseteq)$ is a CPO, and its bottom element is $\bot_A = \varnothing$. It satisfies the ACC as long as we restrict ourselves to the total set of variables appearing in the term. This set is finite because the term is regular and thus has a finite representation.

To implement this, first consider the set of equations: each variable is defined by one equation relating it to the other variables. We keep a guess for each variable, initially set at $\bot_A$, and compute a next guess based on the equation for each variable. This eventually converges and we can return the value of the desired variable. Note that to implement this, the programmer needs to know that A is a CPO satisfying the ACC, and needs to provide two things: a bottom element $\bot_A$, and an equality relation on A that determines when a fixpoint is achieved.

The same technique can be used to implement the solver for the abstract interpretation example, as it is also a least fixpoint in a CPO. This CPO is the subset of the join semilattice of abstract domains containing only the elements greater than or equal to the initial abstract domain. The ACC is ensured by the fact that the

abstract domain is always of finite height. The bottom element is the initial abstract domain. Much of the code is shared with the free variables example. As pointed out before, only the bottom element of A and the equality on A change.

More suprisingly, this technique can also be used in the probability examples. Here the system of equations looks more like a linear system of equation on $\mathbb{R}$. Except in trivial extreme cases, the equations are contracting, thus we can solve them by iterative approximation until getting close enough to a fixpoint. The initial element $\perp_A$ is 0. The equality test on A is the interesting part: since it determines when to stop iterating, two elements of A are considered equal if and only if they differ by less than ε, the precision of the approximation. This is specified by the programmer in the definition of equality on A. Of course, such a linear system could also be solved with Gaussian elimination, as presented below in §6.4.4.

It can be seen from these examples that the least fixpoint solver is quite generic and works for a large class of problems. We need only parameterize with a bottom element to use as an initial guess and an equality test.

6.4.3 Generating Coinductive Elements and Substitution

Let us return to the substitution example. Suppose we wanted to replace y in Fig. 6.1(b) by the term of Fig. 6.1(a) to obtain Fig. 6.1(c). The extracted equations would be

```
v1 = App(v2, v3)
v2 = Var("x")
v3 = App(v4, v1)
v4 = App(Var "x", Var "x")
```

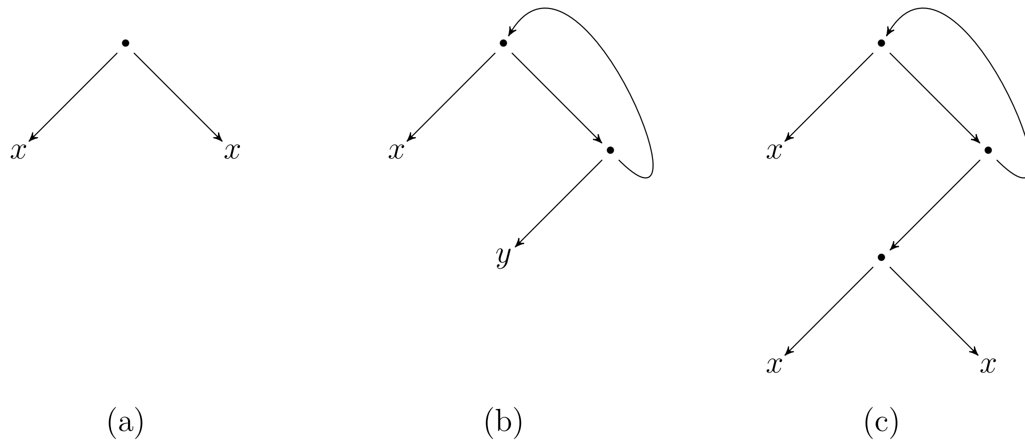


Figure 6.1: A substitution example.

and we are interested in the value of `v1`. Finding such a `v1` is easily done by executing the following code in OCaml:

```
let rec v1 = App(v2, v3)
    and v2 = Var("x")
    and v3 = App(v4, v1)
    and v4 = App(Var "x", Var "x")
in v1
```

This code can be easily generated (as a string of text) from the equations. Unfortunately, there is no direct way of generating the element that this code would produce. One workaround is to use the module `Toploop` of OCaml that provides the ability to dynamically execute code from a string, like `eval` in Javascript. But that is not a satisfying solution.

Another solution is to allow the program to manipulate terms by making all subterms mutable using references:

```
type term =
```

```
  | Var of string
  | App of term ref * term ref
  | Lam of string * term ref
```

This type allows the creation of the desired term by going down the equations and building the terms progressively, back-patching if necessary when encountering a loop. But this is also unsatisfactory, as we had to change the type of `term` to allow references.

The missing piece is mutable variables, which are currently not supported in the ML family of languages. A variable is mutable if it can be dynamically rebound, as with the Scheme `set!` feature or ordinary assignment in imperative languages. In ML, variables are only bound once when they are declared and cannot be rebound.

References can simulate mutable variables, but this corrupts the typing and forces the programmer to work at a lower pointer-based level. Moreover, there are subtle differences in the aliasing behavior of references and mutable variables. The language constructs we propose should ideally be created in a programming language with mutable variables.

6.4.4 Gaussian Elimination

In many of the examples on probabilities and streams, a set of linear equations is generated. One of the examples on probabilistic protocols of §6.2.2 requires us to find a float `var1` such that

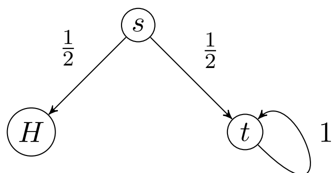
```
var1 = 0.5 + 0.5 * var2
var2 = 0.5 * var1
```

In the case where the equations are contractive, we have already seen that the solution is unique and we can approximate it by iteration. We have also implemented

a Gaussian elimination solver that can be used to get a more precise answer or when the map is not contractive but the solution is still unique.

But what happens when the linear system has no solution or an infinite number of solutions? If the system does not have a solution, then there is no fixpoint for the function, and the function is undefined on that input. If there are an infinite number of solutions, it depends on the application. For example, in the case of computing the probability of heads in a probabilistic protocol, we want the least such solution such that all variables take values between 0 and 1.

For example, let us consider the following probabilistic protocol: Flip a fair coin. If it comes up heads, output heads, otherwise flip again. Ignore the result and come back to this last state, effectively flipping again forever. This protocol can be represented by the following probabilistic automaton:



The probability of heads starting from s and t, respectively, is given by:

$$\mathsf{Pr}_H(s) = \frac{1}{2} + \frac{1}{2} \cdot \mathsf{Pr}_H(t) \qquad\qquad \mathsf{Pr}_H(t) = 1 \cdot \mathsf{Pr}_H(t).$$

The set of solutions for these equations for $\mathsf{Pr}_H(t)$ is the interval $[0, 1]$, thus the set of solutions for $\mathsf{Pr}_H(s)$ is the interval $[\frac{1}{2}, 1]$. The desired result, however, is the least of those solutions, namely 1/2 for $\mathsf{Pr}_H(s)$, because the protocol halts with result heads only with probability 1/2.

Again, the Gaussian solver is quite generic and would be applicable to a large class of problems involving linear equations.

6.5 Automatic Partitioning

In §6.4, we described a mock-up implementation that demonstrates the feasibility of our approach. In this implementation, the programmer needs to provide the elements of the `SOLVER` module. We now describe our ideas for future work, and in particular, ideas to make the task of the programmer easier by automatically generating some of those elements.

Providing all the elements to a `SOLVER` module requires from the programmer a good understanding of the concepts explained in this chapter and a method to solve equations. On the other hand, examples show that the same solving techniques arise again and again. Ideally, we would like the programmer to have to write only:

```
type term =                          let rec[...] fv = function
  | Var of string                      | Var v -> {v}
  | App of term * term                 | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam of string * term               | Lam (x,t) -> (fv t) − {x}
```

where the keyword `rec` has been parameterized by the name of a module implementing the `SOLVER` interface for a particular codomain, such as a generic iteration solver for CPOs or contractive maps or a Gaussian elimination solver for linear equations.

This definition is almost enough to generate the `SOLVER` module. Only three more things need to be specified by the programmer:

- the function `equal` on coalgebras, which is just `==` in most cases; and
- the two elements needed in the least fixpoint algorithm: a bottom element $\bot_A$ and an equality test $=_A$ on the algebra A, written `algebra` in the code.

The other elements can be directly computed from a careful analysis of the function definition:

- The function can be typed with the usual typing rules for recursive functions. Then `algebra` is defined as its input type and `coalgebra` as its output type.

- An analysis of the abstract syntax trees of the clauses of the function definition can determine what is executed before the recursive calls, which comprises γ, and what is executed after the recursive calls, which comprises α. An analysis of the arguments that are passed to the recursive calls, as well as the variables that are still alive across the boundary between `gamma` and `alpha`, determine the functor `f`.

- The function `fh` can be defined by induction on the structure of the abstract syntax tree defining `'a f`. The only difficult case is the product, where we apply `h` to every element of type `'a` in the product, passing through the element of type `'e`, and returning a reconstructed product of the results.

- The type `equation` is always defined in the same way.

- Finally, the `solve` function is generic for all functions solved as a least fixpoint by iteration, just depending on the bottom element and the equality on the algebra.

6.6 Conclusion

Coalgebraic (coinductive) datatypes and algebraic (inductive) datatypes are similar in many ways. Nevertheless, there are some important distinctions. Algebraic types have a long history, are very well known, and are heavily used in modern applications, especially in the ML family of languages. Coalgebraic types, on the other hand, are the subject of more recent research and are less well known. Not all

modern languages support coalgebraic types—for example, Standard ML and F# do not—and even those that do may not do so adequately.

The most important distinction is that coalgebraic objects can be cyclic, whereas algebraic objects are always well-founded. Functions defined by structural recursion on well-founded data always terminate and yield a value under the standard semantics of recursion, but not so on coalgebraic data. A more subtle distinction is that constructors can be interpreted as functions under the algebraic interpretation, as they are in Standard ML, but not under the coalgebraic interpretation as in OCaml.

Despite these differences, there are some strong similarities. They are defined in the same way by recursive type equations, algebraic types as initial solutions and coalgebraic types as final solutions. Because of this similarity, we would like to program with them in the same way, using constructors and destructors and writing recursive definitions using pattern matching.

In this chapter we have shown through several examples that this approach to computing with coalgebraic types is not only useful but viable. For this to be possible, it is necessary to circumvent the standard semantics of recursion, and we have demonstrated that this obstacle is not insurmountable. We have proposed new programming language features that would allow the specification of alternative solutions and methods to compute them, and we have given mock-up implementations that demonstrate that this approach is feasible.

The chief features of our approach are the interpretation of a recursive function definition as a scheme for the specification of equations, a means for extracting a finite such system from the function definition and its (cyclic) argument, a means for specifying an equation solver, and an interface between the two. In many cases, such as an iterative fixpoint on a codomain satisfying the ascending chain condition,

the process can be largely automated, requiring little extra work on the part of the programmer.

We have mentioned that mutable variables are essential for manipulating coalgebraic data. Current functional languages in the ML family do not support mutable variables; thus true coalgebraic data can only be constructed explicitly using **let rec**, not programmatically. Moreover, once constructed, a coalgebraic object cannot be changed dynamically. These restrictions currently constitute a severe restriction the use of coalgebraic datatypes. One workaround is to simulate mutable variables with references, but this is a grossly unsatisfactory alternative, because it confounds algebraic elegance and forces the programmer to work at a lower pointer-based level. In the next chapter we present CoCaml, a smoother and more realistic implementation of these ideas in an ML-like language with mutable variables.

Chapter 7

CoCaml: Functional Programming with Regular Coinductive Types

We present CoCaml, a functional programming language extending OCaml, which allows us to define functions on coinductive datatypes parameterized by an equation solver. We provide numerous examples that attest to the usefulness of the new programming constructs, including operations on infinite lists, infinitary λ-terms and p-adic numbers.

In CoCaml, functions defined by equations, like the ones presented in chapter 6, can be supplied with an extra parameter, namely a solver for the given equations. For instance, the example `fv` of §6.1 would be almost the same in CoCaml:

```
let corec[iterator ∅] fv = function
   | Var v -> {v}
   | App (t1,t2) -> (fv t1) ∪ (fv t2)
   | Lam (x,t) -> (fv t) − {x}
```

The construct `corec` with the parameter `iterator ∅` specifies to the compiler that the equations generated as in §6.3 should be solved using an iterator—in this case

a least fixpoint computation—starting with the initial element $\varnothing$.

7.1 Preliminaries

In this section, we present the basics of coinductive types and the theoretical foundations on well-definedness of functions on coinductive types, which we will use to define the new language constructs. We also describe *capsule semantics*, a heap-free mathematical semantics for higher order functional and imperative programs, on which our implementation is based.

7.1.1 ML with Coalgebraic Datatypes

Coalgebraic (coinductive) datatypes are very much like algebraic (inductive) datatypes in that they are defined by recursive type equations. The set of algebraic objects form the least (initial) solution of these equations and the set of coalgebraic objects the greatest (final) solution.

Algebraic types have a long history going back to the initial algebra semantics of Goguen and Thatcher [GT74]. They are very well known and are heavily used in modern applications, especially in the ML family of languages. Coalgebraic types, on the other hand, are the subject of more recent research and are less well known. Not all modern functional languages support them—for example, Standard ML and F# do not—and even those that do support them do not do so adequately.

The most important distinction is that coalgebraic objects can have infinite paths, whereas algebraic objects are always well-founded. *Regular* coalgebraic objects are those with finite (but possibly cyclic) representations. We would like to define recursive functions on coalgebraic objects in the same way that we define

recursive functions on algebraic data objects, by structural recursion. However, whereas functions so defined on well-founded data always terminate and yield a value under the standard semantics of recursion, this is not so with coalgebraic data because of the circularities.

In Standard ML, constructors are interpreted as functions, and thus coinductive objects cannot be formed. Whereas in OCaml, coinductive objects can be defined, and constructors are not functions. Formally, in call-by-value languages, constructors can be interpreted as functions under the algebraic interpretation, as they are in Standard ML, but not under the coalgebraic interpretation as in OCaml. In Standard ML, a constructor is a function:

```
- SOME;
val it = fn : 'a -> 'a option
```

Since it is call-by-value, its arguments are evaluated, which precludes the formation of coinductive objects. In OCaml, a constructor is not a function. To use it as a function, one must wrap it in a lambda:

```
> Some;;
Error: The constructor Some expects 1 argument(s),
       but is applied here to 0 argument(s)
> fun x -> Some x;;
- : 'a -> 'a option = <fun>
```

This allows the formation of coinductive objects:

```
> type t = C of t;;
type t = C of t
> let rec x = C x;;
```

```
val x : t = C (C (C (C (C (C (C (C ...)))))))
```

Despite these differences, inductive and coinductive data share some strong similarities. We have mentioned that they satisfy the same recursive type equations. Because of this, we would like to define functions on them in the same way, using constructors and destructors and writing recursive definitions using pattern matching. However, to do this, it is necessary to circumvent the standard semantics of recursion, which does not necessarily halt on cyclic objects.

For full functionality in working with coalgebraic data, *mutable variables* are essential. Current functional languages in the ML family do not support mutable variables; thus true coalgebraic data can only be constructed explicitly using `let rec`, provided we already know what they look like at compile time. Once constructed, they cannot be changed, and they cannot be created programmatically. This constitutes a severe restriction on the use of coalgebraic datatypes. One workaround is to simulate mutable variables with references, but this is ugly; it corrupts the algebraic typing and forces the programmer to work at a lower pointer-based level. Capsules, which we describe next, offer the right abstraction to avoid the use of reference, making construction and manipulation of coalgebraic data easy.

7.1.2 Capsule Semantics

Our implementation is based on *capsule semantics*, described in chapter 2. In capsule semantics, regular coinductive types and recursive functions are defined in the same way. There is a special uninitialized value `<>` for each type. The capsule evaluation rules consider a variable to be irreducible if it is bound to this value. The variable can be used in computations as long as there is no attempt to deconstruct it; any such attempt results in a runtime error. "Deconstruction" here means different things

for different types. For a coinductive type, it means applying a destructor. For `int`, it would mean attempting to perform arithmetic with it. But it can be used as the argument of a constructor or can appear on the left-hand side of an assignment without error, as these do not require deconstruction. This allows coalgebraic values and recursive functions to be created in a uniform way via back-patching (a.k.a. Landin's knot). Thus, `let rec x = d in e` is syntactic sugar for

```
let x = <> in (x := d);e
```

which in turn is syntactic sugar for

```
(fun x -> (x := d);e) <>
```

For example, `let rec x = (x,x) in snd (snd x)` becomes

```
let x = <> in (x := (x,x)); snd (snd x)
```

During the evaluation of `(x,x)`, the variable `x` is bound to `<>`, so `x` is not reduced. [1] The value of the expression is just `(x,x)`. Now the assignment `x := (x,x)` is performed, and `x` is rebound to the expression `(x,x)` in the environment. We have created an infinite coinductive object, namely an infinite complete binary tree. Evaluating `snd (snd x)` results in the value `(x,x)`.

Note that we never need to use placeholders or substitution to create cycles, as we are using the binding of `x` in the environment for this purpose. This is a major advantage over previous approaches [HLW03, ST98, Sym06, yW11]. Once `x` is rebound to a non-`<>` value, it can be deconstructed after looking it up in the environment.

The variable `x` also gives a handle into the data structure that allows it to be manipulated dynamically. For example, here is a program that creates a cyclic

[1] Actually, this is not quite true—a fresh variable is substituted for x by α-conversion first. But we ignore this step to simplify the explanation.

object of length 3, then extends it to length 4:

```
> let rec x =  1 :: 2 :: 3 :: x;;
val x : int list = [1; 2; 3; 1; 2; 3; ...]
> let y = x in x := 0 :: y; x;;
- : int list = [0; 1; 2; 3; 0; 1; 2; 3; ...]
```

Any cycle must always contain at least one such variable. Note that these two cyclic data objects actually represent the same infinite object, namely the infinite term `C(C(C(...` . Two elements of a coalgebraic type are considered equal iff they are bisimilar (see §7.4.3). For this reason, coalgebraic types are not really the same as the circular data structures as studied in [ST98, BZ02, HLW03, Sym06]. The example above shows that we now allow run time definition of regular coinductive lists, whereas OCaml only allows infinite lists that are defined statically.

A downside to this approach is that the presence of the value `<>` requires a runtime check on value lookup. This is a sacrifice we have made to accommodate functional and imperative programming styles in a common framework, which is one of the main motivating factors behind capsules. We introduced capsule semantics in chapter 2; for a full account of capsule semantics in the presence of coalgebraic types, see [Koz12].

7.2 Equations and Solvers

When the programmer makes a function call $f(a_0)$, where f was defined using the `corec` keyword, execution happens in three distinct steps:

- a set of *equations* is generated;
- the equations are sent to a solver; the solver can be built-in or user-defined;

- the result of running the solver on the set of equations is returned as the result of function call $f(a)$.

In this section we describe in detail how equations are generated, and the different possible choices for the solver.

7.2.1 Equation Generation

An *equation* denotes an equality between two terms. Its left-hand-side is a variable x_i that stands for the call of f on some input a_i. Its right-hand-side is a partially evaluated abstract syntax tree: it is an expression of the language which can contain other variables x_j.

When calling a recursive function f on an inductive (well-founded) term a_0, this function can make recursive calls, generating new calls to function f. The reason this computation finishes is because the computation is well-founded: every path in the call tree reaches a base case.

Similarly, if the function f was defined with the `corec` keyword, its call on a coinductive term a_0 might involve some recursive calls; those recursive calls might themselves involve some recursive calls, and so on. This time the computation is not well-founded, but because a_0 has a finite representation, the set of possible such calls is finite, for example recursive calls were made on $a_1, \ldots, a_n$.

While executing those recursive calls, an variable x_i is generated for each a_i, and the call to $f(a_i)$ is partially evaluated to generate an equation, replacing the calls to $f(a_j)$ by their corresponding x_j. We thus generate a set of equations whose solution is the value of $f(a_0)$. Of course, the arguments $a_0, \ldots, a_n$ are not known in advance, so the x_i have to be generated while the program is exploring the recursive calls. This is achieved by keeping track of all the a_i that have been seen so far, along with

their associated unknowns x_i.

A *solver* takes a set of equations and returns a solution, or fails. We currently have four built-in solvers implemented, four of which are quite versatile and can be used in many different applications. We also give to the programmer the ability to define its own solvers.

7.2.2 The `iterator` Solver

In many cases the set of equations can be seen as defining a fixpoint of a monotone function. For example, when the codomain is a CPO, and the operations on the right-hand sides of the equations are monotone, then the Knaster–Tarski theorem ensures that there is a least fixpoint. Moreoever, if the CPO is finite or otherwise satisfies the ascending chain condition (ACC), then the least fixpoint can be computed in finite time by iteration, starting from the bottom element of the CPO.

The `iterator` solver takes an argument b representing the initial guess for each unknown. In the case of a CPO, this would typically be the bottom element.

Internally, a guess is made for each unknown, initially b. At each iteration, a new guess is computed for each unknown by evaluating the corresponding right-hand side, where the unknowns have been replaced by current guesses. When all the new guesses equal the old guesses, we stop, as we have reached a fixpoint. The right-hand sides are evaluated in postfix order, i.e., in the reverse order of seeing and generating new equations, because it usually makes the iteration converge faster.

Note that this `iterator` solver is closely related to the least fixpoint solver described in [Pot], but it can also be used in applications where the desired fixpoint is not necessarily the least.

Example. We revisit the example from the introduction by applying this solver to create a function `set` that computes the set of all elements appearing in a list. A regular list, even if it is infinite, has only finitely many elements. If A is the type of the elements, the codomain of `set` is the CPO $(\mathcal{P}(A), \subseteq)$ with bottom element $\varnothing$. Restricted to subsets of the set of variables appearing in the list, it satisfies the ascending chain condition, which ensures that the least fixed point can be computed in finite time by iteration.

For the implementation, we represent a set as an ordered list. The function `insert` inserts an element into an ordered list without duplicating it if it is already there. The function `set` can be defined as:

```
let corec[iterator []] set l = match l with
  | [] -> []
  | h :: t -> insert h (set t)
```

Internally, a guess is made for each unknown, initially `[]`. At each iteration, a new guess for each set is computed for each unknown by evaluating the corresponding right-hand side. When all the new guesses equal the old guesses, we stop, as we have reached a fixpoint, the intended result. The complexity of this solver depends on the number of iterations; at each iteration every equation is evaluated, which leads to a complexity on the order of the product of the number of iterations by the number of equations.

7.2.3 The `constructor` Solver

The `constructor` solver can be used when a function tries to build a data structure that could be cyclic, representing a regular coinductive element. Internally, `constructor` first checks that the right-hand side of every equation is a value (an

integer, float, string, Boolean, unit, tuple on values or unknowns, injection on a value or unknown). Then it replaces the unknown variables on the right-hand sides with normal variables and adds them to the environment, thus creating the capsule representing the desired data structure. Its complexity is linear in the number of equations.

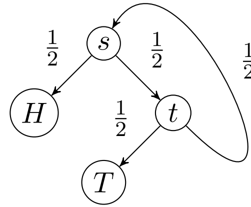
Example. The `map` function on lists takes a function `f` and a list `l`, applies `f` on every element `h` of `l`, and returns the list of the results `f h`. The `constructor` solver can be used to create a `map` functions that works on all lists, finite or infinite.

```
let corec[constructor] map arg = match arg with
  | f, [] -> []
  | f, h :: t -> f(h) :: map (f,t)
```

7.2.4 The `gaussian` Solver

The `gaussian` solver is designed to be used when the function computes a linear combination of recursive calls. The set of equations is then a Gaussian system that can be solved by standard techniques.

Example. Imagine one wants to simulate a biased coin, say a coin with probability 2/3 of heads, with a fair coin. Here is a possible solution: flip the fair coin. If it comes up heads, output heads, otherwise flip again. If the second flip is tails, output tails, otherwise repeat from the start. This protocol can be represented succinctly by the following probabilistic automaton:



Operationally, starting from states s and t, the protocol generates series that converge to 2/3 and 1/3, respectively.

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{8} + \tfrac{1}{32} + \tfrac{1}{128} + \cdots = \tfrac{2}{3}$$
$$\mathsf{Pr}_H(t) = \tfrac{1}{4} + \tfrac{1}{16} + \tfrac{1}{64} + \tfrac{1}{256} + \cdots = \tfrac{1}{3}.$$

However, these values can also be seen to satisfy a pair of mutually recursive equations:

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{Pr}_H(t) \qquad\qquad \mathsf{Pr}_H(t) = \tfrac{1}{2} \cdot \mathsf{Pr}_H(s).$$

In CoCaml, we can model the automaton by a coinductive type and define a function computing the probability of Heads using the `gaussian` solver:

```
type tree = Heads | Tails
          | Flip of float * tree * tree

let corec[gaussian] probability t = match t with
    Heads -> 1.
  | Tails -> 0.
  | Flip(p, t1, t2) -> p *. probability t1 +.
                         (1. -. p) *. probability t2
```

7.2.5 The separate Solver

In both OCaml and CoCaml, the default printer for lists prints up to some preset depth, printing "..." when this depth is exceeded. This will always happen if the list is circular.

```
let rec ones = 1 :: ones;;
val ones : int list = [1; 1; 1; 1; 1; 1; 1; ...]
```

This is not very satisfying. Often it may appear as if some pattern is repeating, but what if for instance a 2 appears in 50th position and is not printed? A better solution might be to print the non-repeating part normally, followed by the repeating part in parenthesis. For example, the list `[1; 2]` might be printed `12` and the list `1 :: 2 :: ones` be printed `12(1)`. This can be achieved by creating a special solver `separate`, which from the equations defining the lists outputs two finite lists, the non-repeating part and the repeating part. From there it is easy to finish.

Internally, the equations given to the solver are a graph representing the list. A simple cycle-detection algorithm allows us to solve the equations as desired. Its complexity is linear in the number of equations.

However, this example is not completely satisfying. In fact, the solver is quite ad hoc, which contrasts greatly with the solvers we have seen so far. Moreover, the type `sep` we have introduced exists merely to make the type checker happy. Conceptually, the solver takes a list as an argument and returns a pair of lists. This example shows the limits of the typing mechanism as applied to functions on coinductive data.

7.2.6 User-defined Solvers

The solvers we have presented so far are implemented directly in the interpreter. However, as versatile as these solvers are, the programmer sometimes needs to define his/her own solver. This can be done by defining a module of type `Solver`.

```
module type Solver = sig
  type var
  type expr
  type t
  val fresh : unit -> var
  val unk : var -> expr
  val solve : var -> (var * expr) list -> t
end
```

Type `var` is the type of the variables in the equations, and also the type of the left-hand sides of the equations; type `expr` is the type of the right-hand sides of the equations; and type `t` is the return type of the solver, and thus also of the function that is being defined.

Function `fresh` generates fresh elements of type `var`, it is called on each element of the coalgebra that is encountered; it is the responsibility of the user to provide a function that generates elements of type `var` that are all different. In most cases type `var` is simply `string`, and fresh strings can easily be generates, for example with the function:

```
let fresh = let c = 0 in
 (fun (x:unit) -> c := c+1;
                 "fresh" @ (string_of_int c))
```

But the programmer could choose a different type `var`, for instance to store more information in it.

To represent variables on the right-hand sides of equations, we need to be able to inject an element of type `var` into the type `expr`. Function `unk` provides this. Typically `expr` is a sum type that contains a special case `Unknown of var`, and the injection is just `fun x -> Unknown x`.

Finally, `solve` is the solver itself. By construction, an equation always has a variable on its left-hand side, and an `expr` on its right-hand side, and is thus represented as a pair of type `var * expr`. Given an element x of type `var`, and a set (represented as a list) of equations, it returns an element of type `t` that is a solution for the variable x satisfying those equations.

Example. We could define the gaussian solver, as a user-defined solver `Gaussian` by taking `var = string`, `t = float`, and

```
type expr =
  Val of float
| Plus of expr * expr
| Minus of expr * expr
| Mul of expr * expr
| Unknown of var
```

`fresh` and `unk` are the typical ones shown above, and `solve` implements a gaussian-elimination algorithm in CoCaml. In the definition of the type `expr`, we could have chosen to have `Mul` with arguments `float*expr` which would automatically keep the equations linear. We write instead `Mul of expr*expr` which is more general, and we check linearity dynamically. The declaration of function `to_floati`

on p-adic integers becomes

```
let corec[Gaussian] probability t = match t with
    Heads -> Val 1.
  | Tails -> Val 0.
  | Flip(p, t1, t2) ->
  Plus(Mul(Val p, probability t1),
       Mul(Val (1.-. p), probability t2))
```

The right-hand side is slightly different and less clear than in the original definition. In some sense, instead of working with the abstract syntax tree of the whole language, the programmer is able to define his/her own small abstract syntax tree to work with, representing right-hand sides of functions. This is reminiscent of a known technique to solve corecursive equations by defining a coalgebra whose carrier is a set of expressions comprising the intermediate steps of the unfolding of the equations [SR07, SR10, Cap11].

7.3 Examples

In this section, we show several examples of functions on coinductive types, including finite and infinite lists, a library for p-adic numbers, and infinitary λ-terms.

7.3.1 Finite and Infinite Lists

We present an application of our main solvers through examples on lists, one of the simplest examples of coinductive types. Through these examples, we show how easy it is to create recursive functions on regular coinductive datatypes, as the process is very close to creating recursive functions on inductive datatypes.

Test of Finiteness

We would like to be able to test whether a list is finite or infinite. The most intuitive way of doing this is to write a function like:

```
let rec is_finite l = match l with
  | [] -> true
  | h :: t -> is_finite t
```

Of course, this does not terminate on infinite lists under the standard semantics of recursive functions. However, if we use the `corec` keyword, the equations generated for `[0]` will look like

```
is_finite [0] = is_finite []
is_finite [] = true
```

and the result will be `true`. For the infinite list `ones`, the only equation will look like

```
is_finite(ones) = is_finite(ones)
```

and we expect the result to be `false`. Intuitively, the result of solving the equations should be `true` if and only if the expression `true` appears on the right-hand side of one of the equations. This can be achieved with the iterator solver, using as first guess the value we should observe if it is not finite, here `false`:

```
let corec[iterator false] is_finite = function
  | [] -> true
  | h :: t -> is_finite t
```

List `exists`

Given a Boolean-valued function `f` that tests a property of elements of a list `l`, we would like to define a function that tests whether this property is satisfied by at least one element of `l`. The function can simply be programmed using the `iterator` solver, where the default value should be `false`:

```
let corec[iterator false] exists arg =
  match arg with
     | f, [] -> false
     | f, h :: t -> f(h) || exists (f, t)
```

Note that for this function to work, it is critical that the "or" operator `||` be lazy, so that the partial evaluation of the expression `f(h) || exists (f, t)` can return `true` directly whenever `f(h)`, even if the result of evaluating `exists(f, t)` is not known.

The Curious Case of Filtering

Given a Boolean-valued function function `f` and a list `l`, we would like to define a function that creates a new list `l1` by keeping only the elements of `l` that satisfy `f`. The first approach is to use the `constructor` solver and do it as if the list were always finite:

```
let corec[constructor] filter_naive arg =
  match arg with
    | f, [] -> []
    | f, h :: t ->
       if f(h) then h :: filter_naive(f, t)
```

```
              else filter_naive(f, t)
```

However, this does not quite work. For example, if called on the function `fun x -> x <= 0` and the list `ones`, it generates only one equation

```
filter_naive(ones) = filter_naive(ones)
```

and it is not clear which solution is desired by the programmer. However, it is clear that in this particular case, the set `[]` should be returned. The problem arises whenever the function is called on an infinite list `l` such that no element of `l` satisfies `f`. Rather than modify the solver, our solution is to be a little bit more careful and return `[]` explicitly when needed:

```
let corec[constructor] filter arg = match arg with
  | f, [] -> []
  | f, h :: t: ->
                if f(h) then h :: filter(f, t)
                else if exists(f, t) then filter(f, t)
                                     else []
```

The main problem with this solution is that it has a quadratic complexity. It is possible to get to a linear complexity using an ad hoc solver, that we won't developed here.

Other Examples on Lists

We have presented a few examples of functions on infinite lists. Some of them are inspired by classic functions on lists supported by the List module of OCaml. Some functions of the List module, like sorting, do not make sense on infinite lists. But most other functions of the List module can be implemented in similar ways. We

refer to the implementation provided as attachment for more details.

7.3.2 A Library for p-adic Numbers

In this section we present a library for p-adic numbers and operations on them.

The p-adic Numbers

The p-adic numbers [Bak11, Wik12] are a well-studied mathematical structure with applications in several areas of mathematics. For a fixed prime p, the p-adic numbers $\mathbb{Q}_p$ form a field that is the completion of the rationals under the p-adic metric in the same sense that the reals are the completion of the rationals under the usual Euclidean metric. The p-adic metric is defined as follows. Define $|\cdot|_p$ by

- $|0|_p = 0$;
- if $x \in \mathbb{Q}$, write x as $x = ap^n/b$, where n, a and b are integers and neither a nor b is divisible by p. Then $|x|_p = p^{-n}$.

The distance between x and y in the p-adic metric is $|x - y|_p$. Intuitively, x and y are close if their difference is divisible by a high power of p.

Just as a real number has a decimal representation with a finite number of nonzero digits to the left of the decimal point and a potentially infinite number of nonzero digits to the right, a p-adic number has a representation in base p with a finite number of p-ary digits to the right and a potentially infinite number of digits to the left. Formally, every element of $\mathbb{Q}_p$ can be written in the form $\sum_{i=k}^{\infty} d_i p^i$, where the d_i are integers such that $0 \le d_i < p$ and k is an integer, possibly negative. An important fact is that this representation is unique (up to leading zeros), in contrast to the decimal representation, in which $1 = 0.999\dots$. If $d_k = 0$ for $k < 0$, then the

number is said to be a *p-adic integer*. If b is not divisible by p, then the rational number a/b is a p-adic integer. Finally, p-adic numbers for which the sequence $(d_k)_k$ is regular (ultimately periodic) are exactly the rational numbers. This is similar to the decimal representations of real numbers. Since our lists must be regular so that they can be represented in finite memory, these are the numbers we are interested in. We fix the prime p (written `p` in programs) once and for all, for instance as a global variable.

Equality and Normalization

We represent a p-adic number $x = \sum_{i=k}^{\infty} d_i p^i$ as a pair of lists:

- the list $d_0, d_1, d_2, \ldots$ in that order, which we call the *integer part* of x and which can be finite or infinite; and
- if $k < 0$ and $d_k \neq 0$, the list containing $d_{-1}, d_{-2}, \ldots, d_k$, which we call the *fractional part* of x and which is always finite.

Since the representation $x = \sum_{i=k}^{\infty} d_i p^i$ is unique up to leading zeros, the only thing we have to worry about when comparing two p-adic integers is that an empty list is the same as a list of zeros, finite or infinite. The following function `equali` uses the `iterator` solver and compares two integer parts of p-adic numbers for equality:

```
let corec[iterator true] equali x =
  match x with
    | [], [] -> true
    | h1 :: t1, h2 :: t2 ->
         h1 = h2 && equali_aux (t1, t2)
    | 0 :: t1, [] -> equali_aux (t1, [])
```

```
      | [], 0 :: t2 -> equali_aux (t2, [])
      | _ -> false
```

Interestingly, comparing the fractional parts is almost the same code, with the `rec` keyword instead of the `corec` keyword, and no auxiliary function.

```
let rec equalf x = match x with
  | [], [] -> true
  | h1 :: t1, h2 :: t2 ->
       h1 = h2 && equalf (t1, t2)
  | 0 :: t1, [] -> equalf (t1, [])
  | [], 0 :: t2 -> equalf (t2, [])
  | _ -> false


let equal x1 x2 = match x1, x2 with
  (i1, j1), (i2, j2) ->
      equali (i1, i2) && equalf (j1, j2)
```

This happens quite often: if one knows how to do something with inductive types, the solution for coinductive types often involves only changing the `rec` keyword to `corec` and some other minor adjustments. However, one must take care, as there are exceptions to this rule. In this example, since here `equali` also works on inductive types, we could have used `equali` instead of `equalf` in `equal`.

Now that we have equality, normalization of a p-adic integer becomes easy using the `constructor` solver:

```
let corec[constructor] normalizei i =
  if equali(i, []) then []
```

```
    else match i with i :: t -> i :: normalizei t
```

The function `normalizei` only requires equality with zero (represented as `[]`), which is much easier than general equality. We can now write a normalization on the fractional parts as a simple recursive function (once again, with the same code), or just use `normalizei`, which also works on the fractional parts.

Conversion from a Rational

We wish to convert a given rational a/b with $a, b \in \mathbb{Z}$ to its p-adic representation. Let us first try to convert $x = a/b$ into a p-adic integer if b is not divisible by p. Since x is a p-adic integer, we know that x can be written $x = \sum_{i=0}^{\infty} d_i p^i$, thus multiplying both sides by b gives

$$a = b \sum_{i=0}^{\infty} d_i p^i.$$

Taking both sides modulo p, we get $a = bd_0 \bmod p$. Since b and p are relatively prime, this uniquely determines d_0 such that $0 \leq d_0 < p$, which can be found by the Euclidean algorithm. We can now substract bd_0 to get

$$a - bd_0 = b \sum_{i=1}^{\infty} d_i p^i.$$

This can be divided by p by definition of d_0, which leads to the same kind of problem recursively.

This procedure defines an algorithm to find the digits of a p-adic integer. Since we know it will be cyclic, we can use the `constructor` solver:

```
let corec[constructor] from_rationali (a,b) =
  if a = 0 then []
  else let d = euclid p a b in
  d :: from_rationali ((a - b*d)/p, b)
```

where the call `euclid p a b` is a recursive implentation of a (slightly modified) Euclidean algorithm for finding d_0 as above.

If b is divisible by p, it can be written $p^n b_0$ where b_0 is not divisible by p, and we can first find the representation of a/b_0 as an integer, then shift by n digits to simulate division by p^n.

Conversion to a Float

Given a p-adic integer $x = \sum_{i=0}^{\infty} d_i p^i$, define $x_k = \sum_{i=0}^{\infty} d_{k+i} p^i$. Then for all $k \geq 0$, $x_k = d_k + p x_{k+1}$. If the sequence $(d_k)_k$ is regular, so is the sequence $(x_k)_k$, thus there exist $n, m > 0$ such that $x_{k+m} = x_k$ for all $k \geq n$. It follows that

$$x = x_0 = \sum_{i=0}^{n-1} d_i p^i + p^n x_n \qquad\qquad x_n = \sum_{i=0}^{m-1} d_{n+i} p^i + p^m x_n,$$

and further calculation reveals that $x = a/b$, where

$$a = \sum_{i=0}^{n+m-1} d_i p^i - \sum_{i=0}^{n-1} d_i p^{m+i} \qquad\qquad b = 1 - p^m.$$

But even without knowing m and n, the programmer can write a function that will automatically construct a system of $m + n$ linear equations $x_k = d_k + p x_{k+1}$ in the unknowns $x_0, \ldots, x_{m+n-1}$ and solve them by Gaussian elimination to obtain the desired rational representation. To accomplish this, we can just use our `gaussian` solver:

```
let corec[gaussian] to_floati i = match i with
  | [] -> 0.
  | d :: t -> (float_of_int d) +.
                (float_of_int p) *. (to_floati t)
```

This function returns the floating point representation of a given p-adic integer. It is interesting to note that, apart from the mention of `corec[gaussian]`, this is

exactly the function we would have written to calculate the floating-point value of an integer written in p-ary notation using Horner's rule.

A similar program can be used to convert the floating part of a p-adic number to a float. Adding the two parts gives the desired result.

Addition

Adding two p-adic integers is suprisingly easy. We can use (a slight adaptation of) the primary school algorithm of adding digit by digit and using carries. A carry might come from adding the floating parts, so the algorithm really takes three arguments, the two p-adic integers to add and a carry. Using the `constructor` solver, this gives:

```
let corec[constructor] addi arg = match arg with
  | [], [], c ->
      if c = 0 then []
      else (c mod p) :: addi ([], [], c/p)
  | h :: t, [], c ->
       addi (h :: t, [0], c)
  | [], h :: t, c ->
       addi ([0], h :: t, c)
  | hi :: ti, hj :: tj, c ->
      let res = hi + hj + c in
      (res mod p) :: addi (ti, tj, res / p)
```

Once again, once we have addition on p-adic integers, it is easy to program addition on general p-adic numbers.

Multiplication and Division

The primary school algorithm and the `constructor` solver can also be used for multiplication. However, we need to proceed in two steps. We first create a function `mult1` that takes a p-adic integer `i`, a digit `j`, and a carry `c`, and calculates `i*j+c`. We then create a function `multi` that takes two p-adic integers `i` and `j` and a carry `c` and calculates `i*j+c`.

```
let corec[constructor] mult1 arg = match arg with
 | [], d, c -> if c = 0 then []
               else (c mod p) :: mult1 ([], d, c/p)
 | hi :: ti, d, c ->
        let res = hi * d + c in
        (res mod p) :: mult1 (ti, d, res / p)


let corec[constructor] multi arg = match arg with
     | n1, [], c -> c
     | n1, h2 :: t2, c ->
      (match (addi (mult1 (n1, h2, 0), c, 0)) with
          | [] -> 0 :: multi (n1, t2, 0)
          | hr :: tr -> hr :: multi (n1, t2, tr) )
```

To extend this to general p-adic numbers, we can multiply both `i` and `j` by suitable powers of p before applying `multi`, then divide the result back as necessary.

Division of p-adic integers can be done with only one function using a `constructor` solver in much the same way as addition or multiplication. The algorithm also uses the `euclid` function and is closely related to `from_rational`.

Some of the examples above on p-adic numbers could be done with lazy evaluation, namely the arithmetic operations and conversion to a rational. However equality, normalization, conversion to a float and printing (showing the cycle) could not.

7.3.3 Equality

Now that we have recursive functions on coinductive types, we might ask whether it would be possible to program equality on a coinductive type. The answer is yes. The function is built in much the same way as the `equali` function, with the `iterator true` solver and an auxiliary function. This is a general trend for coinductive equality: two elements are equal unless there is evidence that they are unequal.

The code can be found below, with the small simplification of having expressions on pairs instead of general tuples.

```
type expr =
    | Var of string
    | Int of i
    | Inj of string * expr
    | Pair of expr * expr

let corec[iterator true] equal arg =
  match arg with
  | (Var x1, env1), (Var x2, env2) ->
        equal_aux ((assoc x1 env1, env1),
                   (assoc x2 env2, env2))
```

```
| (Var x1, env1), s2 ->
      equal_aux ((assoc x1 env1, env1), s2)
| s1, (Var x2, env2) ->
      equal_aux (s1, (assoc x2 env2, env2))
| (Int i1, env1), (Int i2, env2) -> i1 = i2
| (Inj (inj1, e3), env1), (Inj (inj2, e4), env2)
    ->  inj1 = inj2 &&
       equal_aux ((e3, env1), (e4, env2))
| (Pair(e1, e2), env1), (Pair(e3, e4), env2) ->
      equal_aux ((e1, env1), (e3, env1)) &&
      equal_aux ((e1, env1), (e3, env1))
| _ -> failwith "type␣error"
```

7.4 Implementation

We have implemented an interpreter for CoCaml. The implementation is about 5000 lines of OCaml. We have also used CoCaml on a number of examples, of which we have presented a few here. So far we have written about 1500 lines of CoCaml. The implementation is provided in the supplementary material.

7.4.1 Overview

In this section, we explain how the translation of the construct `corec[solver]` is implemented. Briefly, the body of the function is partially evaluated, replacing recursive calls by variables, and this forms the right-hand side of an equation. The generated equations are then solved using the parameter `solver`.

Note that equations can only be correctly generated if all the recursive calls are applied to an argument, and none of them are nested. This argument needs to be explicit, and examples such as

```
let corec[constructor] alternating_bools
       = false :: map not alternating_bools
```

are thus ruled out. Right-hand sides can contain calls to previously defined `corec` functions as long as they are not nested.

When a function f is defined using the `corec` keyword, it is bound in the current environment. For simplicity, we impose the restriction that f take only one argument (by forbidding curried definitions with the `corec` keyword). This is a mild restriction, as this argument can be a tuple. Also, because of how functions defined with `corec` are evaluated, we disallow nested recursive calls to f.

The interesting part occurs when the function f is called on an argument a. Since our language is call-by-value, a is first evaluated and bound in the current environment. We then proceed to generate the recursive equations that the value of $f(a)$ must satisfy.

7.4.2 Partial Evaluation

Partial evaluation is much like normal evaluation except when encountering a recursive call to f. When such a recursive call $f(a_j)$ is encountered, its argument is evaluated, and the call is replaced by a variable x_j corresponding to a_j. The variable x_j might be fresh if a_j had not been seen before, or it might be the one already associated with a_j.

Coming back up the abstract syntax tree, some operations cannot be performed. If the condition of an `if` statement was only partially evaluated, we cannot know

which branch to evaluate next; the same thing happens for the condition of a `while` loop or an argument that is pattern-matched.

Particular care must be taken when evaluating the `\&\&` and `||` constructs. These are usually implemented lazily. If the first argument of `\&\&` evaluates to `false`, then it should return `false`. But if it only partially evaluates, then the second argument cannot be evaluated. However, we choose to partially evaluate it anyway, in case it contains recursive calls; thus our implementations of `\&\&` and `||` in the partial evaluator are not strictly lazy.

7.4.3 Equality of Regular Coinductive Terms

Equality of values, and in particular equality of cyclic data structures, plays a central role in the process of generating the equations corresponding to the call of a recursive function. A new equation is generated for each recursive call whose argument has not been previously seen. To assess whether the argument has been previously seen, a set of objects previously encountered is maintained. At each new recursive call, the argument is tested for membership in this set by testing equality with each member of the set. To ensure termination, equality on values that are observationally equivalent must return true.

Unfortunately, OCaml's documentation tells us that "equality between cyclic data structures may not terminate." In practice, the OCaml equality test returns `false` if it can find a difference in finite time, otherwise continues looping forever. In short, it never returns `true` when the arguments are cyclic and bisimilar.

```
> let rec zeros = 0 :: zeros and ones = 1 :: ones;;
val zeros : int list = [0; 0; 0; 0; 0; 0; 0; ...]
val ones : int list = [1; 1; 1; 1; 1; 1; 1; ...]
```

```
> zeros = ones;;
- : bool = false
> zeros = zeros;; (* does not terminate *)
> let rec zeros2 = 0 :: 0 :: zeros2;;
val zeros2 : int list = [0; 0; 0; 0; 0; 0; 0; ...]
> zeros = zeros2;; (* does not terminate *)
```

We would like to create a new equality, simply denoted `=`, that would work the same as in OCaml on every value except cyclic data structures. On cyclic data structures, this equality should correspond to observational equality, so that both calls `zeros = zeros` and `zeros = zeros2` above should return `true`. Note that the OCaml physical identity relation `==` is not suitable: `zeros == zeros2` would return `false`. More importantly, even two instances of a pair of integers formed at different places in the program would not be equal under `==`, although they are observationally equivalent.

To allow cyclic data structures and recursive functions, values are represented internally with capsules. We are thus interested in creating observational equality on capsules. Recall that capsules are essentially finite coalgebras, finite coalgebraic representations of a regular closed λ-coterm. Let us describe the equality algorithm on a simplification of our language where value expressions can only be variables, literal integers, injections into a sum type or tuples.

Let Cap be the set of capsules. The domain of the equality is the set of pairs of capsules – $\mathsf{Cap}^2 = \mathsf{Cap} \times \mathsf{Cap}$. The codomain is the two-element Boolean algebra 2. The diagram (5.1) is instantiated to

$$\begin{array}{ccc} \mathsf{Cap}^2 & \xrightarrow{\quad h \quad} & \mathbb{2} \\ \gamma \downarrow & & \uparrow \alpha \\ \mathbb{2} + \mathsf{Cap}^2 + \mathsf{list}\ (\mathsf{Cap}^2) & \xrightarrow[\mathsf{id}_{\mathbb{2}} + h + \mathsf{map}\ h]{} & \mathbb{2} + \mathbb{2} + \mathsf{list}\ \mathbb{2} \end{array}$$

where the functor is $FX = \mathbb{2} + X + \mathsf{list}\ X$. Here, $\mathsf{list}\ X$ denotes lists of elements of type X, and the map function iterates a function over a list, returning a list of the results.

The function γ matches on the first component of each capsule distinguishing between the base cases and then ones in which equality needs to be recursively determined. If they are both literal integers, it returns $\iota_1(\mathsf{true})$ if they are equal and $\iota_1(\mathsf{false})$ otherwise. If either one is a variable, it looks up its value in the corresponding environment. If they are injections of e_1 and e_2, it returns $\iota_2(e_1, e_2)$. If they are tuples, it creates a list l of pairs whose nth element is the pair of the nth elements of the first and second tuple and returns $\iota_3(l)$. The function α, which processes the results of recursive calls, is the identity on the first two projections, and on $\iota_3(l)$ returns the conjunction of all Boolean values in the list l, that is true if all the elements of l are true, false otherwise.

The naive algorithm given by this diagram is quadratic and it compares all pairs of variables in the capsules. However it turns out that we can see the capsules as finite automata, and they are equal if and only if their corresponding automata are equivalent. There is a known $n\alpha(n)$ algorithm by Hopcroft and Karp [HK71], where α is the inverse of the Ackermann function.[2]

[2] Hopcroft and Karp initially thought the algorithm was linear, but the complexity was later corrected to $n\alpha(n)$

7.5 Conclusions

Coalgebraic (coinductive) datatypes and algebraic (inductive) datatypes are similar in many ways. They are defined in the same way by recursive type equations, algebraic types as least (or initial) solutions and coalgebraic types as greatest (or final) solutions. Because of this similarity, one would like to program with them in the same way, by defining functions by structural recursion using pattern matching. However, because of the non-well-foundedness of coalgebraic data, it must be possible for the programmer to circumvent the standard semantics of recursion and specify alternative solution methods for recursive equations. Up to now, there has been little programming language support for this.

In this chapter we have presented CoCaml, an extension of OCaml with new programming language constructs to address this issue. We have shown through numerous examples that coalgebraic types can be useful in many applications and that computing with them is in most cases no more difficult than computing with algebraic types. Although these alternative solution methods are nonstandard, they are quite natural and can be specified in succinct ways that fit well with the familiar style of recursive functional programming.

Chapter 8

Related Work

Many of the problems addressed in this dissertation — representing and reasoning about the state of computation, reasoning about locality, and computing with coinductive types — have been addressed before, with varying degrees of success. This chapter presents previous work in those topics, and highlights the differences with our work.

8.1 Representation of the state of computation

Reasoning about the state of computation has been studied for decades and we only present here the major historical steps. There is much previous work on reasoning about the local state and references [FH92a, MT92, Pit97, Pit00, PS93, PS98]. State is typically modeled by some form of heap from which storage locations can be allocated and deallocated [HMT84, MT, MT89a, MT91, MS77, Sco72, Sto81]. Others use game semantics to reason about local state [AHM98, AM96, Lai04]. Moggi [Mog91] uses monads to model state. Our approach is most closely related to the work of Mason and Talcott [MT, MT89a, MT91], Felleisen and Hieb [FH92a], to the infini-

tary λ-calculus [BK09, KdV05], and especially to the syntactic theories of control and state of Felleisen, Findler, and Flatt [FFF09]. Abadi, Cardelli, Curien and Lévy study substitutions explicitly [ACCL91], while Curien develops a calculus based on closures [Cur91]. Moran and Sands develop an abstract machine to handle the call-by-need λ-calculus [MS99]. Objects can be modeled as collections of mutable bindings, as for example in the ς-calculus of Abadi and Cardelli [AC96]. In Chapters 2 and 3, we have avoided the introduction of mutable datatypes other than λ-terms in order to develop the theory in its simplest form and to emphasize that no auxiliary datatypes are needed to provide a basic operational semantics for a statically-scoped higher-order language with functional and imperative features.

8.2 Separation logic

There are two main differences between our work of Chapter 4 and the previous work on separation logic. In previous work, the authors usually adopt either an imperative, C-style programming language with low-level heap operations, or a functional, ML-style programming language with immutable variables and explicit references. According to Mason and Talcott [MT89b], in functional languages there are two approaches to introducing objects with memory: the LISP approach, where all variables are mutable, and the ML approach, where all variables are immutable and references are introduced. One of the reasons why the ML view is usually chosen for separation logic on functional programs is that having immutable variables is the only way to get a correct semantics based on closures [Pot12]. By using capsules instead, we are able to use mutable variables in the style of LISP.

The second main difference in this work is that we insist that all capsule environments σ are closed, i.e., that every free variable appearing in a $\sigma(x)$ should be

defined in σ. To us, this seems like a very natural thing. But as far as we know, none of the previous work requires anything like this. When using C-style languages with an environment and a heap, writing down a similar condition would require both the environment and the heap, whereas the separation logic definitions usually only use heaps. Even Neelakantan Krishnaswami et al. [KBAR07], though using an ML-style language, explicitly say that they permit dangling pointers as long as the pointers themselves are well typed. Note that, if trying to relate the semantics of capsules with, say, a more traditional semantics using closures and a heap, the capsule environment behaves like a heap rather than like an environment in the traditional sense, as we saw in Chapter 3.

The original work on separation logic, summarized by Reynolds [Rey02], uses an imperative, C-style programming language with low-level commands and already gives a proof of a version of the frame rule.

Our work is most closely related to work by Krishnaswami, Birkedal, Aldrich and Reynolds [KBAR07, Kri10], who give a separation logic for ML. However, our system allows mutable variables in the style of LISP, whereas theirs uses explicit references allocated in an explicit heap.

Birkedal, Torp-Smith and Yang [BTSY06] also study the frame rule in the context of a higher-order language, idealized Algol extended with heaps, but their stack variables are immutable as well.

There has been some work on so-called higher-order stores [RS06, BRSY08, SBRY09], where some code can be stored in a heap cell. Because any λ-abstractions can be stored in the environment, and executing some of them can have side-effects, our setup naturally supports higher-order stores.

8.3 Non-Well-Founded Computation

The work of Chapters 5, 6 and 7 was inspired by work on recursive coalgebras [ALM07] and Elgot algebras [AMV06]. In Chapter 5, we have extended and clarified the results in [ALM07] by providing a different proof that works on a larger class of functors, as well as provided several examples of functions defined using this scheme.

The theoretical results of Adámek, Lücke, and Milius [ALM07] and the results of Chapters 6 and 7 are concerned with the properties of the domain C ensuring unique solutions to the diagram (5.1). Capretta, Uustalu and Vene [CUV09] studied the dual problem of characterizing properties of the codomain A ensuring this property. The work of Adámek, Milius, and Velebil on Elgot algebras [AMV06] is relevant to our work on recursive definitions that do not have unique solutions. Elgot algebras provide specified canonical solutions rather than unique ones. The canonical solutions must satisfy two axioms, the first ensuring that solutions are independent of the representation of the input and are thus well-defined on a final coalgebra, and the second that allows multiple fixpoints to be parameterized and computed sequentially. The latter property gives an alternative approach to mutually recursive functions.

Syme [Sym06] describes the "value recursion problem" and proposes an approach involving laziness and substitution, eschewing mutability. He also gives a formal calculus for reasoning about the system, along with several examples. One major concern is with side effects, but this is not a particular concern for us. His approach is not essentially coalgebraic, as bisimilar objects are not considered equal. Whereas he must perform substitution on the circular object, we can use variable binding in the environment, as this is invisible with respect to bisimulation, which is correspondingly much simpler. He also claims that "compelling examples of the im-

portance of value recursion have been missing from the literature". We have tried to fill the gap in this dissertation. Many more examples appear in other works, notably in work of Ancona which we discuss below.

Sperber and Thiemann [ST98] propose replacing ref cells with a safe pointer mechanism for dealing with mutable objects. Again, this is not really coalgebraic. They state that "ref cells, when compared to mechanisms for handling mutable data in other programming languages, impose awkward restrictions on programming style," a sentiment with which we wholeheartedly agree.

Capsule semantics address the same issues as Felleisen's and Hieb's theories of syntactic state [FH92b], but capsules are considerably simpler. A major advantage is the elimination of the explicit context present in [FH92b].

Hirschowitz, Leroy, and Wells [HLW03] suggest a safe initialization method for cyclic data structures. Again, their approach is not coalgebraic and uses substitution, which precludes further modification of the data objects once they are created.

Close to our work is the recent paper by Widemann [yW11], which is explicitly coalgebraic. He uses final coalgebras to interpret datatype definitions in a heap-based model with call-by-value semantics. Circular data objects are represented by cycles of pointers created by substitution. The main focus is low-level implementation of evaluation strategies, including cycle detection, and examples are mainly search problems. He also proposes a "front-end language" constructs as an important problem for future work, which is one of the issues we have addressed here.

The question of equality of circular data structures in OCaml has been subject of investigation in, e.g, [Gre10], where the `cyclist` library can be found. The `cyclist` library provides some functions on infinite lists in OCaml. However, this is limited to lists and does not handle any other coinductive type. Another relevant paper

where functions on lists in an ML like language are discussed is [CP98]. There is also work in Scheme [AD08] which defines observational equality for trees and graphs. Our language constructs could also be easily transferred to Scheme.

In the area of logic programming, similar challenges have been tackled. Coinductive logic programming (coLP) [SMB+06, SMB+07] has been recently introduced as a step forward toward developing logic programs containing both finite and regular coinductive terms. The operational semantics is obtained by computing the greatest fixed point of a logic program. Inspired by coLP, Ancona and Zucca defined corecursive FeatherWeight Java (coFJ) [AZ12a] which extends FeatherWeight Java with language support for cyclic structures. In [AZ12b] they provide a translation from coFJ into coLP, clarifying the connections between the two frameworks, and, more notably, providing an effective implementation of coFJ. In [ZA13], they define a type system for coFJ that allows the user to specify that certain methods are not allowed to return undetermined when the solution of the equation system is not unique. Ancona has also improved the state of the art on regular corecursion in Prolog [Anc12], by extending the interpreter with a new clause that enables simpler definitions of coinductive predicates.

Work on cyclic structures in lazy languages can be found in e.g. [FS96, TW01, dSOC12, GHUV06]. In these works, explicit modeling of back pointers (and visited nodes) is used, requiring for instance the use of nested datatypes, and no new program construct is proposed. In our work, we do not touch the datatype definition: the cyclic structure of an object is detected automatically.

Chapter 9

Summary and Future Directions

This dissertation has presented capsules, a simple algebraic representation of the state of computation for a programming language with both functional and imperative paradigms. We have shown how capsules are the mathematical concept behind closures, and how they can be used in logic and verification, with the example of separation logic. Capsules also provide a clean way of representing coalgebraic datatypes. This has helped us design programming language constructs to handle coalgebraic datatypes in a way similar to the way we handle algebraic datatypes. These constructs are implemented in the CoCaml programming language.

However, these results do not close the book on higher-order imperative programming languages and non-well-founded computation. On the contrary, they can be extended in many directions. This chapter identifies future areas of research.

A capsule-based verification system for Javascript. In chapter 4, we just scratched the surface on capsules and verification. We believe that capsules, by their simplicity, can help in designing verification systems for languages with both functional and imperative paradigms. As an application, we would like to design a

verification method for Javascript based on capsules.

Abstract machines and Capsules. Landin's SECD machine was the first model of an abstract machine for the λ-calculus [Lan64, Dan05]. It has been highly influential in the treatment of functional programs, and we would like to develop a similar framework for higher-order imperative programming languages, by developing an abstract machine around capsules.

Static checking of functions on coinductive types. The current implementation of CoCaml (version 4) provides very few static checks on functions defined with the `corec` keyword. For example, a function defined with the `constructor` solver can only return a result if every recursive call is the argument of a constructor, but this is not statically checked. This `constructor` solver is closely related to the `cofix` keyword of the Coq proof assistant [Cdt13], which checks this property statically. However, for other solvers, it is not clear what the static checks should be and what they could guarantee. We would like to explore such static checks, and be able to statically and automatically rule out some wrong definitions.

More solvers and examples in CoCaml. In this dissertation we have already presented many examples where CoCaml can be used. We are currently exploring the possibility of defining a solver that would define coinductive results by iteration, thus having some properties of both the `constructor` and the `iterator` solvers. This solver would be helpful to define the `filter` function on streams more intuitively, as well as defining operations on infinite floats similar to the operations on p-adic numbers we currently have.

CoCaml programs seen as proofs. In Coq, the `cofix` keyword can define both coinductive terms and proofs using corecursion [Cdt13,HNDV13]. Using the proofs-as-programs principle, we would like to explore what programs defined with the `corec` keyword look like when we see them as proofs, and whether they could lead to interesting new proof principles.

Appendix A

Derivations of examples

In this appendix we provide the derivations of the examples found in Chapters 2 and 3.

A.1 Capsules

Example 2.5.1 ($\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0) \xrightarrow{*}_{\mathsf{ca}} 1$

Proof.

$$
\begin{array}{rll}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, & [\,]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = \lambda y.x'\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, & [x' = 1]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ \mathsf{let}\ x = 2\ \mathsf{in}\ f'\ 0, & [x' = 1,\ f' = \lambda y.x']\ \rangle \\
\to_{\mathsf{ca}} & \langle\ f'\ 0, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ (\lambda y.x')\ 0, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ x', & [x' = 1,\ f' = \lambda y.x',\ x'' = 2,\ y' = 0]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ 1, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2,\ y' = 0]\ \rangle
\end{array}
$$

□

Example 2.5.2 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \xrightarrow{*}_{\mathsf{ca}} 2$

Proof.

$$
\begin{array}{lll}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0, & [\,]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = \lambda y.x'\ \mathsf{in}\ x' := 2; f\ 0, & [x' = 1]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ x' := 2; f'\ 0, & [x' = 1,\ f' = \lambda y.x']\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ f'\ 0, & [x' = 2,\ f' = \lambda y.x']\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ (\lambda y.x')\ 0, & [x' = 2,\ f' = \lambda y.x']\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ x', & [x' = 2,\ f' = \lambda y.x',\ y' = 0]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ 2, & [x' = 2,\ f' = \lambda y.x',\ y' = 0]\ \rangle
\end{array}
$$

□

Example 2.5.3 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = (\mathsf{let}\ x = 2\ \mathsf{in}\ \lambda y.x)\ \mathsf{in}\ f\ 0) \xrightarrow{*}_{\mathsf{ca}} 2$

Proof.

$$
\begin{array}{lll}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = (\mathsf{let}\ x = 2\ \mathsf{in}\ \lambda y.x)\ \mathsf{in}\ f\ 0, & [\,]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = (\mathsf{let}\ x = 2\ \mathsf{in}\ \lambda y.x)\ \mathsf{in}\ f\ 0, & [x' = 1]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = \lambda y.x''\ \mathsf{in}\ f\ 0, & [x' = 1,\ x'' = 2]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ f\ 0, & [x' = 1,\ x'' = 2,\ f = \lambda y.x'']\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ (\lambda y.x'')\ 0, & [x' = 1,\ x'' = 2,\ f = \lambda y.x'']\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ x'', & [x' = 1,\ x'' = 2,\ f = \lambda y.x'',\ y' = 0]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ 2, & [x' = 1,\ f = \lambda y.x',\ y' = 0]\ \rangle
\end{array}
$$

□

Example 2.5.4 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f := \lambda y.x; f\ 0) \xrightarrow{*}_{\mathsf{ca}} 2$

Proof.

$$
\begin{array}{lll}
& \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f := \lambda y.x; f\ 0, & [\,]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f := \lambda y.x; f\ 0, & [x' = 1]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ \mathsf{let}\ x = 2\ \mathsf{in}\ f' := \lambda y.x; f'\ 0, & [x' = 1,\ f' = \lambda y.x']\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ f' := \lambda y.x''; f'\ 0, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ f'\ 0, & [x' = 1,\ f' = \lambda y.x'',\ x'' = 2]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ (\lambda y.x'')\ 0, & [x' = 1,\ f' = \lambda y.x'',\ x'' = 2]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ x'', & [x' = 1,\ f' = \lambda y.x'',\ x'' = 2,\ y' = 0]\ \rangle \\
\rightarrow_{\mathsf{ca}} & \langle\ 2, & [x' = 1,\ f' = \lambda y.x'',\ x'' = 2,\ y' = 0]\ \rangle
\end{array}
$$

□

Example 2.5.5 $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3) \xrightarrow{*}_{\mathsf{ca}} 6$

Proof. In this example e stands for $\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n$.

$$
\begin{array}{lll}
& \langle\ \mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3, & [\,]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ f\ 3, & [f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ \mathsf{if}\ n_1 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n_1 - 1) \times n_1, & [f = e,\ n_1 = 3]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ (f\ 2) \times n_1, & [f = e,\ n_1 = 3]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ (\mathsf{if}\ n_2 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n_2 \times f(n_2 - 1)) \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ (f\ 1) \times n_2 \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ (\mathsf{if}\ n_3 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n_3 \times f(n_3 - 1)) \times n_2 \times n_1, & \\
& & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ (f\ 0) \times n_3 \times n_2 \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 3]\ \rangle \\
\xrightarrow{*}_{\mathsf{ca}} & \langle\ (\mathsf{if}\ n_4 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n_4 \times f(n_4 - 1)) \times n_3 \times n_2 \times n_1, &
\end{array}
$$

$$[f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1,\ n_4 = 0]\ \rangle$$

$$\xrightarrow{*}_{\mathsf{ca}}\ \langle\ 1 \times n_3 \times n_2 \times n_1, \quad [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1,\ n_4 = 0]\ \rangle$$

$$\xrightarrow{*}_{\mathsf{ca}}\ \langle\ 6, \quad [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1,\ n_4 = 0]\ \rangle$$

□

A.2 Closures

Example 3.2.5 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0) \xrightarrow{*}_{\mathsf{cl}} 1$

Proof.

$$\langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, \quad [\ ], [\ ]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda x.\mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0,\ [\ ]\}\ 1, \quad [\ ], [\ ]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0\ \square, \quad [x = \ell_1] :: [\ ], [\ell_1 = 1]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda f.\mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0,\ [x = \ell_1]\}\ \lambda y.x\ \square, \quad [x = \ell_1] :: [\ ], [\ell_1 = 1]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda f.\mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0,\ [x = \ell_1]\}\ \{\lambda y.x,\ [x = \ell_1]\}\ \square, \quad [x = \ell_1] :: [\ ], \quad [\ell_1 = 1]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0\ \square\ \square, \quad [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\ ], \quad [\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda x.f\ 0,\ [x = \ell_1, f = \ell_2]\}\ 2\ \square\ \square, \quad [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\ ], \quad [\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ f\ 0\ \square\ \square\ \square, \quad [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\ ], \quad [\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda y.x,\ [x = \ell_1]\}\ 0\ \square\ \square\ \square,$$

$$[f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,],$$

$$[\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ x\ \square\ \square\ \square\ \square,$$

$$[x = \ell_1, y = \ell_4] :: [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,],$$

$$[\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ 1\ \square\ \square\ \square\ \square,$$

$$[x = \ell_1, y = \ell_4] :: [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,],$$

$$[\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ 1\ \square\ \square\ \square, \qquad [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,],$$

$$[\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ 1\ \square\ \square, \qquad [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,],$$

$$[\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ 1\ \square, \qquad [x = \ell_1] :: [\,], [\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ 1, \qquad [\,], [\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle$$

□

Example 3.2.6 $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \xrightarrow{*}_{\mathsf{cl}} 2$

Proof.

$$\langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0, \qquad [\,], [\,]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda x.\mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0,\ [\,]\}\ 1, \qquad [\,], [\,]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0\ \square, \qquad [x = \ell_1] :: [\,], [\ell_1 = 1]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda f.x := 2; f\ 0,\ [x = \ell_1]\}\ \lambda y.x\ \square, \qquad [x = \ell_1] :: [\,], [\ell_1 = 1]\ \rangle$$

$$\rightarrow_{\mathsf{cl}}\ \langle\ \{\lambda f.x := 2; f\ 0,\ [x = \ell_1]\}\ \{\lambda y.x,\ [x = \ell_1]\}\ \square, \qquad [x = \ell_1] :: [\,], [\ell_1 = 1]\ \rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, x := 2; f\ 0\ \Box\ \Box, \quad [x=\ell_1, f=\ell_2] :: [x=\ell_1] :: [\,], [\ell_1 = 1, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}]\,\rangle$$

$$\xrightarrow{*}_{\mathsf{cl}} \langle\, f\ 0\ \Box\ \Box, [x=\ell_1, f=\ell_2] :: [x=\ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}]\,\rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, \{\lambda y.x,\ [x=\ell_1]\}\ 0\ \Box\ \Box, \quad [x=\ell_1, f=\ell_2] :: [x=\ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}]\,\rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, x\ \Box\ \Box\ \Box, \quad [x=\ell_1, y=\ell_3] :: [x=\ell_1, f=\ell_2] :: [x=\ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}, \ell_3 = 0]\,\rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, 2\ \Box\ \Box\ \Box, \quad [x=\ell_1, y=\ell_3] :: [x=\ell_1, f=\ell_2] :: [x=\ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}, \ell_3 = 0]\,\rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, 2\ \Box\ \Box, \quad [x=\ell_1, f=\ell_2] :: [x=\ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}, \ell_3 = 0]\,\rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, 2\ \Box, \quad [x=\ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}, \ell_3 = 0]\,\rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\, 2, \quad [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x,\ [x=\ell_1]\}, \ell_3 = 0]\,\rangle$$

□

This final example is particularly interesting as it shows how nested $\Box$ allow to interpret the same variable in different scopes. In all the example e stands for $\{\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n,\ [f = \ell_1]\}$, and d stands for $\{\lambda n.n,\ [\,]\}$, a dummy value used when creating the recursive function f.

Example 3.2.7 $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3) \xrightarrow{*}_{\mathsf{cl}} 6$

Proof.

$$
\begin{aligned}
& \langle\ \mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3, \qquad [\,], [\,]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ f := \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n; f\ 3\ \square, \quad [f = \ell_1] :: [\,], [\ell_1 = d]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ f\ 3\ \square, \qquad [f = \ell_1] :: [\,], [\ell_1 = e]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \square\ \square, \qquad [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
& \qquad [\ell_1 = e, \ell_2 = 3]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ (f\ 2) \times n\ \square\ \square, \qquad [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
& \qquad [\ell_1 = e, \ell_2 = 3]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ (\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \square) \times n\ \square\ \square, \\
& \qquad [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
& \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ ((f\ 1) \times n\ \square) \times n\ \square\ \square, \quad [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
& \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ ((\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \square) \times n\ \square) \times n\ \square\ \square, \\
& \qquad [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
& \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ (((f\ 0) \times n\ \square) \times n\ \square) \times n\ \square\ \square, \\
& \qquad [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
& \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1]\ \rangle \\
\xrightarrow{*}_{\mathsf{cl}} & \langle\ (((\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \square) \times n\ \square) \times n\ \square) \times n\ \square\ \square, \\
& \qquad [f = \ell_1, n = \ell_5] :: [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: \\
& \qquad [f = \ell_1] :: [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle
\end{aligned}
$$

$$\rightarrow_{\mathsf{cl}} \langle\ (((\mathsf{if}\ 0 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \Box) \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box,$$

$$[f = \ell_1, n = \ell_5] :: [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] ::$$

$$[f = \ell_1] :: [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\ (((1\ \Box) \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box,$$

$$[f = \ell_1, n = \ell_5] :: [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] ::$$

$$[f = \ell_1] :: [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle$$

$$\rightarrow_{\mathsf{cl}} \langle\ ((1 \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box,$$

$$[f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,],$$

$$[\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle$$

$$\xrightarrow{*}_{\mathsf{cl}} \langle\ ((1 \times 1) \times n\ \Box) \times n\ \Box\ \Box, [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,],$$

$$[\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle$$

$$\xrightarrow{*}_{\mathsf{cl}} \langle\ ((1 \times 1) \times 2) \times n\ \Box\ \Box, \quad [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,],$$

$$[\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle$$

$$\xrightarrow{*}_{\mathsf{cl}} \langle\ 6, \quad [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle$$

$\Box$

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996. **Cited** on page 181.

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. **Cited** on page 181.

[AD08] Michael D. Adams and R. Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In *Proc. 13 ACM SIGPLAN Int. Conf. Functional Programming*, pages 179–188, 2008. **Cited** on page 185.

[AH01] Kamal Aboul-Hosn. Programming with private state. Honors Thesis, The Pennsylvania State University, December 2001. **Cited** on page 10.

[AHK06] Kamal Aboul-Hosn and Dexter Kozen. Relational semantics for higher-order programs. In Tarmo Uustalu, editor, *Proc. 8th Int. Conf. Mathematics of Program Construction (MPC'06)*, volume 4014 of *Lecture Notes in Computer Science*, pages 29–48. Springer, July 2006. **Cited** on page 10.

[AHK07] Kamal Aboul-Hosn and Dexter Kozen. Local variable scoping and Kleene algebra with tests. *J. Log. Algebr. Program.*, 2007. DOI: 10.1016/j.jlap.2007.10.007. **Cited** on page 10.

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, pages 334–344, Washington, DC, USA, 1998. IEEE Computer Society. **Cited** on page 180.

[ALM07] Jiří Adámek, Dominik Lücke, and Stefan Milius. Recursive coalgebras of finitary functors. *Theoretical Informatics and Applications*, 41:447–462, 2007. **Cited** on pages 6, 92, 94, 95, 96, 100, 101, 103, 107, 109, 113, 115, and 183.

[AM96] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized ALGOL with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996. **Cited** on page 180.

[AMV06] Jiří Adámek, Stefan Milius, and Jiří Velebil. Elgot algebras. *Log. Methods Comput. Sci.*, 2(5:4):1–31, 2006. **Cited** on pages 92, 95, 106, 115, and 183.

[Anc12] Davide Ancona. Regular corecursion in Prolog. In Sascha Ossowski and Paola Lecca, editors, *SAC*, pages 1897–1902. ACM, 2012. **Cited** on page 185.

[AZ12a] Davide Ancona and Elena Zucca. Corecursive Featherweight Java. In *FTfJP'012 - Formal Techniques for Java-like Programs*, 2012. **Cited** on page 185.

[AZ12b] Davide Ancona and Elena Zucca. Translating Corecursive Featherweight Java in Coinductive Logic Programming. In *Co-LP 2012 - A workshop on Coinductive Logic Programming*, 2012. **Cited** on page 185.

[Bak11] Andrew Baker. An introduction to p-adic numbers and p-adic analysis. `http://www.maths.gla.ac.uk/~ajb/dvi-ps/padicnotes.pdf`, March 2011. School of Mathematics and Statistics, University of Glasgow. **Cited** on page 166.

[BBTS07] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29, August 2007. **Cited** on pages 77 and 88.

[BCY05] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. In *Proc. 21st Conf. Math. Found. Programming Semantics*, pages 247–276, 2005. **Cited** on page 77.

[BK09] Henk P. Barendregt and Jan Willem Klop. Applications of infinitary lambda calculus. *Inf. and Comput.*, 207(5):559–582, 2009. **Cited** on pages 10 and 181.

[BRSY08] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. A simple model of separation logic for higher-order store. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 348–360, Berlin, Heidelberg, 2008. Springer-Verlag. **Cited** on page 182.

[BTSY06] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. *CoRR*, abs/cs/0610081, 2006. **Cited** on page 182.

[BZ02] Gérard Boudol and Pascal Zimmer. Recursion in the call-by-value lambda-calculus. In Zoltán Ésik and Anna Ingólfsdóttir, editors, *FICS*, volume NS-02-2 of *BRICS Notes Series*, pages 61–66. University of Aarhus, 2002. **Cited** on page 153.

[Cap07] Venanzio Capretta. An introduction to corecursive algebras. `http://www.cs.ru.nl/~venanzio/publications/brouwer_seminar_4_12_2007.pdf`, 2007. **Cited** on pages 96 and 113.

[Cap11] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theor. Comput. Sci.*, 412(38):5006–5024, 2011. **Cited** on page 162.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 238–252, Los Angeles, 1977. ACM Press, New York. **Cited** on page 127.

[Cdt13] The Coq development team. The Coq proof assistant reference manual. 2013. **Cited** on pages 187 and 188.

[cH98] Naim Çagman and J. Roger Hindley. Combinatory weak reduction in lambda calculus. *Theor. Comput. Sci.*, 198(1-2):239–247, 1998. **Cited** on page 15.

[Cho10] Stephen Chong. Lecture notes on abstract interpretation. `http://www.seas.harvard.edu/courses/cs152/2010sp/lectures/lec20.pdf`, 2010. Harvard University. **Cited** on page 127.

[CoC12] CoCaml project. `http://www.cs.cornell.edu/Projects/CoCaml/`, December 2012. **Cited** on page 121.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proc. 22nd Annual IEEE Symp. Logic in Computer Science (LICS07)*, pages 366–378. IEEE, 2007. **Cited** on pages 77, 78, 79, 82, and 84.

[CP98] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. *Electr. Notes Theor. Comput. Sci.*, 11:1–21, 1998. **Cited** on page 185.

[Cur91] Pierre-Louis Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82(2):389–402, 1991. **Cited** on page 181.

[CUV09] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Corecursive algebras: A study of general structured corecursion. In Marcel Oliveira and Jim Woodcock, editors, *12th Brazilian Symp. Formal Methods*, volume

5902 of *Lecture Notes in Computer Science*, pages 84–100, Berlin, 2009. Springer. **Cited** on pages 92, 113, and 183.

[Dan05] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In *Implementation and Application of Functional Languages*, pages 52–71. Springer, 2005. **Cited** on page 187.

[dSOC12] Bruno C. d. S. Oliveira and William R. Cook. Functional programming with structured graphs. In Peter Thiemann and Robby Bruce Findler, editors, *ICFP*, pages 77–88. ACM, 2012. **Cited** on page 185.

[Epp99] Adam Eppendahl. Coalgebra-to-algebra morphisms. *Electronic Notes in Theoretical Computer Science*, 29, 1999. **Not cited**

[FFF09] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. **Cited** on pages 10 and 181.

[FH92a] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992. **Cited** on pages 10 and 180.

[FH92b] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992. **Cited** on page 184.

[FMS05] Maribel Fernández, Ian Mackie, and François-Régis Sinot. Closed reduction: explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005. **Cited** on page 15.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 284–294, New York, NY, USA, 1996. ACM. **Cited** on page 185.

[GHUV06] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In H. Nilsson, editor, *Proc. of 7th Symp. on Trends in Functional Programming, TFP 2006*, pages 173–188. Univ. of Nottingham, 2006. **Cited** on page 185.

[Gre10] Dmitry Grebeniuk. Library `ocaml-cyclist`. `https://forge.ocamlcore.org/projects/ocaml-cyclist/`, June 2010. **Cited** on page 184.

[GT74] J. A. Goguen and J. W. Thatcher. Initial algebra semantics. In *15th Symp. Switching and Automata Theory*, pages 63–77. IEEE, 1974. **Cited** on page 149.

[HK71] John E Hopcroft and Richard M Karp. A linear algorithm for testing equivalence of finite automata. Technical report, Cornell University, 1971. **Cited** on page 178.

[HK84] David Harel and Dexter Kozen. A programming language for the inductive sets, and applications. *Information and Control*, 63(1–2):118–139, 1984. **Cited** on pages 96 and 115.

[HLW03] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *PPDP 2003*, pages 160–171, 2003. **Cited** on pages 152, 153, and 184.

[HMT84] Joseph Y. Halpern, Albert R. Meyer, and Boris A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Proc. 11th ACM Symp. Principles of Programming Languages (POPL'84)*, pages 245–257, New York, NY, USA, 1984. **Cited** on pages 2, 10, and 180.

[HNDV13] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–206. ACM, 2013. **Cited** on page 188.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM. **Cited** on page 77.

[Jea11] Jean-Baptiste Jeannin. Capsules and closures. *Electron. Notes Theor. Comput. Sci.*, 276:191–213, September 2011. **Cited** on page 6.

[Jea12] Jean-Baptiste Jeannin. Capsules and closures: A small-step approach. In R. L. Constable and A. Silva, editors, *Kozen Festschrift, LNCS 7230*, pages 106–123. Springer-Verlag, April 2012. **Cited** on page 6.

[JK12a] Jean-Baptiste Jeannin and Dexter Kozen. Capsules and separation. In Nachum Dershowitz, editor, *Proc. 27th ACM/IEEE Symp. Logic in Computer Science (LICS'12)*, pages 425–430, Dubrovnik, Croatia, June 2012. IEEE. **Cited** on page 6.

[JK12b] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Proc. Conf. Descriptional Complexity of Formal Systems (DCFS 2012)*, volume 7386 of *Lecture Notes in Computer Science*, pages 1–19, Braga, Portugal, July 2012. Springer. **Cited** on page 5.

[JKS12] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Programming with coinductive types. Technical Report `http://hdl.`

handle.net/1813/30798, Computing and Information Science, Cornell University, December 2012. **Cited** on page 7.

[JKS13a] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Language constructs for non-well-founded computation. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, pages 61–80, Berlin, Heidelberg, 2013. Springer-Verlag. **Cited** on page 6.

[JKS13b] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Well-founded coalgebras, revisited. Technical Report http://hdl.handle.net/1813/33330, Computing and Information Science, Cornell University, May 2013. **Cited** on page 6.

[KBAR07] Neelakantan R. Krishnaswami, Lars Birkedal, Jonathan Aldrich, and John C. Reynolds. Idealized ML and its separation logic. http://www.cs.cmu.edu/~neelk/, 2007. **Cited** on page 182.

[KdV05] Jan W. Klop and Roel C. de Vrijer. Infinitary normalization. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005. **Cited** on pages 10 and 181.

[Koz11] Dexter Kozen. Realization of coinductive types. In Michael Mislove and Joël Ouaknine, editors, *Proc. 27th Conf. Math. Found. Programming Semantics (MFPS XXVII)*, pages 148–155, Pittsburgh, PA, May 2011. Elsevier Electronic Notes in Theoretical Computer Science. **Cited** on pages 95, 96, 99, and 100.

[Koz12] Dexter Kozen. New. In Ulrich Berger and Michael Mislove, editors, *Proc. 28th Conf. Math. Found. Programming Semantics (MFPS XXVIII)*, pages 13–38, Bath, England, June 2012. Elsevier Electronic Notes in Theoretical Computer Science. **Cited** on pages 42 and 153.

[Kri10] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2010. **Cited** on page 182.

[Lai04] James Laird. A game semantics of local names and good variables. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2004. **Cited** on page 180.

[Lam68] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968. **Cited** on page 101.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964. **Cited** on pages 11, 18, and 187.

[McC81] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of programming languages I*, pages 173–185. ACM, 1981. **Cited** on pages 16 and 18.

[ML71] Saunders Mac Lane. *Categories for the Working Matematician*. Springer, 1971. **Cited** on page 99.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991. **Cited** on pages 2, 10, and 180.

[MS77] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA, 1977. **Cited** on pages 2, 10, and 180.

[MS99] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL*, pages 43–56, 1999. **Cited** on page 181.

[MT] Ian Mason and Carolyn Talcott. Programming, transforming, and proving with function abstractions and memories. **Cited** on pages 10 and 180.

[MT89a] Ian Mason and Carolyn Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Fourth Annual Symposium on Logic in Computer Science. IEEE*, pages 284–293. IEEE Computer Society Press, 1989. **Cited** on pages 10 and 180.

[MT89b] Ian Mason and Carolyn Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Fourth Annual Symposium on Logic in Computer Science. IEEE*, pages 284–293. IEEE Computer Society Press, 1989. **Cited** on page 181.

[MT91] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects, 1991. **Cited** on pages 10 and 180.

[MT92] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *Seventh Annual Symposium on Logic in Computer Science*, pages 186–197. IEEE, 1992. **Cited** on page 180.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007. **Cited** on page 89.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, 2001. Springer-Verlag. **Cited** on page 77.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM. **Cited** on page 77.

[PBC06] Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 137–146, Washington, DC, USA, 2006. IEEE Computer Society. **Cited** on page 77.

[Pit97] Andrew M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997. **Cited** on page 180.

[Pit00] Andrew M. Pitts. Operational semantics and program equivalence. Technical report, INRIA Sophia Antipolis, 2000. Lectures at the International Summer School On Applied Semantics, APPSEM 2000, Caminha, Minho, Portugal, September 2000. **Cited** on page 180.

[Pot] Francois Pottier. Lazy least fixed points in ml. Available from `pauillac.inria.fr/~fpottier/publis/fpottier-fix.pdf`. **Cited** on page 155.

[Pot08] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *Twenty-Third Annual IEEE Symposium on Logic In Computer Science (LICS'08)*, pages 331–340, Pittsburgh, Pennsylvania, June 2008. **Cited** on page 88.

[Pot12] François Pottier, 2012. Private communication. **Cited** on page 181.

[PS93] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 1993. **Cited** on page 180.

[PS98] Andrew M. Pitts and Ian D. B. Stark. Operational reasoning in functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998. **Cited** on page 180.

[Rey00] John C. Reynolds. Intuitionistic reasoning about shared mutable data structures. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000. **Cited** on page 77.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symp. Logic in Computer Science (LICS'02)*, pages 55–74. IEEE, 2002. **Cited** on pages 77, 78, 79, 80, 83, 85, and 182.

[RS06] Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In *In Proc. CSL*, pages 575–590. Springer, 2006. **Cited** on page 182.

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000. **Cited** on pages 26 and 28.

[SBRY09] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. In *In Proceedings of the 18th EACSL Annual Conference on Computer Science Logic*, 2009. **Cited** on pages 88 and 182.

[Sco72] Dana S. Scott. Mathematical concepts in programmng language semantics. In *Proc. 1972 Spring Joint Computer Conferences*, pages 225–34, Montvale, NJ, 1972. AFIPS Press. **Cited** on pages 2, 10, and 180.

[SMB+06] Luke Simon, Ajay Mallya, Ajay Bansal, , and Gopal Gupta. Coinductive logic programming. In Sandro Etalle and Mirosław Truszczyński, editors, *22nd Int. Conf. Logic Programming (ICLP 2006)*, volume 4079 of *Lecture Notes in Computer Science*, pages 330–345. Springer, August 2006. **Cited** on page 185.

[SMB+07] Luke Simon, Ajay Mallya, Ajay Bansal, , and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *34th Int. Colloq. Automata, Languages and Programming (ICALP 2007)*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer, July 2007. **Cited** on page 185.

[SR07] Alexandra Silva and Jan J. M. M. Rutten. Behavioural differential equations and coinduction for binary trees. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 4576 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2007. **Cited** on page 162.

[SR10] Alexandra Silva and Jan J. M. M. Rutten. A coinductive calculus of binary trees. *Inf. Comput.*, 208(5):578–593, 2010. **Cited** on page 162.

[SS98] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11:405–439, 1998. 10.1023/A:1010035624696. **Cited** on page 45.

[ST98] Michael Sperber and Peter Thiemann. ML and the address operator. In *1998 ACM SIGPLAN Workshop on ML*, September 1998. **Cited** on pages 152, 153, and 184.

[Sto81] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1981. **Cited** on pages 2, 10, and 180.

[SYB+10] Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. A semantic foundation for hidden state. In *FOSSACS*, pages 2–17, 2010. **Cited** on page 88.

[Sym06] Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *Proc. ACM-SIGPLAN Workshop on ML (2005)*. Elsevier, March 2006. **Cited** on pages 152, 153, and 183.

[Tay99] Paul Taylor. *Practical Foundations of Mathematics*. Number 59 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999. **Not cited**

[TW01] Franklyn A. Turbak and J. B. Wells. Cycle therapy: A prescription for fold and unfold on regular trees. In *PPDP*, pages 137–149. ACM, 2001. **Cited** on page 185.

[Wik12] Wikipedia. p-adic numbers. `http://en.wikipedia.org/w/index.php?title=P-adic_number&oldid=553107165`, 2012. **Cited** on page 166.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993. **Cited** on pages 35 and 127.

[YO02] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Proc. 5th Foundations of Software Science and Computation Structures (FOSSACS02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer-Verlag, 2002. **Cited** on pages 77 and 83.

[yW11] Baltasar Trancón y Widemann. Coalgebraic semantics of recursion on circular data structures. In Corina Cirstea, Monika Seisenberger, and Toby Wilkinson, editors, *CALCO Young Researchers Workshop (CALCO-jnr 2011)*, pages 28–42, August 2011. **Cited** on pages 152 and 184.

[ZA13] Elena Zucca and Davide Ancona. Safe Corecursion in coFJ. In *FTfJP'012 - Formal Techniques for Java-like Programs. 2013*, 2013. **Cited** on page 185.