PROPERTIES OF SEQUENT-CALCULUS-BASED LANGUAGES

by

PHILIP ALDEN JOHNSON-FREYD

A DISSERTATION

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

December 2017

DISSERTATION APPROVAL PAGE

Student: Philip Alden Johnson-Freyd

Title: Properties of Sequent-Calculus-Based Languages

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Zena M. Ariola	Chair
Geoffrey Hulette	Core Member
Boyana Norris	Core Member
Michal Young	Core Member
Alison Kwok	Institutional Representative

and

Sara D. Hodges	Interim Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded December 2017

DISSERTATION ABSTRACT

Philip Alden Johnson-Freyd

Doctor of Philosophy

Department of Computer and Information Science

December 2017

Title: Properties of Sequent-Calculus-Based Languages

Programmers don't just have to write programs, they are have to reason about them. Programming languages aren't just tools for instructing computers what to do, they are tools for reasoning. And, it isn't just programmers who reason about programs: compilers and other tools reason similarly as they transform from one language into another one, or as they optimize an inefficient program into a better one. Languages, both surface languages and intermediate ones, need therefore to be both efficiently implementable and to support effective logical reasoning. However, these goals often seem to be in conflict.

This dissertation studies programming language calculi inspired by the *Curry-Howard correspondence*, relating programming languages to proof systems. Our focus is on calculi corresponding logically to classical sequent calculus and connected computationally to abstract machines. We prove that these calculi have desirable properties to help bridge the gap between reasoning and implementation.

Firstly, we explore a persistent conflict between extensionality and effects for lazy functional programs that manifests in a loss of confluence. Building on prior work, we develop a new rewriting theory for lazy functions and control which we first prove corresponds to the desired equational theory and then prove, by way of reductions into a smaller system, to be confluent. Next, we turn to the inconsistency between weak-head normalization and extensionality. Using ideas from our study of confluence, we develop a new operational semantics and series of abstract machines for head reduction which show us how to retain weak-head reduction's ease of implementation.

After demonstrating the limitations of the above approach for call-by-value or types other than functions, we turn to typed calculi, showing how a type system can be used not only for mixing different kinds of data, but also different evaluation strategies in a single program. Building on variations of the reducibility candidates method such as biorthogonality and symmetric candidates, we present a uniform proof of strong normalization for our mixed-strategy system which works so long as all the strategies used satisfy criteria we isolate.

This dissertation includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Philip Alden Johnson-Freyd

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
University of Puget Sound, Tacoma, WA

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2017, University of Oregon
Bachelor of Arts, Philosophy and Computer Science, 2011, University of Puget Sound

AREAS OF SPECIAL INTEREST:

Programming Language Theory
Type Theory
Compilers

PROFESSIONAL EXPERIENCE:

Year Round Technical Intern, Sandia CA, 2016-present

Technical Summer Intern, Sandia CA, 2015

Technical Summer Intern, Sandia CA, 2014

Intern Center for Cyber Defenders, Sandia CA, 2013

Graduate Employee, University of Oregon, 2012-present

GRANTS, AWARDS AND HONORS:

Phillip Seely Graduate Fellowship, University of Oregon, 2017

PUBLICATIONS:

Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. Call-by-name extensionality and confluence. Journal of Functional Programming, 27:e12, 2017. doi: 10.1017/S095679681700003X. URL `https://doi.org/10.1017/S095679681700003X`.

Philip Johnson-Freyd, Geoffrey C. Hulette and Zena M. Ariola. Verification by Way of Refinement: A Case Study in the Use of Coq and TLA in the Design of a High Consequence System. In FMICS-AVoCS 2016. LNCS Volume 9933, pg 205-213, Cham. Springer.

Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. First class call stacks: Exploring head reduction. In Olivier Danvy and Ugo de'Liguoro, editors, Proceedings of the Workshop on Continuations, WoC 2015, London, UK, April 12th 2015., volume 212 of EPTCS, pages 1835, 2015. doi: 10.4204/EPTCS.212.2.

Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Structures for structural recursion. In ICFP 15, pages 127-139, New York, NY, USA, 2015. ACM.

ACKNOWLEDGEMENTS

First of all, I would like to thank those who provided financial support for my studies including the National Science Foundation which supported me under Grant No. CCF-1423617 as well as the University of Oregon and especially donor Phillip Seeley who supported me through a fellowship awarded by the Department of Computer Science.

I am grateful of my committee and all of my professors at Oregon who have guided me through my time here. An extra level of thanks goes to my advisor, Zena Ariola without whom none of this would have been possible. She tolerated my tendency to jump to new ideas and new projects while making sure I finished at least some of them. Anyone who works with her gains from her incredible attention to details and her laser focus on finding the core of what is going on in complex material. Her mentorship has been invaluable to me every step of the way.

I would also like to thank my fellow graduate students at Oregon, and especially Paul Downen, Luke Maurer, and Zach Sullivan, who as office mates, friends, and collaborators, I have had so many productive conversations over the years. A special thank you is also in order to Raleigh Foster who always at least pretends to be interested in my late night type-theoretic ravings.

Additionally, I would like to thank the formal methods group at Sandia National Laboratories in California who have treated me as one of their own and given me a place to sit and source of income over the summers. Their always interesting problems and constant reminders to put "school first" and to "not let the perfect be the enemy of the graduation" have often been just what I needed.

In particular, I feel an enourmous debt of gratitude to Geoffrey Hulette who first brought me on and who has mentored me for years.

Finally, I need to thank my family, starting with my brother Theo, who has always been a role model, and my sister Sasha, who is an inspiration. The greatest appreciation is owed to my dad, who first taught me math and programming and whom I miss every day, and to my mom, whose passion, brilliance, and generosity I can only strive to emulate and whose support has never wavered.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Despina. Salvete, amabiles / Buonae puellae!
Fiordiligi e Dorabella. Parla un linguaggio / Che non sappiamo.
Despina. Come comandano / Dunque parliamo: / So il greco e l'arabo, / So il turco e il vandalo; / Lo svevo e il tartaro / So ancor parlar.

Lorenzo Da Ponte and Wolfgang Amadeus Mozart, Così fan tutte

The design of programming languages began even before the invention of the computer. Church's untyped λ-calculus was first conceived for the purpose of organizing logic (Church (1932)). Later, it was shown to be a universal model of computation (Church (1936)), equivalent to the Turing machine (Turing (1937)). Very many programming languages have been developed since, however, the connection between logic and computation remains.

Programming languages should be easy to write, but also easy to reason about. Moreover they should be effectively implementable: we need to be able to efficiently run programs as implemented, and to transform them into more efficient programs. To do these things we need to prove properties about programs and about programming languages. In many instances it is easier not to directly work with the surface languages users write, but rather to transform these into intermediate languages more amenable to formal analysis. Even when full languages are impractical or do not satisfy all the properties of interest, it may be desirable to study smaller core calculi which do satisfy those properties.

This dissertation is situated at this intersection of computation and logic. We design and prove properties about a series of calculi of interest in the design of intermediate languages. These calculi, following Curien and Herbelin (2000), are inspired by a formal connection to logic and in particular the proof theoretic idea of sequent calculus, which helps guide us along the path of good design choices. However, our calculi are in each case motivated by concrete computational questions.

Overview

Roughly the first half of this dissertation–namely Chapters II, III, IV, and V–are focused on untyped systems. Of these, Chapters II and III are, when taken together, largely self contained. They are an expansion of material previously published as Johnson-Freyd, Downen, and Ariola (2017) which I developed in collaboration with Paul Downen and Zena M. Ariola. Some additional material from that publication is also used in the first part of Chapter V. Although sharing insights, Chapter IV can be understood without Chapter III. It incorporates previous published material which I wrote with Paul Downen and Zena M. Ariola (Johnson-Freyd, Downen, and Ariola (2015)).

The latter parts of this dissertation, namely Chapters V and VI concern typed systems. The model in Chapter VI derives, with major modifications, from an appendix to Downen, Johnson-Freyd, and Ariola (2015), which I developed in collaboration with Paul Downen and Zena M. Ariola. That chapter also incorporates some subsequent ideas and improvements I developed in later collaboration with Paul Downen.

In more detail, this dissertation can be broken into four main problem areas.

- In Chapter III we study the problem of rewriting effectful functional programs in a way which respects the equational property that programs which behave the same are the same. The particular problem we study is that of confluence: how can we be assured that the *order* at which we apply rewrite rules does not matter? This problem is of potential practical relevance to the designers of compilers who may want to use rewriting as a tool for optimization, and for whom then confluence (even of a subset of the rewriting rules) can be a big win.

- In Chapter IV we apply the insights from our study of confluence and rewriting to the problem of how even to run lazy functions in a way that at least respects extensionality. As such, we address a contradiction between the way lazy functional programs are usually evaluated (weak head normalization) and the prevailing equational theory for these programs (which incorporates extensionality). Our solution is a form of head normalization tailored for efficiency and which avoids looking under lambdas.

- In Chapter V we reconsider the solutions given in the early chapters, and show that, while they are well suited to lazy functions, they break down with other data types and evaluation strategies, including the more common call-by-value approach used by most popular languages. Consequently, we introduce new *typed* calculi allowing us not only to choose strategies other than call-by-name, but also to even mix strategies in one program.

– Finally, in Chapter VI we take advantage of the introduction of the typed system to prove a property which could not be shown for the untyped calculi: namely, strong normalization. That is, every reduction path for our calculus eventually terminates. Again, this is a property which may reassure compiler writers: a compiler is free to apply the reduction rules from the theory and still be assured that compilation will complete in finite time.

Before developing these contributions, however, we will investigate in detail the nature of lazy functions in Chapter II, motivating the sequent calculus based languages we use.

First though, as the remainder of this introduction, we give a brief introductory tutorial on some of the ideas from programming languages and logic which inform our designs.

λ-Calculus and Programming

$$v \in \text{Terms} ::= x \mid \lambda x.v \mid v\ v$$

$$\begin{array}{rll} \lambda x.v &=_\alpha \lambda y.(v\{y/x\}) & y \notin FV(v) \\ (\lambda x.v)\ v' &=_\beta v\{v'/x\} & \\ \lambda x.v\ x &=_\eta v & x \notin FV(v) \end{array}$$

FIGURE 1. Syntax and Equational Theory of Untyped λ-calculus

The untyped λ-calculus is among the oldest and simplest programming languages. Its grammar, in Fig. 1, is given by a single syntactic sort and just three productions, x a variable, $v\ v'$ a function application, and $\lambda x.v$ defines a function by abstracting over the variable x. The most basic equation, α equivalence, tells us that we are permitted to freely rename local variables. By contrast, the β axiom tells us the meaning of λ-abstraction in terms of how a function behaves when given an argument. The notation $v\{v'/x\}$ refers to *capture avoiding substitution* which simply means replacing all free appearances of the variable x in v for the term v', but doing so in a way which respects α-equivalence (Church (1932)). Finally, η-equivalence tells us that a function which does nothing other than delegate to another function is equivalent to the function it delegates to. Implicitly, the equational theory this defines is the least congruence (an equivalence relation also applied *inside* terms) closed under the rules.

An equational theory tells us when two programs are the same, but in itself the equational theory doesn't tell us how to run programs. To do that we need to, at the very least, orient the equations to produce a *reduction theory*

$$\begin{aligned}(\lambda x.v)\ v' &\to v\{v'/x\} \\ \lambda x.v\ x &\to v && x \notin FV(v).\end{aligned}$$

Here, we again allow reductions to occur inside of terms, and we adopt the practice which we will maintain throughout this dissertation of considering programs up to α-equivalence across all calculi. This reduction theory has desirable properties. First among them is that it is confluent, meaning that its reflexive transitive closure, written $\twoheadrightarrow$, has the diamond property: if v, v_1, v_2 are terms such that $v \twoheadrightarrow v_1$ and $v \twoheadrightarrow v_2$ then there must exist a v_3 such that $v_1 \twoheadrightarrow v_3$ and $v_2 \twoheadrightarrow v_3$.

$$\begin{tikzcd} v \arrow[r, two heads] \arrow[d, two heads] & v_1 \arrow[d, two heads, dashed] \\ v_2 \arrow[r, two heads, dashed] & v_3 \end{tikzcd}$$

Even a reduction theory though does not provide enough information to implement a complete language. To do that, we need a rule to pick which reduction to perform each chance we get. For instance, the top-leftmost redex.

The λ-calculus is *lazy*, which we usually refer to as *call-by-name*. That is, when we can β reduce with *any* term. Many languages however are based instead on call-by-value where the β law only applies when the argument to a function is a *value* and done computing. We will explore this issue in much more depth in Section 4.4.

The λ-calculus is a complete language, allowing us to represent all computable functions on numbers. Related to this fact is that the λ-calculus must include programs which can not be reduced to a normal form (a term for which there are no more reductions). An example of such a non-normalizing term is $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ since

$$(\lambda x.x\ x)(\lambda x.x\ x) \to (x\ x)\{(\lambda x.x\ x)/x\} = (\lambda x.x\ x)(\lambda x.x\ x)$$

Closely tied to this fact is the reason the untyped λ-calculus is unsuitable for Church's goal of an organizing tool for logic. In the original design, λ calculus would be extended with constants for forming propositions (thought of as λ-terms) out of other λ-terms and then λ terms would be interpreted as truth value For instance, we might allow a constant $\neg$ such that $\neg(v)$ is a term whenever v is where $\neg(v)$ is true exactly when v is not. However, this allows paradoxical terms (Kleene and Rosser (1935)).

$$P = (\lambda x.\neg(x\ x))(\lambda x.\neg(x\ x))$$

we can see that

$$\begin{aligned} P &= (\lambda x.\neg(x\ x))(\lambda x.\neg(x\ x)) \\ &=_\beta \neg((\lambda x.\neg(x\ x))(\lambda x.\neg(x\ x))) \\ &= \neg(P) \end{aligned}$$

which is clearly a contradiction.

Types

$$A, B \in \text{Types} ::= A \to B \mid a$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash v : B}{\Gamma \vdash \lambda x.v : A \to B} \qquad \frac{\Gamma \vdash v : A \to B \qquad \Gamma \vdash v' : A}{\Gamma \vdash v\ v' : B}$$

FIGURE 2. Assignment of Simple Types for the λ-Calculus

In order to avoid these problems, Church (1940) introduced the simply typed λ-calculus, which we present in Fig. 2. Here, we have an inductively defined inference system delineating what terms are *well-typed* with base type a and function types of the form $A \to B$. The judgment $\Gamma \vdash v : A$ says that v has type A under the assumption Γ. Typing resolves the non-termination problem: β reduction always terminates, and so dubious cases like the paradox above are eliminated. As such, the simply typed lambda calculus can be used as an organizing tool for logic, as has become standard in the HOL family of proof assistants (Gordon (2000)).

However, the simply typed λ-calculus is connected to logic in another way. Forgetting the terms, and looking only at the types, we can interpret $\Gamma \vdash v : A$ as saying A is *true* assuming Γ and then interpret the function type constructor $\rightarrow$ as logical implication. Then, the rules of the simply typed λ-calculus are exactly the rules of the proof system known as *intuitionistic natural deduction* (Howard (1980)). The correspondence between typed calculi and proof systems motivates the approach taken in this dissertation.

Sequent Logic

In the years of 1934 and 1935 Gerhard Gentzen published his "Untersuchungen ber das logische Schließen" (Gentzen (1935)). Gentzen described the system of *natural deduction* for both classical and intuitionistic predicate logics. Natural deduction was as "close as possible to actual reasoning". Further, Gentzen posited his 'Hauptsatz'[1] that every proof in natural deduction could be reduced to a normal form. Although he did not present the details, Gentzen claimed a direct proof of this fact for intuitionistic natural deduction, but could not determine a way to reduce *classical* proofs making use of the law of the excluded middle. For this he introduced a new system called *sequent calculus* and showed the equivalence of classical and intuitionistic sequent calculi to natural deduction as well as to a "Hilbert style" system. For sequent calculus, even in the classical case, Gentzen was able to show his Hauptsatz in the form of "cut elimination." It is hard to overstate the significance of this result: the 'Hauptsatz' is the principle theorem in proof theory. It implies, among other things, that these logics are *internally consistent* in that they are unable to prove the proposition *false*.

[1] "Main Theorem."

The fundamental judgment of sequent calculus is written

$$\Gamma \vdash \Delta$$

where Γ (the assumptions) and Δ (the conclusions) are sets of logical formulae. The intuitive interpretation of a sequent is that "if every formula in Γ is true then at least one formula in Δ is true."

The simplest rule of sequent calculus is the *axiom*, which says that any formula entails itself.

$$\frac{}{\Gamma, P \vdash P, \Delta}$$

The ability to work with lemmas is fundamental to mathematics. In sequent calculus, lemmas are supported by way of the cut rule.

$$\frac{\Gamma \vdash P, \Delta \qquad \Gamma, P \vdash \Delta}{\Gamma \vdash \Delta}$$

To define logical connectives in the sequent calculus we must do two things: describe how to prove a sequent which has that connective in its conclusion, and, describe how to prove a sequent which has that connective as an assumption.

The propositional connective we are most interested in is *implication*. We should be able to prove $P \rightarrow Q$ if we can prove Q under the assumption P. The *right introduction rule* for implication encodes this view.

$$\frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \to Q, \Delta}$$

On the other hand, the *left introduction rule* for implication must show us how to prove a sequent of the form $\Gamma, P \to Q \vdash \Delta$. Thinking through this, we know that something in Δ holds under the assumption of $P \to Q$ if something in Δ holds under the assumption Q and if either P or Δ holds. Equivalently, we can think of a left rule as a rule for *refutation.* We can refute $P \to Q$ if we can prove P and refute Q, since, classically $\neg(P \to Q) \equiv \neg(\neg P \vee Q) \equiv P \wedge \neg Q$.

$$\frac{\Gamma, Q \vdash \Delta \qquad \Gamma \vdash P, \Delta}{\Gamma, P \to Q \vdash \Delta}$$

We have given a short semantic justification for the introduction rules for implication, but one still might wonder how can we be sure they are correct? First, we observe that the axiom is *derivable* for the formula $P \to Q$ assuming it holds at each of P and Q.

$$\frac{\dfrac{\dfrac{}{\Gamma, Q, P \vdash Q, \Delta} \qquad \dfrac{}{\Gamma, P \vdash P, Q, \Delta}}{\Gamma, P \to Q, P \vdash Q, \Delta}}{\Gamma, P \to Q \vdash P \to Q, \Delta}$$

The cut rule is also derivable. The ability to remove cuts, *cut elimination,* is the primary correctness property for sequent calculus. Suppose we have a proof that ends in a right introduction of $P \rightarrow Q$ and another proof that ends in a left introduction of the same formula. A cut between these two proofs can be eliminated in so far as it can be reduced to a proof that only makes use of cut at structurally smaller formulae.

$$\dfrac{\dfrac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \rightarrow Q, \Delta} \qquad \dfrac{\Gamma, Q \vdash \Delta \qquad \Gamma \vdash P, \Delta}{\Gamma, P \rightarrow Q \vdash \Delta}}{\Gamma \vdash \Delta}$$

$$\Downarrow$$

$$\dfrac{\Gamma \vdash P, \Delta \qquad \dfrac{\Gamma, P \vdash Q, \Delta \qquad \Gamma, Q \vdash \Delta}{\Gamma, P \vdash \Delta}}{\Gamma \vdash \Delta}$$

The cut elimination procedure works to eliminate all cuts from our proofs. The existence of cut elimination means that if something is provable, it is provable without the use of cuts. This property ensures the consistency of the logic: what can be shown about a connective is exactly what can be shown by the introduction rules. Observe however that cut elimination is inherently nondeterministic. For example, we could give an alternative derivation of cut for implication.

$$\dfrac{\dfrac{\Gamma \vdash P, \Delta \qquad \Gamma, P \vdash Q, \Delta}{\Gamma \vdash Q, \Delta} \qquad \Gamma, Q \vdash \Delta}{\Gamma \vdash \Delta}$$

In addition to the introduction rules for connectives, there are the *structural rules.* We should not have to care about the order of assumptions and conclusions in our sequent, and thus Gentzen gives his rules of exchange.

$$\frac{\Gamma, P, Q, \Gamma' \vdash \Delta}{\Gamma, Q, P, \Gamma' \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, P, Q, \Delta'}{\Gamma \vdash \Delta, Q, P, \Delta'}$$

The weakening rules state that if a sequent is provable, the same sequent is provable with any number of extra assumptions and conclusions.

$$\frac{\Gamma \vdash \Delta}{\Gamma, \Gamma' \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \Delta'}$$

While the rules of contraction state that two assumptions (conclusions) of the same formula can be shared.

$$\frac{\Gamma, P, P \vdash \Delta}{\Gamma, P \vdash \Delta} \qquad \frac{\Gamma \vdash P, P, \Delta}{\Gamma \vdash P, \Delta}$$

Uses of the structural rules are often kept implicit. In particular, because we treat Γ and Δ as *sets* rather than purely formal sequences of formulae, exchange and contraction are completely automatic.

Unlike the single conclusion natural deduction, which is by nature *intuitionistic*, sequent calculus with multiple conclusions corresponds to *classical logic.* This can be seen in the derivation of Peirce's law–$((P \to Q) \to P) \to P$– a statement, which when added as an axiom, has the consequence of turning intuitionistic logic into classical logic.

$$\dfrac{\dfrac{\dfrac{}{P \vdash P} \qquad \dfrac{\dfrac{}{P \vdash P, Q}}{\vdash P \to Q, P}}{(P \to Q) \to P \vdash P}}{\vdash ((P \to Q) \to P) \to P}$$

CHAPTER II

FUNCTIONS AS PATTERN MATCHING

This largely expository chapter incorporates material derived from (Johnson-Freyd et al. (2017)) of which I was the primary author. I would like to thank my coauthors Paul Downen and Zena M. Ariola for their assistance in writing that publication.

Locating Redexes

In order to ever actually run a λ-calculus program we have to rewrite it in a specific order. For example, we can only apply the β rule at the top of a program to rewrite $v\ v'$ into a simpler term if v is already a λ-abstraction. Otherwise, we must first rewrite *inside* v. Indeed, we can never rewrite that top level application until *after* v has been rewritten into a λ-abstraction.

Thus, in order to actually implement λ-calculus, it is valuable to identify where we have to rewrite first. The notion of *evaluation contexts* captures this content (Felleisen and Hieb (1992)), drawing out the top-most left-most possible redex in a program. The idea is to simply define a syntax for the contexts around this redex.

$$E \in \text{Evaluation Contexts} ::= \Box \mid E\ v$$

Here the box, $\Box$, is the "hole" where a possible redex may occur. We can then use the filling notation $E[v]$, which just means E but with v in place of the box, to give an alternative presentation of the β-rule:

$$E[(\lambda x.v)\ v'] =_\beta E[v\{v'/x\}]$$

Abstract Machines

Alternatively, instead of using evaluation contexts, which are syntax for ordinary terms with holes in them, we could switch to an "abstract machine" which stores an evaluation context as a data structure we call a *co-term* (Krivine (2007)). A complete program is then represented as a command consisting of a term, together with a co-term encoding the context around it.

$$e \in \text{Co-terms} ::= \mathsf{tp} \mid v \cdot e$$

$$c \in \text{Command} ::= \langle v \| e \rangle$$

The co-term tp here represents the top-level (empty) context, while $v \cdot e$ could be thought of as a call-stack containing the argument v together with the rest of the call-stack e. In order to compute with this we need a recast β-rule

$$\langle \lambda x.v \| v' \cdot e \rangle =_\beta v\{v'/x\}.$$

However the β-rule alone is definitely not enough. We would also need a rule to refocus (Danvy and Nielsen (2004)) a function application into a call-stack.

$$\langle v\ v' \| e \rangle =_{\texttt{reassoc}} \langle v \| v' \cdot e \rangle$$

Together these rules, oriented as reductions, define the Krivine abstract machine for reducing λ-terms to weak head normal form.

However, they alone don't correspond to the λ-calculus in its full glory. For one thing, even together, they only allow for applying the β-rule to the left-most top-most redex.

However, rewriting programs in this way suggests an interesting possibility. If we look at the reassocation axiom, we see that its meaning has to do not with just manipulating terms, but manipulating machine states. It tells us that the meaning of a function application $v\ v'$ is to rewrite machine state by pushing v' on to the call-stack by extending the prior co-term with $v'\cdot$ and then using this new co-term to evaluate v.

Naming the Call-Stack

Let us generalize this idea (Curien and Munch-Maccagnoni (2010); Munch-Maccagnoni and Scherer (2015)). If co-terms are first class entities, why not give them names? To do so, we only need to introduce a new class of co-variables ranging over co-terms, written with Greek letters, and a new way of forming terms out of commands, $\mu\alpha.c$, together with a single equation

$$\langle \mu\alpha.c \| e \rangle =_{\mu} c\{e/\alpha\}$$

This is analogous to the evaluation of the term `let` $x\ =\ v$ `in` v', where v is substituted for each occurrence of x in v'. In fact, with μ, we don't need function application syntax at all anymore. We can just regard the syntax $v\ v'$ as syntactic sugar for $\mu\alpha.\langle v \| v\cdot\alpha\rangle$. We also don't need the separate notion of a top-level context `tp`, as it can be just another co-variable.

In addition to the output abstraction, it would make sense to include a dual input abstraction $\tilde{\mu}x.c$ which defines a co-term by abstracting over the variable x. The equational property associated with this rule then would be

$$\langle v \| \tilde{\mu}x.c \rangle =_{\tilde{\mu}} c\{v/x\}.$$

With $\tilde{\mu}$ we can simplify our β-rule, such that only the μ and $\tilde{\mu}$ rules do substitution:

$$\langle \lambda x.v \| v' \cdot e \rangle =_\beta \langle v' \| \tilde{\mu} x. \langle v \| e \rangle \rangle$$

which is clearly equivalent (assuming the Barendregt convention is obeyed).

However, we must be wary. Including both μ and $\tilde{\mu}$ rules in a calculus will destroy the equational theory. To see why, consider any two commands c_1 and c_2. Then, we have

$$c_1 =_\mu \langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle =_{\tilde{\mu}} c_2.$$

That is, the equational theory would collapse, relating every program to every other program. To avoid this we will have to either leave out $\tilde{\mu}$ or restrict one or both of the rules in some way. We can do so by restricting what can be substituted to only a subset of terms called "values" (which we will denote with a capital V) and a subset of co-terms called "co-values" (denoted E). However, we have some choice in what we take as (co-)values. If we take all terms to be values, and then use a very restricted set of co-values as just co-variables and call-stacks made up of hereditary (co-)values, we recover the call-by-name parameter passing technique. If we take all co-terms to be co-values and restrict values down to just variables and λ-abstractions, we get the call-by-value system. (Co-)variables only range over (co-)values, and so we can regard both disciplines as answers to the question: "what is a (co-)value?"

Thus the parametric calculus $\bar{\lambda}\mu\tilde{\mu}$ given in Fig. 3 only has λ-abstractions and output abstractions as terms. Moreover, this calculus no longer has the limitation that β can only be applied to the left-most top-most redex.

$$\begin{aligned} c \in \text{Command} &::= \langle v \| e \rangle \\ v \in \text{Terms} &::= x \mid \lambda x.v \mid \mu\alpha.c \\ e \in \text{Co-Terms} &::= \alpha \mid v \cdot e \mid \tilde{\mu}x.c \end{aligned}$$

$$\begin{aligned} \langle \mu\alpha.c \| E \rangle &=_{\mu} c\{E/\alpha\} \\ \langle V \| \tilde{\mu}x.c \rangle &=_{\tilde{\mu}} c\{v/x\} \\ \mu\alpha.\langle v \| \alpha \rangle &=_{\eta_\mu} v \\ \tilde{\mu}x.\langle x \| e \rangle &=_{\eta_{\tilde{\mu}}} e \\ \langle \lambda x.v' \| v \cdot e \rangle &=_{\beta} \langle v \| \tilde{\mu}x.\langle v' \| e \rangle \rangle \end{aligned}$$

FIGURE 3. The parametric $\bar{\lambda}\mu\tilde{\mu}$-calculus

$$\begin{aligned} [\![\lambda x.v]\!] &= \lambda x.[\![v]\!] \\ [\![v\ v']\!] &= \mu\alpha.\langle v \| v' \cdot \alpha \rangle \\ [\![x]\!] &= x \end{aligned}$$

FIGURE 4. Translation from λ-calculus to $\bar{\lambda}\mu\tilde{\mu}$

Adding output abstraction actually allows for writing more programs than could be written in pure λ-calculus. The difference is that now there are extra programs which don't come from any λ-terms. For example, a term which "aborts" by abstracting over the output and then ignoring it (assume α not free in v) is

$$\mu\alpha.\langle v \| \beta \rangle.$$

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta} \qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \| e \rangle : (\Gamma \vdash \Delta)}$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}$$

$$\frac{\Gamma \mid e : B \vdash \Delta \quad \Gamma \vdash v : A \mid \Delta}{\Gamma \mid v \cdot e : A \to B \vdash \Delta} \qquad \frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta}$$

FIGURE 5. Simple types for $\bar{\lambda}\mu\tilde{\mu}$

The difference becomes obvious if we consider a type assignment as we do in Fig. 5. As we can see, our assignment corresponds to a sequent calculus instead of natural deduction. In particular, it is a sequent calculus with both active judgments (typing terms and co-terms) and non-active ones (typing commands), and a rule for activation. Moreover, having multiple conclusions, it is a classical sequent calculus. We can thus write closed terms which correspond to classical proofs, like the proof of Pierce's law (note, we drop extraneous premises to fit the proof on a page):

$$\dfrac{\dfrac{\dfrac{}{\mid \alpha : P \vdash \alpha : P} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : P \vdash x : P \mid} \qquad \dfrac{}{\mid \alpha : P \vdash \alpha : P}}{\langle x \| \alpha \rangle : (x : P \vdash \alpha : P)}}{x : P \vdash \mu_{-}.\langle x \| \alpha \rangle : Q \mid \alpha : P}}{\vdash \lambda x.\mu_{-}.\langle x \| \alpha \rangle : P \to Q \mid \alpha : P}}{\mid (\lambda x.\mu_{-}.\langle x \| \alpha \rangle) \cdot \alpha : (P \to Q) \to P \vdash \alpha : P} \qquad \dfrac{}{f : (P \to Q \to P) \vdash f : (P \to Q \to P) \mid}}{\dfrac{\langle f \| (\lambda x.\mu_{-}.\langle x \| \alpha \rangle) \cdot \alpha \rangle : (f : (P \to Q \to P) \vdash \alpha : P)}{\dfrac{f : (P \to Q) \to P \vdash \mu\alpha.\langle f \| (\lambda x.\mu_{-}.\langle x \| \alpha \rangle) \cdot \alpha \rangle : P \mid}{\vdash \lambda f.\mu\alpha.\langle f \| (\lambda x.\mu_{-}.\langle x \| \alpha \rangle) \cdot \alpha \rangle : ((P \to Q) \to P) \to P \mid}}}$$

The computational difference corresponding to classicality is that the system $\bar{\lambda}\mu\tilde{\mu}$ has *control effects*. Programs such as the proof of Pierce's law above have jumps in them creating non-local control flow. This can actually be quite desirable for writing efficient programs as it allows patterns such as the early exit from a search procedure. Indeed, such jumping constructs were developed prior to the computational interpretation of classical logic and included as a feature in some λ-calculus based languages (e.g. Scheme's `call/cc`).

Let us now reexamine what we have done.

The λ-calculus is one sided, emphasizing only one aspect of computation. However, the calculus $\bar{\lambda}\mu\tilde{\mu}$ reveals computation's two sided nature. Complete programs are written as *commands* which are two sided entities, where the two sides represent two opposing forces of computation—the production (via *terms*) and consumption (via *co-terms*) of information. These correspond in the type system to the two sides of the cut rule from proof theory, and cuts are the only place where computation can happen. For example, a λ-abstraction $\lambda x.v$, as is written in the λ-calculus, is a term that describes the necessary behavior for creating a functional value, whereas a call stack $v \cdot e$ is a co-term that describes the method for using a function. Additionally, we have generic abstractions which allow each side to give a name to the other, regardless of their specific form. The $\tilde{\mu}$-abstraction $\tilde{\mu}x.c$ is a consumer which names its input x before running the command c. Dually, the μ-abstraction $\mu\alpha.c$ is a producer which names the location for its output α before running c. The typing judgment for terms corresponds to the collection of proofs with an *active* formula on the right, and dually, co-terms correspond to proofs with an active formula on the left. Readers unaccustomed to the sequent calculus may find it helpful to think of co-terms as contexts in the λ-calculus and the construct $\langle v \| e \rangle$ as filling the hole in e with the term v. In this view, $\tilde{\mu}x.c$ corresponds roughly to $\texttt{let}\ x = \Box\ \texttt{in}\ c$ while $v \cdot e$ corresponds to the context $e[\Box\ v]$. We can see that $\tilde{\mu}$ allows us to build more complicated compound contexts, beyond just the applicative contexts formed with the call stack constructor (i.e. the $\cdot$ constructor). Similarly, μ-abstraction lets us build more complicated compound terms beyond just λ. Indeed, the μ operator can be seen as arising as a solution to the problem of how to encode elimination forms, and their associated reduction equations, into an abstract machine language like $\bar{\lambda}\mu\tilde{\mu}_\eta$

The μ-abstraction $\mu\alpha.c$ can also be read as $\texttt{call/cc}(\lambda\alpha.c)$, and a co-variable α is akin to a context of the form $\texttt{throw}\ \alpha\ \Box$, which invokes a stored continuation. However, we find the sequent calculus presentation instructive as it emphasizes symmetries, reducing the number of distinct concepts. In a λ-calculus presentation, control operators often come across as exotic and are difficult to build an intuition about. Here, μ-abstraction is simply understood as providing a name to an output channel in a computation: it is no more complicated than $\tilde{\mu}$-abstraction which provides a name to an input.

Replacing Lambda

Even though the μ rule seems very different from the application rule they are actually very similar, when seen side-by-side:

$$\langle \mu\alpha.c \| E\rangle = c\{E/\alpha\}$$

$$\langle \lambda x.v \| v' \cdot E\rangle = \langle v\{v'/x\} \| E\rangle$$

Note that they both inspect the context or co-term. Similar to the way a μ-term corresponds to a let-term, the λ-abstraction corresponds to a term of the form `let` $(x, y) \;=\; v$ `in` v', which decomposes v while naming its sub-parts. So in contrast to the μ-term, the λ-abstraction decomposes the co-term instead of just naming it. To emphasize this view we write a function as a special form of μ-abstraction which *pattern-matches* on the context. The term $\mu[x \cdot \alpha.c]$ gives names to both its argument x and the remainder of its context α in the command c. Operationally, $\mu[x \cdot \alpha.c]$ can be thought of as waiting for a context $v \cdot E$ at which point computation continues in c with v substituted in for x and E substituted in for α. This view emphasizes that a function is given primarily by how it is used rather than how it is defined; the call-stack formation operator $\cdot$ is the most important aspect in the theory of functions (rather than *lambda*-abstraction).

However, the two views of functions are equivalent. We can write the lambda abstraction $\lambda x.v$ as $\mu[x \cdot \alpha.\langle v \| \alpha \rangle]$, given α not free in v. When written in this style β is clearly just a form of pattern matching, only pattern matching on the calling context.

$$\begin{aligned}
c \in \text{Command} &::= \langle v \| e \rangle \\
v \in \text{Terms} &::= x \mu[x \cdot \alpha.c] \mid \mu\alpha.c \\
e \in \text{Co-Terms} &::= \alpha \mid v \cdot e \mid \tilde{\mu}x.c
\end{aligned}$$

$$\begin{aligned}
\langle \mu\alpha.c \| E \rangle &=_{\mu} c\{E/\alpha\} \\
\langle V \| \tilde{\mu}x.c \rangle &=_{\tilde{\mu}} c\{V/x\} \\
\mu\alpha.\langle v \| \alpha \rangle &=_{\eta_\mu} v \\
\tilde{\mu}x.\langle x \| e \rangle &=_{\eta_{\tilde{\mu}}} e \\
\langle \mu[x \cdot \alpha.c] \| v \cdot e \rangle &=_{\beta} \langle v \| \tilde{\mu}x.\langle \mu\alpha.c \| e \rangle \rangle \\
\mu[x \cdot \alpha.\langle f \| x \cdot \alpha \rangle] &=_{\eta} f
\end{aligned}$$

FIGURE 6. The parametric $\mu\tilde{\mu}^{\rightarrow}$-calculus with η

The improved calculus $\mu\tilde{\mu}^{\rightarrow}$ in Fig. 6 takes this view. The type system's connection to sequent calculus is improved too, as we can give new right-introduction for an implication which erases activation,

$$\frac{c : (\Gamma, x : A \vdash \alpha : B, \Delta}{\Gamma \vdash \mu[x \cdot \alpha.c] : A \to B \mid \Delta}$$

and so cleanly differentiates reversible and non-reversible rules in the style of focusing (Andreoli (1992)).

What then about equations? We have already explored a β-rule which does no substitution. With the alternative representation of functions this can instead be given as:

$$\langle \mu[x \cdot \alpha.c] \| v \cdot e \rangle =_{\beta} \langle v \| \tilde{\mu}x.\langle \mu\alpha.c \| e \rangle \rangle.$$

But what about η? In the λ-calculus, η tells us that a function which merely delegates to another function should be equal to the other function. This can be written in a way that works in both call-by-name *and* call-by-value as $f = \lambda x.f x$ for any *variable* f. We should take the same approach in the sequent calculus as a general principle: a pattern match which immediately reconstructs its context to call a variable is equal to that variable:

$$\mu[x \cdot \alpha.\langle f \| x \cdot \alpha\rangle] = f.$$

The system $\mu\tilde{\mu}^{\rightarrow}$, and in particular its call-by-name version, will serve as the main object of study for the next two chapters, as we try to see how to run its programs. Afterwards, we will consider an extended version with additional data types deriving from Downen and Ariola (2014).

CHAPTER III

REASONING WITH LAZYNESS AND EXTENSIONALITY

This chapter is a revised and extended version of (Johnson-Freyd et al. (2017)). I was the primary author of that publication and developed the calculi and detailed proofs appearing here. Paul Downen introduced me to the reflection based technique for showing confluence used in this chapter, while Zena M. Ariola checked my work and found several minor errors in earlier versions of the equational correspondence and reflection proofs. I would like to thank both of them for their invaluable contributions to that publication.

The Challenge of Extensional Rewriting

Extensionality helps make syntactic theories less syntactic—objects that behave the same are the same even though they appear to be different. This is important because we should be able to use a program without knowing how it was written, only how it behaves. We have seen how in the λ-calculus, extensional reasoning is partially captured by the η-law, which says that functional delegation is unobservable: $\lambda x.f\ x\ =\ f$. It is essential to many proofs about functional programs: for example, the well-known "state monad" only obeys the monad laws if η holds (Marlow (2002)). The η-law is compelling because it gives the "maximal" extensionality possible in the untyped λ-calculus: as shown by Böhm (1968), if we attempt to equate any additional β-normal terms beyond β, η, and α, the theory will collapse. Or, put another way, two λ-calculus programs are either the same according to η or there are inputs on which they behave differently.

An equational theory give us a definition of when two programs are the same. But that is all it does. Equational theories don't give us an algorithm to run programs, nor do they provide much in the way of an algorithm to *decide* if two programs are the same: just the conditions for when they are. From the point of view of a compiler then, an equational theory provides a correctness criterion for transformations, but it doesn't tell us what transformations to make. For that, we need to orient the equations into a reduction theory, giving a direction for how to transform programs. With the pure λ-calculus this is easy: we can interpret η as a contraction and β as a function call. Those two reductions give an approach suitable for both running programs to completion and for rewriting programs to become better ones.

Even a reduction theory though isn't necessarily enough. What do we do when multiple reduction rules are available? What if there is both a β and an η Which one do we take? This question is particularly pertinent for a compiler: the compiler will frequently have a choice of rewrites it can perform, how should it decide which to take? One way we might be able to at least partially side step that question is if we use a reduction theory which is confluent. That is, a theory where the chosen order of reductions does not impact the result—all diverging paths will eventually meet back up in the end. In a logic, confluence guarantees consistency: programs which can't be reduced any more are only equated if they are syntactically identical, so we do not end up equating `true` to `false`. Confluence also provides a good vehicle to conduct proofs establishing a standard reduction path, thus demonstrating that the rewriting theory can be implemented in a deterministic fashion. Indeed, the λ-calculus with the extensional η-law is confluent (Barendregt (1984)) which is why all its nice properties hold.

However, all is not well. The pure λ-calculus has these nice properties, but they start to break down the moment additional features are added. In particular, once the calculus is extended with control effects confluence is lost (David and Py (2001)). Even worse, just adding extensional products with a surjectivity law breaks confluence of the pure λ-calculus as well (Klop and de Vrijer (1989)).

Given these troublesome problems, we might be inclined to conclude that confluence is too syntactic and low-level. After all, the consistency of a logic can be demonstrated using alternative techniques (Klop and de Vrijer (1989)), and if we are only interested in capturing program execution, we can focus directly on an operational semantics instead of starting with such a flexible rewriting system. These are sensible arguments, and indeed there are cases where more abstract notions of confluence are needed, as in a calculus with cyclic structures (Ariola and Blom (2002)). However, it might come as a surprise that extensionality creates issues even when focusing on the pure λ-calculus alone without any extensions. For example, continuation-passing style transformations hard-wire the evaluation strategy of a language directly in the program itself. Naturally, we expect that equivalent source programs continue to be equivalent after the transformation. But under a known call-by-name continuation-passing style transformation due to Plotkin (1975), β-equivalence is preserved while η-equivalence is violated.

Since these issues surrounding extensionality show up in multiple disconnected places, we wonder if they act as the canary in a coal mine, signaling that the pieces of the calculus do not quite fit perfectly together and that we should question our basic assumptions. Indeed, both the efforts to solve the confluence problem and the search for a sound continuation-passing style transformation have made use of the same insight about the nature of lazy functions. The goal of this chapter is to clarify this alternative and fundamental view of lazy functions incrementally, using the call-by-name sequent calculus and its core lessons to derive the solution in a principled way. That is, we will come up with a reduction theory for the untyped functional fragment of the call-by-name $\mu\tilde{\mu}_\eta$-calculus. That reduction theory has to correspond to the equational theory presented in the previous chapter, but since a reduction theory is a more fine grained instrument than an equational theory we usually have some freedom in exactly how we define it, and here we use that freedom for good in solving the confluence problem and in bringing out the essence of lazy functions.

The central idea which in hindsight can be recognized in at least Danos, Joinet, and Schellinx (1997), is that functions are co-data (Downen and Ariola (2014); Zeilberger (2009)). They are not concretely constructed objects like tuples in an ML-like language, but rather abstract objects that only respond to messages. The co-data nature of functions suggests a shift in focus away from functions themselves and toward the *observations* that use them. Those observations being defined around the concretely constructed call-stacks in $\mu\tilde{\mu}_\eta^{\rightarrow}$.

Another benefit using the sequent calculus is that control is inherent in the framework rather than being added as an afterthought. But what does control have to do with functions? The two seem unrelated because we are accustomed to associating control with operators like Scheme's `call/cc`, and indeed if we assume that control is just for expressing `call/cc` then there is little justification for using it to explain functions. Instead, control should be seen as a way to give a name to the call stack, similar to the way we name values (Curien and Munch-Maccagnoni (2010)). Thus, much as we match on the structure of a tuple, functions are objects that match on the structure of a call stack. Note that this view of functions as call-stack destructors is independent of the evaluation strategy. It is not tied to call-by-name but is inherent to the primordial notion of a function itself, so it holds just as well in a call-by-value setting (Zeilberger (2009)).

What, then, is the essence of a lazy function? The essential ingredient is that lazy functions perform a lazy pattern match on their call stack, so functional extensionality is captured by the surjectivity of the call stack as a pairing construct. Indeed, this same insight has sprung up as the key linchpin to the problems surrounding the implementation of the η-law. An alternative call-by-name continuation-passing translation based on pairs arises from a polarized translation of classical logic (Girard (1991); Lafont, Reus, and Streicher (1993)). Moreover, at least as long as the pairs in the target are categorical products, that is, surjective pairs, this translation validates the η-law (Fujita (2003); Hofmann and Streicher (2002)). Surjective call stacks have also been used to establish an extensional confluent rewriting system in two independently developed calculi: the $\Lambda\mu$-calculus (Nakazawa and Nagai (2014)), which is interesting for its connections with Böhm separability (Saurin (2010)) and delimited control (Herbelin and Ghilezan (2008)); and the stack calculus, which can be seen as the *dual* to natural deduction (Carraro, Ehrhard, and Salibra (2012)). It is surprising to see that these different problem domains unveil the same key insight about lazy functions.

The main technical contribution of this chapter is in leveraging this view of lazy functions to get confluence for our larger calculus and not just the kernel stack calculus. Along the way we recast the existing proofs in new light and develop a general method for lifting confluence from a kernel calculus to a larger system.

In the end, we have a satisfying solution to confluence in the λ-calculus with both extensionality and classical control effects as well as a sound call-by-name continuation-passing style transformation. This immediately invites the question: does the solution scale up to more realistic calculi with other structures, like products and sums, found in programming languages? Unfortunately, the answer is a frustrating “no.” The solution does not easily apply to programming constructs like sum types, which involve decision points with multiple branches. Worse, confluence appears to break down again when we combine both data and co-data. However, since the use of surjective call stacks works so well for functions in isolation, we see these roadblocks as prompts for a new avenue of study into extensional rewriting systems for other programming structures.

An Extensional Call-By-Name Equational Theory

$$
\begin{aligned}
c \in \text{Command} &::= \langle v \| e \rangle \\
v \in \text{Terms} &::= x \mid \lambda x.v \mid \mu\alpha.c \\
e \in \text{Co-Terms} &::= \alpha \mid v \cdot e \mid \tilde{\mu}x.c \\
E \in \text{Co-Values} &::= \alpha \mid v \cdot E
\end{aligned}
$$

$$
\begin{aligned}
\langle \mu\alpha.c \| E \rangle &=_{\mu} c\{E/\alpha\} \\
\langle v \| \tilde{\mu}x.c \rangle &=_{\tilde{\mu}} c\{v/x\} \\
\mu\alpha.\langle v \| \alpha \rangle &=_{\eta_\mu} v \\
\tilde{\mu}x.\langle x \| e \rangle &=_{\eta_{\tilde{\mu}}} e \\
\langle \lambda x.v' \| v \cdot e \rangle &=_{\beta} \langle v \| \tilde{\mu}x.\langle v' \| e \rangle \rangle \\
\lambda x.\mu\alpha.\langle v \| x \cdot \alpha \rangle &=_{\eta} v
\end{aligned}
$$

FIGURE 7. The call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus extended with η ($\bar{\lambda}\mu\tilde{\mu}_\eta$)

As a warm up, let us make formal the idea from the prior chapter of functions as pattern matching on their call stacks. We can do this by formally defining untyped calculi for functions in both λ and pattern matching styles, and then, proving these calculi to be equivalent in an appropriate sense.

$$\begin{aligned}
c \in \text{Command} &::= \langle v \| e \rangle \\
v \in \text{Terms} &::= x \mid \mu[(x \cdot \alpha).c] \mid \mu\alpha.c \\
e \in \text{Co-Terms} &::= \alpha \mid v \cdot e \mid \tilde{\mu}x.c \\
E \in \text{Co-Values} &::= \alpha \mid v \cdot E
\end{aligned}$$

$$\begin{aligned}
\langle \mu\alpha.c \| E \rangle &=_{\mu} c\{E/\alpha\} \\
\langle v \| \tilde{\mu}x.c \rangle &=_{\tilde{\mu}} c\{v/x\} \\
\mu\alpha.\langle v \| \alpha \rangle &=_{\eta_\mu} v \\
\tilde{\mu}x.\langle x \| e \rangle &=_{\eta_{\tilde{\mu}}} e \\
\langle \mu[(x \cdot \alpha).c] \| v \cdot e \rangle &=_{\beta} \langle v \| \tilde{\mu}x.\langle \mu\alpha.c \| e \rangle \rangle \\
\mu[(x \cdot \alpha).\langle v \| x \cdot \alpha \rangle] &=_{\eta} v
\end{aligned}$$

FIGURE 8. The call-by-name $\mu\tilde{\mu}^{\rightarrow}_{\eta}$-calculus

First, we define a calculus using λ but in the style of $\bar{\lambda}\mu\tilde{\mu}_\eta$ (Figure 7). Second, we adopt the alternative notation for functions given in the first chapter (see Figure 8). A function is not constructed; instead it is a procedure that reacts by pattern matching on its observing call stack. As such, we write a function as $\mu[(x \cdot \alpha).c]$, which says that the context is decomposed into an argument named x and a place named α for the result. Indeed, either representation for functions can be seen as syntactic sugar for the other:

$$\mu[(x \cdot \alpha).c] \triangleq \lambda x.\mu\alpha.c \qquad \text{or} \qquad \lambda x.v \triangleq \mu[(x \cdot \alpha).\langle v \| \alpha \rangle] \tag{3.1}$$

From a logical standpoint, this alternative interpretation of functions corresponds to the right introduction rule for implication, which does not preserve activation on the right.

In order to make the equivalence between these two different views of functions more precise, we utilize the concept of an equational correspondence from Sabry and Felleisen (1993). To distinguish the different equational theories, we use the notation $T \vdash t = t'$ to indicate that t and t' are equated by the theory T. Given two equational theories S and T, the translations $\flat : S \to T$ and $\sharp : T \to S$ form an *equational correspondence* whenever the following four conditions hold:

1. ($\flat$) For all terms s_1 and s_2 of S, $S \vdash s_1 = s_2$ implies $T \vdash s_1^\flat = s_2^\flat$.

2. ($\sharp$) For all terms t_1 and t_2 of T, $T \vdash t_1 = t_2$ implies $S \vdash t_1^\sharp = t_2^\sharp$.

3. ($\flat\sharp$) For all terms s of S, $S \vdash s = (s^\flat)^\sharp$.

4. ($\sharp\flat$) For all terms t of T, $T \vdash t = (t^\sharp)^\flat$.

Proposition 1. *$\bar{\lambda}\mu\tilde{\mu}_\eta$ and $\mu\tilde{\mu}_\eta^{\rightarrow}$ are in equational correspondence.*

Proof. The translations of the equational correspondence are given by the macro definitions between the two syntactic representations of functions (Equation 3.1). The translation is not a syntactic isomorphism, because $\lambda x.v$ in $\bar{\lambda}\mu\tilde{\mu}_\eta$ becomes $\mu[(x\cdot \alpha).\langle v \| \alpha\rangle]$ in $\mu\tilde{\mu}_\eta^{\rightarrow}$, which in turn translates back to $\bar{\lambda}\mu\tilde{\mu}_\eta$ as $\lambda x.\mu\alpha.\langle v \| \alpha\rangle$. However, we have

$$\lambda x.v =_{\eta_\mu} \lambda x.\mu\alpha.\langle v \| \alpha\rangle$$

in $\bar{\lambda}\mu\tilde{\mu}_\eta$. Similarly, $\mu[(x \cdot \alpha).c]$ in $\mu\tilde{\mu}_\eta^{\rightarrow}$ becomes $\lambda x.\mu\alpha.c$, which roundtrips back to the term $\mu[(x \cdot \beta).\langle \mu\alpha.c \| \beta\rangle]$, but

$$\mu[(x \cdot \alpha).c] =_\alpha \mu[(x \cdot \beta).c\{\beta/\alpha\}] =_\mu \mu[(x \cdot \beta).\langle \mu\alpha.c \| \beta\rangle] \ .$$

All other syntactic constructs roundtrip to themselves exactly. The equational correspondence is completed by observing that all axioms of $\bar{\lambda}\mu\tilde{\mu}_\eta$ are derivable in $\mu\tilde{\mu}_\eta^{\rightarrow}$ after translation, and vice versa. □

An Extensional Call-By-Name Reduction Theory

Having seen the equational theory for the $\mu\tilde{\mu}^{\rightarrow}_{\eta}$-calculus, we now look for a corresponding confluent reduction theory. The simplest starting point is to take the left-to-right reading of each axiom as a reduction. However, this system lacks confluence due to a conflict between the η- and μ-rules:

$$\mu\delta.\langle y \| \beta\rangle \leftarrow_{\eta} \mu[(x\cdot\alpha).\langle \mu\delta.\langle y\|\beta\rangle \| x\cdot\alpha\rangle] \rightarrow_{\mu} \mu[(x\cdot\alpha).\langle y\|\beta\rangle] \tag{3.2}$$

This issue is not caused by the two-sided sequent calculus presentation; it is part of a fundamental conflict between lazy functions and control. Indeed, a similar issue occurs in the $\lambda\mu$-calculus (Parigot (1992)), a language with control based on the λ-calculus (David and Py (2001)); given a term of the form $\lambda x.M\ x$, a reduction in $M\ x$ could ruin the pattern for the η-rule. This phenomenon does not occur in plain λ-calculus since both reducts are the same. To combat this issue, David and Py introduce a new rule, written in $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ as

$$\mu\delta.c \rightarrow_{\nu} \mu[(x\cdot\alpha).c\{x\cdot\alpha/\delta\}]$$

The above diverging diagram can thus be brought back together:

$$\mu\delta.\langle y\|\beta\rangle \rightarrow_{\nu} \mu[(x\cdot\alpha).\langle y\|\beta\rangle]$$

The ν rule can be understood as performing not an *η-reduction* but rather an *η-expansion* followed by a μ-reduction.

$$\mu\delta.c \leftarrow_{\eta} \mu[(x\cdot\alpha).\langle \mu\delta.c\|x\cdot\alpha\rangle] \rightarrow_{\mu} \mu[(x\cdot\alpha).c\{x\cdot\alpha/\delta\}]$$

Because ν involves expansion, it risks eliminating the concept of (finite) normal forms. Moreover, confluence is not restored for arbitrary open terms, since terms with a free co-variable can still lack confluence:

$$\langle y \| \beta \rangle \leftarrow_{\mu} \langle \mu\delta.\langle y\|\beta\rangle \| \delta \rangle \leftarrow_{\eta} \langle \mu[(x \cdot \alpha).\langle \mu\delta.\langle y\|\beta\rangle \| x \cdot \alpha \rangle] \| \delta \rangle \rightarrow_{\mu} \langle \mu[(x \cdot \alpha).\langle y \| \beta \rangle] \| \delta \rangle$$

The $\Lambda\mu_{\text{cons}}$-calculus (Nakazawa and Nagai (2014)) provides an interesting alternative solution to the conflict between functional extensionality and control. The $\Lambda\mu_{\text{cons}}$-calculus is an extension of the $\lambda\mu$-calculus (Parigot (1992)) which adds not only an explicit representation of call stacks as co-terms, similar to $\mu\tilde{\mu}^{\rightarrow}_{\eta}$, but also projections *out* of these call stacks: `car` projects out the argument and `cdr` projects out the return continuation. This calculus suggests a third interpretation of functions based on projections, instead of either λ-abstractions or pattern matching. We can transport this alternative interpretation of functions to the sequent calculus by extending $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ with the new term $\texttt{car}(e)$, co-term $\texttt{cdr}(e)$, and co-value $\texttt{cdr}(E)$, giving us the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ reduction theory in Figure 9. Note that to avoid spurious infinite reduction sequences, the ς-rules are restricted to only lift out non-co-values. Effectively, the `exp` rule implements functions as a μ-abstraction with projections out of the given call stack. Thus, although $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ does not have a direct reduction rule corresponding to the β-law, function calls are still derivable:

$$\begin{aligned} \langle \mu[(x \cdot \alpha).c] \| v \cdot E \rangle &\rightarrow_{\texttt{exp}} \langle \mu\beta.c\{\texttt{car}(\beta)/x, \texttt{cdr}(\beta)/\alpha\} \| v \cdot E \rangle \\ &\rightarrow_{\mu} c\{\texttt{car}(v \cdot E)/x, \texttt{cdr}(v \cdot E)/\alpha\} \\ &\rightarrow_{\texttt{cdr}}\rightarrow_{\texttt{car}} c\{v/x, E/\alpha\} \end{aligned}$$

Moreover, even the η-law becomes derivable as a sequence of reductions:

$$\mu[(x \cdot \alpha).\langle v \| x \cdot \alpha\rangle] \to_{\mathtt{exp}} \mu\beta.\langle v \| \mathtt{car}(\beta) \cdot \mathtt{cdr}(\beta)\rangle \to_{\eta.} \mu\beta.\langle v \| \beta\rangle \to_{\eta_\mu} v$$

Yet even though both μ and η are derivable in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$, the `exp` rule brings our previous critical pairs back together (as usual, $\twoheadrightarrow$ stands for multiple steps of reduction):

$$\langle y \| \beta\rangle \twoheadleftarrow \langle \mu[(x \cdot \alpha).\langle \mu\delta.\langle y \| \beta\rangle \| x \cdot \alpha\rangle] \| \delta\rangle \to_{\mu} \langle \mu[(x \cdot \alpha).\langle y \| \beta\rangle] \| \delta\rangle \to_{\mathtt{exp}} \langle \mu\gamma.\langle y \| \beta\rangle \| \delta\rangle \to_{\mu} \langle y \| \beta\rangle$$

From a logical standpoint, the new forms $\mathtt{car}(-)$ and $\mathtt{cdr}(-)$ correspond to *elimination* rules for implication (Figure 10). Note that the $\mathtt{cdr}(-)$ rule is a *left* elimination rule.

Before showing that the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ reduction theory is indeed confluent we need to demonstrate that its associated equational theory—obtained as the symmetric, transitive and reflexive closure of its reductions—is equivalent to $\mu\tilde{\mu}^{\rightarrow}_{\eta}$.

Remark 1. First-class control aside, η presents other challenges, as witnessed by the fact that it took several years before finding a continuation-passing style (CPS) transformation (written as $[\![\cdot]\!]$) validating it. In fact, according to the original call-by-name transformation in Plotkin (1975), the CPS-transformed term $[\![\lambda x.y\ x]\!]$, where y is a free variable, is not $\beta\eta$-equal to $[\![y]\!]$ despite the fact that the original λ-calculus terms are η-equivalent. An alternative approach originally presented in Lafont et al. (1993) and proven sound and complete in Hofmann and Streicher (2002) and Fujita (2003) provides a solution which relies on the same key idea as the one described here to solve a completely different problem. The two main ingredients of their transformation are (1) a continuation is not seen as a black

$$\begin{aligned}
c \in \text{Command} &::= \langle v \| e \rangle \\
v \in \text{Terms} &::= x \mid \mu\alpha.c \mid \mu[(x \cdot \alpha).c] \mid \texttt{car}(e) \\
e \in \text{Co-Terms} &::= \alpha \mid v \cdot e \mid \tilde{\mu}x.c \mid \texttt{cdr}(e) \\
E \in \text{Co-Values} &::= \alpha \mid v \cdot E \mid \texttt{cdr}(E)
\end{aligned}$$

$$\begin{aligned}
\langle \mu\alpha.c \| E \rangle &\to_{\mu} c\{E/\alpha\} \\
\langle v \| \tilde{\mu}x.c \rangle &\to_{\tilde{\mu}} c\{v/x\} \\
\mu\alpha.\langle v \| \alpha \rangle &\to_{\eta_\mu} v \\
\tilde{\mu}x.\langle x \| e \rangle &\to_{\eta_{\tilde{\mu}}} e \\
\texttt{car}(E) \cdot \texttt{cdr}(E) &\to_{\eta.} E \\
\texttt{car}(v \cdot E) &\to_{\texttt{car}} v \\
\texttt{cdr}(v \cdot E) &\to_{\texttt{cdr}} E \\
\texttt{car}(e) &\to_{\varsigma_{\texttt{car}}} \mu\alpha.\langle \mu\beta.\langle \texttt{car}(\beta) \| \alpha \rangle \| e \rangle && e \notin \text{Co-Values} \\
\texttt{cdr}(e) &\to_{\varsigma_{\texttt{cdr}}} \tilde{\mu}x.\langle \mu\alpha.\langle x \| \texttt{cdr}(\alpha) \rangle \| e \rangle && e \notin \text{Co-Values} \\
v \cdot e &\to_{\varsigma.} \tilde{\mu}x.\langle \mu\alpha.\langle x \| v \cdot \alpha \rangle \| e \rangle && e \notin \text{Co-Values} \\
\mu[(x \cdot \alpha).c] &\to_{\texttt{exp}} \mu\beta.c\{\texttt{car}(\beta)/x, \texttt{cdr}(\beta)/\alpha\}
\end{aligned}$$

FIGURE 9. The $\mu\tilde{\mu}^{\to}_{\text{cons}}$ reduction theory

$$\frac{\Gamma \mid e : A \to B \vdash \Delta}{\Gamma \vdash \texttt{car}(e) : A \mid \Delta}[\to Elim_1] \qquad \frac{\Gamma \mid e : A \to B \vdash \Delta}{\Gamma \mid \texttt{cdr}(e) : B \vdash \Delta}[\to Elim_2]$$

FIGURE 10. Additional typing rules for $\mu\tilde{\mu}^{\to}_{\text{cons}}$

box function but as a structure that can be analyzed and taken apart, and (2) an extensional call-by-name function does not need to wait for the continuation to provide its argument but can lazily take it apart when it wants its components. Intuitively, the first point has been embraced by the sequent calculus, which gives a first-class status to the context as a scrutable structure and the second point is captured by the `exp` law. We can elaborate more on these points by relating how $\lambda x.y\ x$ and y are equated by the CPS transformation and by the $\mu\tilde{\mu}^{\to}_{\text{cons}}$ equational theory. Following the transformation given in Figure 11, we have:

$$\begin{aligned}
[\![x]\!] &\triangleq x \\
[\![\lambda x.M]\!] &\triangleq \lambda k.(\lambda x.[\![M]\!])(\pi_1 k)(\pi_2 k) \\
[\![M\ N]\!] &\triangleq \lambda k.[\![M]\!]([\![N]\!], k)
\end{aligned}$$

FIGURE 11. Product-based CPS for λ-calculus

$$[\![\lambda x.y\ x]\!]k \triangleq (\lambda x.[\![y\ x]\!])(\pi_1 k)(\pi_2 k) \triangleq (\lambda x.\lambda q.y\ (x, q))(\pi_1 k)(\pi_2 k)$$

The association of a term and a continuation now becomes a *command*. Notice the similarity between the function's representation in the CPS translation and its representation in the sequent calculus:

$$(\lambda x.\lambda q.y\ (x, q)) \qquad \mu[(x \cdot \alpha).\langle y \| x \cdot \alpha \rangle]$$

The main difference is that instead of taking products as a primitive notion, the sequent calculus recognizes that the product parallels the left constructor of a function. Continuing with the CPS term, we have:

$$\begin{aligned}
[\![\lambda x.y\ x]\!]k &\triangleq (\lambda x.\lambda q.y\ (x, q)))(\pi_1 k)(\pi_2 k) \\
&=_\beta y\ (\pi_1 k, \pi_2 k) \\
&=_\eta y\ k
\end{aligned}$$

So function extensionality is validated by surjective pairing. On the other hand, we have:

$$\begin{aligned}\langle \mu[(x \cdot \alpha).\langle y \| x \cdot \alpha \rangle] \| k \rangle &=_{\texttt{exp}} \langle \mu\beta.\langle y \| \texttt{car}(\beta) \cdot \texttt{cdr}(\beta) \rangle \| k \rangle \\ &=_{\mu} \langle y \| \texttt{car}(k) \cdot \texttt{cdr}(k) \rangle \\ &=_{\eta\cdot} \langle y \| k \rangle\end{aligned}$$

where surjective pairing has been replaced by surjectivity of the call stack.

What is important here is not the syntax: the target language as in Fujita (2003) is free to syntactically express elimination of pairs using pattern matching. What matters is that those pairs be true categorical products equipped with an equational theory equivalent to surjective pairing. In much the same way, the η-law as expressed using pattern matching in $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ corresponds to the surjectivity law in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$. However, while not relevant equationally, we find moving to a projection based presentation informative for both pairs and functions, and in the later case, deeply helpful when it comes to rewriting.

End remark 1.

Bridging Sequent Calculus And Natural Deduction

Since $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ is designed as a sequent calculus counterpart to the $\Lambda\mu_{\text{cons}}$-calculus, they are predictably related to one another. In particular, the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$-calculus is equivalent to the $\lambda\mu_{\text{cons}}$-calculus shown in Figure 12, which syntactically distinguishes between commands and terms, as did the original $\lambda\mu$-calculus (Parigot (1992)). The type assignment for $\lambda\mu_{\text{cons}}$ given in Figure 13 differs from the type system given in Nakazawa and Nagai (2014) due to this syntactic difference,

$$
\begin{aligned}
v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \mid \mu\alpha.c \mid \texttt{car}\ S \\
S \in \text{Streams} &::= \alpha \mid v :: S \mid \texttt{cdr}\ S \\
c \in \text{Commands} &::= [S]v
\end{aligned}
$$

$$
\begin{aligned}
(\mu\alpha.c)\ v &=_{\beta_T} \mu\alpha.c\{v :: \alpha/\alpha\} \\
[S](\mu\alpha.c) &=_{\mu} c\{S/\alpha\} \\
\lambda x.v &=_{\text{exp}} \mu\alpha.[\texttt{cdr}\ \alpha](v\{(\texttt{car}\ \alpha)/x\}) \\
[v' :: S]v &=_{\text{assoc}} [S](v\ v') \\
\texttt{car}\ (v :: S) &=_{\text{car}} v \\
\texttt{cdr}\ (v :: S) &=_{\text{cdr}} S \\
\mu\alpha.[\alpha]v &=_{\eta_\mu} v \\
(\texttt{car}\ S) :: (\texttt{cdr}\ S) &=_{\eta_{::}} S \\
[\texttt{cdr}\ S](v\ (\texttt{car}\ S)) &=_{\eta'_{::}} [S]v
\end{aligned}
$$

FIGURE 12. The $\lambda\mu_{\text{cons}}$ calculus

making it closer to a traditional logic. In particular, $\lambda\mu_{\text{cons}}$ is typed according to the rules of classical natural deduction extended with left introduction and elimination rules. We can translate between $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ and $\lambda\mu_{\text{cons}}$ based on the standard relationship between the sequent calculus and $\lambda\mu$-calculus (Curien and Herbelin (2000)), as shown in Figure 14. The primary complication in translating from $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ to $\lambda\mu_{\text{cons}}$ is the usual issue that comes up when comparing sequent-based and λ-based languages: the sequent calculus language has additional syntactic categories that must be merged together in a λ-calculus language. In particular, the commands and terms of the two calculi are in correspondence, as are the co-values of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ and streams of $\lambda\mu_{\text{cons}}$ (as shown by the auxiliary translation E^{-}). However, even though co-values and streams correspond, the syntactic treatment of general co-terms in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ is absent in $\lambda\mu_{\text{cons}}$. For example, $\texttt{cdr}(\tilde{\mu}x.c)$ cannot be represented directly as a stream in $\lambda\mu_{\text{cons}}$.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta}[id_R] \qquad \frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}[id_L]$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}[Act] \qquad \frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \mid S : A \vdash \Delta}{[S]v : (\Gamma \vdash \Delta)}[cut]$$

$$\frac{\Gamma \vdash v : A \to B \mid \Delta \qquad \Gamma \vdash v' : A \mid \Delta}{\Gamma \vdash v\ v' : B \mid \Delta}[Modus\ Ponens]$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta}[\to_R] \qquad \frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \mid S : B \vdash \Delta}{\Gamma \mid v :: S : A \to B \vdash \Delta}[\to_L]$$

$$\frac{\Gamma \mid S : A \to B \vdash \Delta}{\Gamma \vdash \mathtt{car}\ S : A \mid \Delta}[\to Elim_1] \qquad \frac{\Gamma \mid S : A \to B \vdash \Delta}{\Gamma \mid \mathtt{cdr}\ S : B \vdash \Delta}[\to Elim_2]$$

FIGURE 13. Possible Simple Type Assignment for $\lambda\mu_{\text{cons}}$

To help bridge the gap between the two calculi, we introduce in $\lambda\mu_{\text{cons}}$ the notion of a *context*, denoted by the metavariable C, that is simply a command with a term-shaped hole in it. These contexts in $\lambda\mu_{\text{cons}}$ correspond to co-terms in $\mu\tilde{\mu}^{\to}_{\text{cons}}$. For example, the $\sharp$-translation of the $\mu\tilde{\mu}^{\to}_{\text{cons}}$ call stack $x \cdot \alpha$ becomes the context $[\alpha](\Box\ x)$ in the $\lambda\mu_{\text{cons}}$-calculus, as opposed to the more direct translation as a stream $(x \cdot \alpha)^- = x :: \alpha$. Given any such context C, its translation as a $\mu\tilde{\mu}^{\to}_{\text{cons}}$ co-term is defined as a $\tilde{\mu}$-abstraction:

$$C^\flat \triangleq \tilde{\mu}x.(C[x])^\flat$$

With this additional technical detail for dealing with contexts due to the loss of co-terms, we form an equational correspondence between $\lambda\mu_{\text{cons}}$ and $\mu\tilde{\mu}^{\to}_{\text{cons}}$.

Theorem 1. *$\mu\tilde{\mu}^{\to}_{cons}$ and $\lambda\mu_{cons}$ are in equational correspondence.*

$$(-)^\sharp : \mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \to \lambda\mu_{\text{cons}}$$

$$\langle v \| e \rangle^\sharp \triangleq e^\sharp[v^\sharp]$$

$$\begin{array}{rlrl}
x^\sharp &\triangleq x & \alpha^\sharp &\triangleq [\alpha]\Box \\
(\mu\alpha.c)^\sharp &\triangleq \mu\alpha.(c^\sharp) & (\tilde{\mu}x.c)^\sharp &\triangleq [\delta]((\lambda x.\mu\delta.(c^\sharp))\Box) \\
\mu[(x\cdot\alpha).c]^\sharp &\triangleq \lambda x.\mu\alpha.(c^\sharp) & (v \cdot e)^\sharp &\triangleq e^\sharp[\Box\ v^\sharp] \\
\mathtt{car}(e)^\sharp &\triangleq \mu\alpha.e^\sharp[\mu\beta.[\alpha](\mathtt{car}\ \beta)] & \mathtt{cdr}(e)^\sharp &\triangleq e^\sharp[\mu\alpha.[\mathtt{cdr}\ \alpha]\Box]
\end{array}$$

$$(-)^- : \text{Co-Values} \to \text{Streams}$$

$$\alpha^- \triangleq \alpha \qquad (v \cdot E)^- \triangleq v^\sharp :: E^- \qquad \mathtt{cdr}(E)^- \triangleq \mathtt{cdr}(E^-)$$

$$(-)^\flat : \lambda\mu_{\text{cons}} \to \mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$$

$$\begin{array}{rlrlrl}
([S]v)^\flat &\triangleq \langle v^\flat \| S^\flat \rangle & (\lambda x.v)^\flat &\triangleq \mu[(x\cdot\alpha).\langle v^\flat \| \alpha \rangle] & \alpha^\flat &\triangleq \alpha \\
x^\flat &\triangleq x & (v_1\ v_2)^\flat &\triangleq \mu\alpha.\langle v_1^\flat \| v_2^\flat \cdot \alpha \rangle & (v :: S)^\flat &\triangleq v^\flat \cdot S^\flat \\
(\mu\alpha.c)^\flat &\triangleq \mu\alpha.c^\flat & (\mathtt{car}\ S)^\flat &\triangleq \mathtt{car}(S^\flat) & (\mathtt{cdr}\ S)^\flat &\triangleq \mathtt{cdr}(S^\flat)
\end{array}$$

FIGURE 14. Translations from $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ to $\lambda\mu_{\text{cons}}$ and vice versa

We build up to the proof in stages. The equational correspondence between $\lambda\mu_{\text{cons}}$ and $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ is between the four different syntactic categories of the two calculi according to the following translations:

1. $\lambda\mu_{\text{cons}}$ commands correspond to $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ commands by $c^\flat$ and $c^\sharp$,

2. $\lambda\mu_{\text{cons}}$ terms correspond to $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ terms by $v^\flat$ and $v^\sharp$,

3. $\lambda\mu_{\text{cons}}$ streams correspond to $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ co-values by $S^\flat$ and E^-, and

4. $\lambda\mu_{\text{cons}}$ contexts correspond to $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ co-terms by $C^\flat$ and $e^\sharp$.

To establish the equational correspondence, we must show that all translations preserve all equalities, and the roundtrip translation of every expression is the same as the original expression up to the respective equational theory. However, the fact that we translate between co-terms and contexts means that we need to be careful about when contexts are equated. We say that any two such contexts are equal if and only if they are equal commands for every possible filling. More formally, given any two contexts C and C', $\lambda\mu_{\text{cons}} \vdash C = C'$ if and only if for all $\lambda\mu_{\text{cons}}$ terms v, $\lambda\mu_{\text{cons}} \vdash C[v] = C'[v]$. As a lighter notation for equating two contexts, we will denote the universally quantified term of the equality by $\Box$, which is just another metavariable for terms when used for this particular purpose, and write C as shorthand for $C[\Box]$.

Preservation of Equality

To begin the correspondence, we consider that the translations preserve equalities. Observe that the $(-)^\flat$, $(-)^\sharp$, and $(-)^-$ translations, as given in Figure 14, are compositional. Therefore, it suffices to show that the axioms of $\lambda\mu_{\text{cons}}$ and $\mu\tilde{\mu}^{\to}_{\text{cons}}$ are preserved by each translation. Besides the fact that substitution commutes with translation in either direction, the key property needed is that E^- is a valid interpretation of co-values as streams according to $\sharp$-translation.

Lemma 1. *For all co-values E, $\lambda\mu_{cons} \vdash E^\sharp = [E^-]\Box$.*

Proof. By induction on E. The case for $\texttt{cdr}(E)$ requires the μ rule and the case for $v \cdot E$ requires the assoc rule. With this fact, the interderivability of the $\lambda\mu_{\text{cons}}$ and $\mu\tilde{\mu}^{\to}_{\text{cons}}$ axioms follows by routine calculation.

–

$$\begin{aligned} \alpha^\sharp &\triangleq [\alpha]\Box \\ &\triangleq [\alpha^-]\Box \end{aligned}$$

–

$$\begin{aligned} (v \cdot E)^\sharp &\triangleq E^\sharp[\Box\ v^\sharp] & \\ &= [E^-](\Box\ v^\sharp) & \text{Inductive Hypothesis} \\ &= [v^\sharp :: E^-]\Box & \text{assoc} \\ &\triangleq [(v \cdot E)^-]\Box & \end{aligned}$$

–

$$\begin{aligned} \mathrm{cdr}(E)^\sharp &= E^\sharp[\mu\alpha.[\mathrm{cdr}\ \alpha]\Box] & \\ &= [E^-](\mu\alpha.[\mathrm{cdr}\ \alpha]\Box) & \text{Inductive Hypothesis} \\ &= [\mathrm{cdr}\ E^-]\Box & \mu \\ &\triangleq [\mathrm{cdr}(E)^-]\Box & \end{aligned}$$

□

Next we show that substitution is well behaved.

Lemma 2 (Substitution property for $\flat$)**.** *For any command, term, or co-term t*[1] *of $\lambda\mu_{cons}$, we have that $(t\{v/x\})^\flat \triangleq t^\flat\{v^\flat/x\}$ and $(t\{S/\alpha\})^\flat \triangleq t^\flat\{S^\flat/\alpha\}$.*

[1]Through out this Chapter we use t as a metavariable to range over all syntactic sorts

Proof. $(-)^\flat$ is compositional and so these follow by induction on t. $\square$

Lemma 3 (Substitution property for $\sharp$)**.** *For t being any command, term, or stack of $\mu\tilde{\mu}^{\rightarrow}_{cons}$ then*

1. $(t\{v/x\})^\sharp \triangleq t^\sharp\{v^\sharp/x\}$ *and*
2. $\lambda\mu_{cons} \vdash (t\{E/\alpha\})^\sharp = t^\sharp\{E^-/\alpha\}$

Proof. The first point follows because $^\sharp$ is compositional. The second point follows by induction on t, with the non-immediate case being the base case of the variable α where

$$\begin{aligned}(\alpha[E/\alpha])^\sharp &\triangleq E^\sharp \\ &= [E^-]\square \qquad \text{Lemma 1} \\ &\triangleq ([\alpha]\square)\{E^-/\alpha\} \\ &\triangleq \alpha^\sharp\{E^-/\alpha\}\end{aligned}$$

$\square$

We need to have the β-law in $\lambda\mu_{\text{cons}}$

Lemma 4 (β derivable for $\lambda\mu_{\text{cons}}$)**.** *The following holds for all terms v_1 and v_2*

$$\lambda\mu_{cons} \vdash (\lambda x.v_1)\ v_2 = v_1\{v_2/x\}$$

Proof. By calculation.

$$\begin{aligned}
(\lambda x.v_1)\ v_2 &=_{\eta_\mu} \mu\alpha.[\alpha]((\lambda x.v_1)\ v_2)\\
&=_{\text{assoc}} \mu\alpha.[v_2 :: \alpha](\lambda x.v_1)\\
&=_{\text{exp}} \mu\alpha.[v_2 :: \alpha](\mu\beta.[\texttt{cdr}\ \beta]v_1\{(\texttt{car}\ \beta)/x\}\\
&=_{\mu} \mu\alpha.[\texttt{cdr}\ (v_1 :: \alpha)]v_1\{(\texttt{car}\ (v_2 :: \alpha)/x\}\\
&=_{\texttt{cdr}} \mu\alpha.[\alpha]v_1\{(\texttt{car}\ (v_2 :: \alpha)/x\}\\
&=_{\texttt{car}} \mu\alpha.[\alpha]v_1\{v_2/x\}\\
&=_{\eta_\mu} v_1\{v_2/x\}
\end{aligned}$$

□

With this we can formally prove that the translation preserves equalities.

Lemma 5. *If t_1 and t_2 are both terms, co-terms, or commands in $\mu\tilde{\mu}^{\rightarrow}_{cons}$ and $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash t_1 = t_2$ then $\lambda\mu_{cons} \vdash t_1^\sharp = t_2^\sharp$.*

Proof. The translation is compositional, so we only need to check each axiom of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$. The η_μ-axiom of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ is precisely given by the η_μ axiom of $\lambda\mu_{\text{cons}}$. The remaining axioms each involve some computation.

– μ:

$$\begin{aligned}
\langle \mu\alpha.c \| E\rangle^\sharp &\triangleq E^\sharp[\mu\alpha.c^\sharp] \\
&= [E^-]\mu\alpha.c^\sharp && \text{Lemma 1} \\
&=_\mu c^\sharp\{E^-/\alpha\} \\
&= c\{E/\alpha\}^\sharp && \text{Lemma 3}
\end{aligned}$$

– $\tilde{\mu}$:

$$\begin{aligned}
\langle v \| \tilde{\mu}x.c\rangle^\sharp &\triangleq [\delta]((\lambda x.\mu\delta.c^\sharp)v^\sharp) \\
&= [\delta]\mu\delta.c^\sharp\{v^\sharp/x\} && \text{Lemma 4} \\
&=_\mu c^\sharp\{v^\sharp/x\} \\
&= c\{v/x\}^\sharp && \text{Lemma 3}
\end{aligned}$$

– $\eta_{\tilde{\mu}}$:

$$\begin{aligned}
(\tilde{\mu}x.\langle x \| e\rangle)^\sharp &\triangleq [\delta]((\lambda x.\mu\delta.e^\sharp[x])\ \Box) \\
&= [\delta](\mu\delta.e^\sharp[\Box]) && \text{Lemma 4} \\
&=_\mu e^\sharp[\Box]
\end{aligned}$$

– $\eta.$:

$$\begin{aligned}
(\texttt{car}(E)\cdot\texttt{cdr}(E))^\sharp &= [(\texttt{car}(E)\cdot\texttt{cdr}(E))^-]\Box && \text{Lemma 1}\\
&\triangleq [\texttt{car}\ E^- :: \texttt{cdr}\ E^-]\Box\\
&=_{\eta_{::}} [E^-]\Box\\
&= E^\sharp && \text{Lemma 1}
\end{aligned}$$

– $\texttt{car}$:

$$\begin{aligned}
(\texttt{car}(v\cdot E))^\sharp &\triangleq \mu\alpha.(v\cdot E)^\sharp[\mu\beta.[\alpha]\texttt{car}\ \beta]\\
&= \mu\alpha.[v^\sharp :: E^-]\mu\beta.[\alpha]\mathrm{car}\ \beta && \text{Lemma 1}\\
&=_\mu \mu\alpha.[\alpha]\texttt{car}\ (v^\sharp :: E^-)\\
&=_{\eta_\mu} \texttt{car}\ (v^\sharp :: E^-)\\
&=_{\mathrm{car}} v^\sharp
\end{aligned}$$

– $\texttt{cdr}$:

$$\begin{aligned}
(\texttt{cdr}(v\cdot E))^\sharp &= [\texttt{cdr}(v\cdot E)]^-\Box && \text{Lemma 1}\\
&\triangleq [\texttt{cdr}\ (v^\sharp :: E^-)]\Box\\
&=_{\mathrm{cdr}} [E^-]\Box\\
&= E^\sharp && \text{Lemma 1}
\end{aligned}$$

– exp :

$$\begin{aligned}
(\mu[(x\cdot\alpha).c])^\sharp &\triangleq \lambda x.\mu\alpha.c^\sharp \\
&=_{\text{exp}} \mu\beta.[\texttt{cdr}\ \beta]\mu\alpha.c^\sharp[(\texttt{car}\ \beta)/x] \\
&=_{\mu} \mu\beta.c^\sharp\{(\texttt{car}\ \beta)/x,(\texttt{cdr}\ \beta)/\alpha\} \\
&\triangleq \mu\beta.c^\sharp\{(\texttt{car}(\beta))^\sharp/x,(\texttt{cdr}(\beta))^-/\alpha\} \\
&= \mu\beta.(c\{\texttt{car}(\beta)/x,\texttt{cdr}(\beta)/\alpha\})^\sharp && \text{Lemma 3} \\
&\triangleq (\mu\beta.c\{\texttt{car}(\beta)/x,\texttt{cdr}(\beta)/\alpha\})^\sharp
\end{aligned}$$

– ς.

$$\begin{aligned}
(v\cdot e)^\sharp &\triangleq e^\sharp[\Box\ v^\sharp] \\
&=_{\eta_\mu} e^\sharp[\mu\alpha.[\alpha](\Box\ v^\sharp)] \\
&=_{\mu} [\delta]\mu\delta.e^\sharp[\mu\alpha.[\alpha](\Box\ v^\sharp)] \\
&= [\delta]((\lambda x.\mu\delta.e^\sharp[\mu\alpha.[\alpha](x\ v^\sharp)])\Box) && \text{Lemma 4} \\
&\triangleq (\tilde\mu x.\langle\mu\alpha.\langle x\|v\cdot\alpha\rangle\|e\rangle)^\sharp
\end{aligned}$$

– $\varsigma_{\texttt{cdr}}$

$$\begin{aligned}
\texttt{cdr}(e)^\sharp &\triangleq e^\sharp[(\mu\beta.[\texttt{cdr}\ \beta]\Box)] \\
&=_{\eta_\mu} e^\sharp[\mu\alpha.[\alpha](\mu\beta.[\texttt{cdr}\ \beta]\Box)] \\
&=_{\mu} [\delta](\mu\delta.e^\sharp[\mu\alpha.[\alpha](\mu\beta.[\texttt{cdr}\ \beta]\Box)]) \\
&= [\delta]((\lambda x.\mu\delta.e^\sharp[\mu\alpha.[\alpha](\mu\beta.[\texttt{cdr}\ \beta]x)])\Box) && \text{Lemma 4} \\
&\triangleq (\tilde\mu x.\langle\mu\alpha.\langle x\|\texttt{cdr}(\alpha)\rangle\|e\rangle)^\sharp
\end{aligned}$$

- $\varsigma_{\texttt{car}}$

$$\begin{aligned}\texttt{car}(e)^\sharp &\triangleq \mu\alpha.e^\sharp[\mu\beta.[\alpha]\texttt{car}\ \beta]\\ &=_\mu \mu\alpha.e^\sharp[\mu\beta.[\beta](\mu\delta.[\alpha]\texttt{car}\ \delta)]\\ &=_\mu \mu\alpha.e^\sharp[\mu\beta.[\alpha]\mu\gamma.[\beta](\mu\delta.[\gamma]\texttt{car}\ \delta)]\\ &\triangleq (\mu\alpha.\langle\mu\beta.\langle\texttt{car}(\beta)\|\alpha\rangle\|e\rangle)^\sharp\end{aligned}$$

□

Lemma 6. *If* $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash E_1 = E_2$ *then* $\lambda\mu_{cons} \vdash E_1^- = E_2^-$

Proof. We first show the general property that for any streams S, S' and (possibly empty) sequence of terms $v_1, v_2, \dots, v_{n-1}, v_n$ we have

$$\begin{gathered}(\lambda\mu_{\text{cons}} \vdash [S](x\ v_1\ v_2\ \dots\ v_{n-1}\ v_n) = [S']x)\\ \Downarrow\\ (\lambda\mu_{\text{cons}} \vdash (v_n :: v_{n-1} :: \dots :: v_2 :: v_1 :: S) = S').\end{gathered}$$

This follows by induction on the length of the derivation of $[S](x\ v_1\ v_2\ \dots\ v_{n-1}\ v_n) = [S']x$. Specifically, the only axioms which can equate a command to $[S](x\ v_1\ v_2\ \dots\ v_{n-1}\ v_n)$ would be

- an equality internal to S or one of the v_i which carries over to $v_n :: v_{n-1} :: \dots :: v_2 :: v_1 :: S$ or

– the assoc axiom (in either direction)

$$[v :: S](x\ v_1\ v_2\ \ldots\ v_{n-1}\ v_n) = [S](x\ v_1\ v_2\ \ldots\ v_{n-1}\ v_n\ v)$$

which directly corresponds to the inductive hypothesis.

Now, suppose $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash E_1 = E_2$, by Lemma 5 we know that $\lambda\mu_{\text{cons}} \vdash E_1^{\sharp} = E_2^{\sharp}$. By Lemma 1 we further know that $\lambda\mu_{\text{cons}} \vdash E_1^{\sharp} = [E_1^{-}]\square$ and that $\lambda\mu_{\text{cons}} \vdash E_2^{\sharp} = [E_2^{-}]\square$. Thus $\lambda\mu_{\text{cons}} \vdash [E_1^{-}]\square = [E_2^{-}]\square$ and so for any term v, $\lambda\mu_{\text{cons}} \vdash [E_1^{-}]v = [E_2^{-}]v$. Specifically, $\lambda\mu_{\text{cons}} \vdash [E_1^{-}]x = [E_2^{-}]x$ which by the general property above means that $\lambda\mu_{\text{cons}} \vdash E_1^{-} = E_2^{-}$. □

Lemma 7.

If t_1 and t_2 are terms, streams, or commands in $\lambda\mu_{cons}$ and $\lambda\mu_{cons} \vdash t_1 = t_2$ then $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash t_1^{\flat} = t_2^{\flat}$

Proof. The translation is compositional so we only need to check each axiom of $\lambda\mu_{\text{cons}}$. The μ-axiom of $\lambda\mu_{\text{cons}}$ translates to the μ-axiom of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ after Lemma 2. The β_T and assoc axioms each also correspond to single uses of the μ-axiom of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$. The car, cdr and η_μ of $\lambda\mu_{\text{cons}}$ correspond exactly to the axioms of the same names in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ and the $\eta_{::}$ axiom is implemented by $\eta_{\cdot}$. That leaves two axioms which require some computation:

– exp:

$$
\begin{aligned}
(\lambda x.v)^\flat &\triangleq \mu[(x \cdot \beta).\langle v^\flat \| \beta\rangle] \\
&=_{\text{exp}} \mu\alpha.\langle v^\flat\{\texttt{car}(\alpha)/x\} \| \texttt{cdr}(\alpha)\rangle \\
&\triangleq \mu\alpha.\langle v^\flat\{(\text{car } \alpha)^\flat/x\} \| \texttt{cdr}(\alpha)\rangle \\
&= \mu\alpha.\langle v\{(\text{car } \alpha)/x\}^\flat \| \texttt{cdr}(\alpha)\rangle && \text{Lemma 2} \\
&\triangleq (\mu\alpha.[\texttt{cdr } \alpha]v\{(\text{car } \alpha)/x\})^\flat
\end{aligned}
$$

– $\eta'_{::}$

$$
\begin{aligned}
([\texttt{cdr } S](v\ (\texttt{car } S)))^\flat &\triangleq \langle \mu\alpha.\langle v^\flat \| \texttt{car}(S^\flat) \cdot \alpha\rangle \| \texttt{cdr}S^\flat\rangle \\
&=_\mu \langle v^\flat \| \texttt{car}(S^\flat) \cdot \texttt{cdr}(S^\flat)\rangle \\
&=_{\eta\cdot} \langle v^\flat \| S^\flat\rangle \\
&\triangleq ([S]v)^\flat
\end{aligned}
$$

□

Roundtrip Equality

To finish the correspondence, we consider the roundtrip translations. First, note that the roundtrip from $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ to $\lambda\mu_{\text{cons}}$ and back is a provable equality. In particular, we will show that the following three properties hold by mutual induction on commands, terms, and co-terms:

1. For all $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ commands c, $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (c^\sharp)^\flat = c$.

2. For all $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ terms v, $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (v^\sharp)^\flat = v$.

3. For all $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ co-terms e and $\lambda\mu_{\text{cons}}$ terms v, $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (e^{\sharp}[v])^{\flat} = \langle v^{\flat} \| e\rangle$.

The third property is generalized from the usual form of roundtrip translation, and additionally expresses the fact that co-terms lost in a context can always be rediscovered no matter what fills them. From the third property, we get the desired roundtrip equality of co-terms: [2]

$$\text{for all } \mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \text{ co-terms } e,\ \mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (e^{\sharp})^{\flat} = e \ .$$

This follows from the translation of contexts in the $\lambda\mu_{\text{cons}}$-calculus into $\tilde{\mu}$-abstractions in the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$-calculus along with the $\eta_{\tilde{\mu}}$ axiom:

$$\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (e^{\sharp})^{\flat} = \tilde{\mu}x.(e^{\sharp}[x])^{\flat} =_3 \tilde{\mu}x.\langle x\|e\rangle =_{\eta_{\tilde{\mu}}} e \ .$$

We can also derive a tighter roundtrip equality for establishing the correspondence between co-values and streams:

$$\text{for all } \mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \text{ co-values } E,\ \mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (E^{-})^{\flat} = E \ .$$

This follows by $\flat$-translating the equality $\lambda\mu_{\text{cons}} \vdash E^{\sharp}[x] = [E^{-}]x$ (where x is not free in E) which can be composed with the instance of co-term roundtrip equality for co-values: $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash \langle x\|E\rangle =_3 (E^{\sharp}[x])^{\flat} = ([E^{-}]x)^{\flat} \triangleq \langle x\|(E^{-})^{\flat}\rangle$. Thus, by the $\eta_{\tilde{\mu}}$ axiom, we have $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash (E^{-})^{\flat} =_{\eta_{\tilde{\mu}}} \tilde{\mu}x.\langle x\|(E^{-})^{\flat}\rangle = \tilde{\mu}x.\langle x\|E\rangle =_{\eta_{\tilde{\mu}}} E$.

In order to make this precise, observe that the axioms of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ were selected based on the needs of a reduction theory: we wanted to avoid spurious loops. However, it is useful to establish some general equations of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$.

[2]Note that we use the syntax, $T \vdash t =_r t'$ to indicate that $T \vdash t = t'$ by the rule named r. Similarly, here we use $=_3$ to indicate that the equality is true for reason of property 3 above.

Lemma 8. *The three lifting rules of $\mu\tilde{\mu}^{\rightarrow}_{cons}$ can be generalized as equations to apply to all co-terms and not just co-values.*

1. $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash \boldsymbol{car}(e) = \mu\alpha.\langle\mu\beta.\langle\boldsymbol{car}(\beta)\|\alpha\rangle\|e\rangle$,

2. $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash \boldsymbol{cdr}(e) = \tilde{\mu}x.\langle\mu\alpha.\langle x\|\boldsymbol{cdr}(\alpha)\rangle\|e\rangle$ *and*

3. $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash v \cdot e = \tilde{\mu}x.\langle\mu\alpha.\langle x\|v\cdot\alpha\rangle\|e\rangle$

Proof. In each case, either e is a co-value or it is not. If it is not a co-value then this is just the ς rule. If it is then the property follows from the μ, $\tilde{\mu}$, $\eta_{\tilde{\mu}}$, and η_μ rules. $\square$

Thus

Lemma 9. 1. *For all $\mu\tilde{\mu}^{\rightarrow}_{cons}$ commands c, $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash (c^\sharp)^\flat = c$.*

2. *For all $\mu\tilde{\mu}^{\rightarrow}_{cons}$ terms v, $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash (v^\sharp)^\flat = v$.*

3. *For all $\mu\tilde{\mu}^{\rightarrow}_{cons}$ co-terms e and $\lambda\mu_{cons}$ terms v, $\mu\tilde{\mu}^{\rightarrow}_{cons} \vdash (e^\sharp[v])^\flat = \langle v^\flat\|e\rangle$.*

Proof. By mutual induction.

–

$$
\begin{aligned}
((\langle v\|e\rangle)^\sharp)^\flat &\triangleq (e^\sharp[v^\sharp])^\flat \\
&= \langle (v^\sharp)^\flat\|e\rangle && \text{Induction hypothesis} \\
&= \langle v\|e\rangle && \text{Induction hypothesis}
\end{aligned}
$$

–

$$\begin{aligned} ((x)^\sharp)^\flat &\triangleq x^\flat \\ &\triangleq x \end{aligned}$$

–

$$\begin{aligned} ((\mu\alpha.c)^\sharp)^\flat &\triangleq (\mu\alpha.(c^\sharp))^\flat \\ &\triangleq \mu\alpha.((c^\sharp)^\flat) \\ &= \mu\alpha.c && \text{Induction hypothesis} \end{aligned}$$

–

$$\begin{aligned} ((\mu[(x\cdot\alpha).c])^\sharp)^\flat &\triangleq (\lambda x.\mu\alpha.c^\sharp)^\flat \\ &\triangleq \mu[(x\cdot\beta).\langle\mu\alpha.(c^\sharp)^\flat\|\beta\rangle] \\ &= \mu[(x\cdot\beta).\langle\mu\alpha.c\|\beta\rangle] && \text{Induction hypothesis} \\ &=_\mu \mu[(x\cdot\alpha).c] \end{aligned}$$

–

$$\begin{aligned}
((\mathrm{car}(e))^\sharp)^\flat &\triangleq (\mu\alpha.e^\sharp[\mu\beta.[\alpha]\mathrm{car}\ \beta])^\flat \\
&\triangleq \mu\alpha.(e^\sharp[\mu\beta.[\alpha]\mathrm{car}\ \beta])^\flat \\
&= \mu\alpha.\langle(\mu\beta.[\alpha]\mathrm{car}\ \beta])^\flat\|e\rangle && \text{Induction hypothesis} \\
&\triangleq \mu\alpha.\langle\mu\beta.\langle\mathrm{car}(\beta)\|\alpha\rangle\|e\rangle \\
&= \mathtt{car}(e) && \text{Lemma 8.1}
\end{aligned}$$

–

$$\begin{aligned}
((v'\cdot e)^\sharp[v])^\flat &\triangleq (e^\sharp[\Box\ (v')^\sharp])[v]^\flat \\
&\triangleq e^\sharp[v\ (v')^\sharp]^\flat \\
&= \langle(v\ (v')^\sharp)^\flat\|e\rangle && \text{Induction hypothesis} \\
&\triangleq \langle\mu\alpha.\langle v^\flat\|(v')^\sharp\cdot\alpha\rangle\|e\rangle \\
&= \langle\mu\alpha.\langle v^\flat\|v'\cdot\alpha\rangle\|e\rangle && \text{Induction hypothesis} \\
&=_{\tilde\mu} \langle v^\flat\|\tilde\mu x.\langle\mu\alpha.\langle x\|v'\cdot\alpha\rangle\|e\rangle\rangle \\
&= \langle v^\flat\|v'\cdot e\rangle && \text{Lemma 8.3}
\end{aligned}$$

–

$$\begin{aligned}
((\alpha)^\sharp[v])^\flat &\triangleq ([\alpha]v)^\flat \\
&= \langle v^\flat\|\alpha\rangle && \text{Induction hypothesis}
\end{aligned}$$

–

$$\begin{aligned}
((\texttt{cdr}(e))^\sharp[v])^\flat &\triangleq (e^\sharp[\mu\alpha.[\texttt{cdr}\ \alpha]\Box])[v]^\flat \\
&\triangleq (e^\sharp[\mu\alpha.[\texttt{cdr}\ \alpha]v])^\flat \\
&= \langle \mu\alpha.\langle v^\flat \| \texttt{cdr}(\alpha)\rangle \| e\rangle && \text{Induction hypothesis} \\
&=_{\tilde\mu} \langle v^\flat \| \tilde\mu x.\langle \mu\alpha.\langle x \| \texttt{cdr}(\alpha)\rangle \| e\rangle\rangle \\
&= \langle v^\flat \| \texttt{cdr}(e)\rangle && \text{Lemma 8.2}
\end{aligned}$$

–

$$\begin{aligned}
((\tilde\mu x.c)^\sharp[v])^\flat &\triangleq ([\gamma]((\lambda x.\mu\gamma.c^\sharp)\ v))^\flat \\
&\triangleq \langle \mu\beta.\langle \mu[(x\cdot\alpha).\langle \mu\gamma.(c^\sharp)^\flat)\|\alpha\rangle] \| v^\flat\cdot\beta\rangle \| \gamma\rangle \\
&= \langle \mu\beta.\langle \mu[(x\cdot\alpha).\langle \mu\gamma.c)\|\alpha\rangle] \| v^\flat\cdot\beta\rangle \| \gamma\rangle && \text{Induction hypothesis} \\
&=_\mu \langle \mu[(x\cdot\alpha).\langle \mu\gamma.c\|\alpha\rangle] \| v^\flat\cdot\gamma\rangle \\
&=_\mu \langle \mu\gamma.c[v^\flat/x] \| \gamma\rangle && \text{Lemma 4} \\
&=_\mu c[v^\flat/x] \\
&=_{\tilde\mu} \langle v^\flat \| \tilde\mu x.c\rangle
\end{aligned}$$

□

Second, we note that the roundtrip from $\lambda\mu_{\text{cons}}$ to $\mu\tilde\mu^{\to}_{\text{cons}}$ and back is a provable equality. In particular, we will show below that the following three properties hold by mutual induction on commands, terms, and streams:

1. For all $\lambda\mu_{\text{cons}}$ commands c, $\lambda\mu_{\text{cons}} \vdash (c^\flat)^\sharp = c$.

2. For all $\lambda\mu_{\text{cons}}$ terms v, $\lambda\mu_{\text{cons}} \vdash (v^\flat)^\sharp = v$.

3. For all $\lambda\mu_{\text{cons}}$ streams S, $\lambda\mu_{\text{cons}} \vdash (S^\flat)^- = S$.

The third property relies on the auxiliary injection of co-values into streams to maintain a tighter roundtrip equality which avoids losing the representation of co-terms into contexts caused by the $\sharp$-translation. However, as a corollary of the third property, we also get the fact that $\flat$ and $\sharp$ are inverses for $\lambda\mu_{\text{cons}}$ streams when they are considered as contexts, up to equality:

$$\text{for all } \lambda\mu_{\text{cons}} \text{ streams } S,\ \lambda\mu_{\text{cons}} \vdash (S^\flat)^\sharp = [S]\Box\ ,$$

which holds by the previously noted fact that

$$\lambda\mu_{\text{cons}} \vdash E^\sharp = [E^-]\Box.$$

Additionally, the first property establishes the fact that $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ co-terms are in exact correspondence to $\lambda\mu_{\text{cons}}$ contexts by their $\flat$-translation as $\tilde{\mu}$-abstractions. Specifically,

$$\text{for all } \lambda\mu_{\text{cons}} \text{ contexts } C,\ \lambda\mu_{\text{cons}} \vdash (C^\flat)^\sharp = C\ ,$$

which is provable by the $\lambda\mu_{\text{cons}}$ equational theory of contexts:

$$\lambda\mu_{\text{cons}} \vdash C^{\flat\sharp} \triangleq (\tilde{\mu}x.(C[x])^\flat)^\sharp \triangleq [\delta]((\lambda x.\mu\delta.(C[x])^{\flat\sharp})\ \Box) =_1 [\delta]((\lambda x.\mu\delta.C[x])\ \Box) = C$$

The full calculation follows

Lemma 10. *1. For all $\lambda\mu_{cons}$ commands c, $\lambda\mu_{cons} \vdash (c^\flat)^\sharp = c$.*

2. For all $\lambda\mu_{cons}$ terms v, $\lambda\mu_{cons} \vdash (v^\flat)^\sharp = v$.

3. For all $\lambda\mu_{cons}$ streams S, $\lambda\mu_{cons} \vdash (S^\flat)^- = S$.

Proof. By mutual induction.

–

$$\begin{aligned}(x^\flat)^\sharp &\triangleq x^\sharp \\ &\triangleq x\end{aligned}$$

–

$$\begin{aligned}((\lambda x.v)^\flat)^\sharp &\triangleq (\mu[(x \cdot \alpha).\langle v^\flat \| \alpha\rangle)^\sharp \\ &\triangleq \lambda x.\mu\alpha.[\alpha](v^\flat)^\sharp \\ &= \lambda x.\mu\alpha.[\alpha]v && \text{Inductive hypothesis} \\ &=_{\eta_\mu} \lambda x.v\end{aligned}$$

–

$$\begin{aligned}((v_1\ v_2)^\flat)^\sharp &\triangleq (\mu\alpha.\langle v_1^\flat \| v_2^\flat \cdot \alpha\rangle)^\sharp \\ &\triangleq \mu\alpha.(v_2^\flat \cdot \alpha)^\sharp[(v_1^\flat)^\sharp] \\ &= \mu\alpha.(v_2^\flat \cdot \alpha)^\sharp[v_1] && \text{Inductive hypothesis} \\ &= \mu\alpha.[(v_2^\flat)^\sharp :: \alpha]v_1 && \text{Lemma 1} \\ &= \mu\alpha.[v_2 \cdot \alpha]v_1 && \text{Inductive hypothesis} \\ &=_{\text{assoc}} \mu\alpha.[\alpha](v_1\ v_2) \\ &=_{\eta_\mu} v_1\ v_2\end{aligned}$$

–

$$\begin{aligned}((\mu\alpha.c)^\flat)^\sharp &\triangleq (\mu\alpha.c^\flat)^\sharp \\ &\triangleq \mu\alpha.(c^\flat)^\sharp \\ &= \mu\alpha.c && \text{Inductive hypothesis}\end{aligned}$$

–

$$\begin{aligned}((\mathrm{car}\ S)^\flat)^\sharp &\triangleq \mathrm{car}(S^\flat)^\sharp \\ &\triangleq \mu\alpha.(S^\flat)^\sharp[\mu\beta.[\alpha]\mathrm{car}\ \beta] \\ &= \mu\alpha.[(S^\flat)^-]\mu\beta.[\alpha]\mathrm{car}\ \beta && \text{Lemma 1} \\ &= \mu\alpha.[S]\mu\beta.[\alpha]\mathrm{car}\ \beta && \text{Inductive hypothesis} \\ &=_\mu \mu\alpha.[\alpha]\mathrm{car}\ S \\ &=_{\eta_\mu} \mathrm{car}\ S\end{aligned}$$

–

$$\begin{aligned}(([S]v)^\flat)^\sharp &\triangleq (\langle v^\flat \| S^\flat\rangle)^\sharp \\ &\triangleq (S^\flat)^\sharp[(v^\flat)^\sharp] \\ &= [(S^\flat)^-](v^\flat)^\sharp && \text{Lemma 1} \\ &= [S]v && \text{Inductive hypothesis}\end{aligned}$$

–

$$\begin{aligned}((\alpha)^\flat)^- &\triangleq \alpha^- \\ &\triangleq \alpha\end{aligned}$$

–

$$\begin{aligned}((v :: S)^\flat)^- &\triangleq (v^\flat \cdot S^\flat)^- \\ &\triangleq (v^\flat)^\sharp :: (S^\flat)^- \\ &= v :: S && \text{Inductive hypothesis}\end{aligned}$$

–

$$\begin{aligned}((\mathrm{cdr}\ S)^\flat)^- &\triangleq \mathrm{cdr}(S^\flat)^- \\ &\triangleq \mathrm{cdr}\ (S^\flat)^- \\ &= \mathrm{cdr}\ S && \text{Inductive hypothesis}\end{aligned}$$

□

Putting these Lemmas together proves our main result of this section.

Theorem 1. *$\mu\tilde{\mu}^{\rightarrow}_{cons}$ and $\lambda\mu_{cons}$ are in equational correspondence.*

Proof. Using the $(-)^\sharp$ and $(-)^\flat$ operations. Preservation of equalities is shown by Lemmas 7, 6 and 5 while the round-trip properties are given by Lemmas 9 and 10. □

Equivalent Views of Functions: Pattern Matching and Projection

The use of car and cdr in function reduction appear so different from the usual treatment of functions that it might come as a surprise. The rules are justified by the previously established call-by-name continuation-passing style transformation using surjective pairs (Hofmann and Streicher (2002)) as well as a stream model (Nakazawa and Nagai (2014)). However, they can also be understood as projection operations defined in terms of pattern matching (Herbelin (2005); Munch-Maccagnoni (2013)) according to the macro expansions

$$\texttt{car}(e) \triangleq \mu\alpha.\langle\mu[(x \cdot \_).\langle x\|\alpha\rangle]\|e\rangle \qquad\qquad \texttt{cdr}(e) \triangleq \tilde{\mu}x.\langle\mu[(\_ \cdot \alpha).\langle x\|\alpha\rangle]\|e\rangle$$

similar to the way that the fst and snd projections out of a tuple can be defined by pattern matching. These definitions give rise to an equational correspondence between the $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ calculus and the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ equational theory.

The only major complication in establishing the correspondence is that the two languages do not quite share the same notion of co-value. In particular, $\texttt{cdr}(E)$ is a co-value in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ but its definition in $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ is not a co-value because cdr expands into a non-trivial $\tilde{\mu}$-abstraction. However, even though $\texttt{cdr}(E)$ is not a syntactic co-value in the $\mu\tilde{\mu}^{\rightarrow}_{\eta}$-calculus, it is still a semantic co-value since it behaves like one in the $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ equational theory. The only axiom of the $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ equational theory that mentions co-values is the μ-axiom, which is derivable for the expansion of $\texttt{cdr}(\alpha)$:

$$\mu\tilde{\mu}^{\rightarrow}_{\eta} \vdash \langle\mu\beta.c\|\texttt{cdr}(\alpha)\rangle = c\{\texttt{cdr}(\alpha)/\beta\} \tag{3.3}$$

This above equality is the lynchpin that lets us bridge the two different notions of co-value, and build an equational correspondence between $\mu\tilde{\mu}_\eta^\to$ and $\mu\tilde{\mu}_{\text{cons}}^\to$.

Before we show it, we observe the following fact about the macro definitions of `cdr` and `car` which is useful and is a consequence of simple substitutions:

Fact 1. – *$\mu\tilde{\mu}_\eta^\to \vdash$ **cdr**$(v \cdot E) = E$, and*

– *$\mu\tilde{\mu}_\eta^\to \vdash$ **car**$(v \cdot E) = v$.*

Proof. By calculation:

–

$$\begin{aligned}
\texttt{cdr}(v \cdot E) &\triangleq \tilde{\mu}x.\langle \mu[(\_ \cdot \alpha).\langle x \| \alpha \rangle] \| v \cdot E \rangle \\
&= \tilde{\mu}x.\langle v \| \tilde{\mu}\_.\langle \mu\alpha.\langle x \| \alpha \rangle \| E \rangle \rangle && \beta \\
&= \langle \mu\alpha.\langle x \| \alpha \rangle \| E \rangle && \tilde{\mu} \\
&= \tilde{\mu}x.\langle x \| E \rangle && \mu \\
&= E && \eta_{\tilde{\mu}}
\end{aligned}$$

–

$$\begin{aligned}
\texttt{car}(v \cdot E) &\triangleq \mu\alpha.\langle \mu[(x \cdot \_).\langle x \| \alpha \rangle] \| v \cdot \rangle \\
&= \mu\alpha.\langle v \| \tilde{\mu}x.\langle E \| \mu\_.\langle x \| \alpha \rangle \rangle \rangle && \beta \\
&= \mu\alpha.\langle E \| \mu\_.\langle v \| \alpha \rangle \rangle && \tilde{\mu} \\
&= \mu\alpha.\langle v \| \alpha \rangle && \mu \\
&= v && \eta_{\mu}
\end{aligned}$$

$$\langle v \| e \rangle^\circ = \langle v^\circ \| e^\circ \rangle$$

$$\begin{array}{rlrl}
\alpha^\circ & \triangleq \alpha & x^\circ & \triangleq x \\
(v \cdot e)^\circ & \triangleq v^\circ \cdot e^\circ & \mu[(x \cdot \alpha).c]^\circ & \triangleq \mu[(x \cdot \alpha).c^\circ] \\
(\tilde{\mu} x.c)^\circ & \triangleq \tilde{\mu} x.(c^\circ) & (\mu\alpha.c)^\circ & \triangleq \mu\alpha.(c^\circ) \\
\mathtt{cdr}(e)^\circ & \triangleq \tilde{\mu} x.\langle \mu[(\_ \cdot \alpha).\langle x \| \alpha \rangle] \| e^\circ \rangle & \mathtt{car}(e)^\circ & \triangleq \mu\alpha.\langle \mu[(x \cdot \_).\langle x \| \alpha \rangle] \| e^\circ \rangle
\end{array}$$

FIGURE 15. Translation from $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ to $\mu\tilde{\mu}^{\rightarrow}_{\eta}$

□

We can now show the main lemma: $\mathtt{cdr}(E)$ is a semantic co-value in $\mu\tilde{\mu}^{\rightarrow}_{\eta}$.

Lemma 11. *For all $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ co-values E, $\mu\tilde{\mu}^{\rightarrow}_{\eta} \vdash \langle \mu\beta.c \| \mathtt{cdr}(E) \rangle = c\{\mathtt{cdr}(E)/\beta\}$*

Proof. It is enough to show that $\mu\tilde{\mu}^{\rightarrow}_{\eta} \vdash \langle \mu\beta.c \| \mathtt{cdr}(\alpha) \rangle = c[\mathtt{cdr}(\alpha)/\beta]$ since then

$$\mu\tilde{\mu}^{\rightarrow}_{\eta} \vdash \langle \mu\beta.c \| \mathtt{cdr}(E) \rangle =_\mu \langle \mu\alpha.\langle \mu\beta.c \| \mathtt{cdr}(\alpha) \rangle \| E \rangle = \langle \mu\alpha.c\{\mathtt{cdr}(\alpha)/\beta\} \| E \rangle =_\mu c\{\mathtt{cdr}(E)/\beta\}$$

$$\begin{aligned}
\langle \mu\beta.c \| \mathtt{cdr}(\alpha) \rangle &= \langle \mu\beta.c \| \tilde{\mu} z.\langle \mu[(x \cdot \gamma).\langle z \| \gamma \rangle] \| \alpha \rangle \rangle \\
&= \langle \mu[(x \cdot \gamma).\langle \mu\beta.c \| \gamma \rangle] \| \alpha \rangle && \tilde{\mu} \\
&= \langle \mu[(x \cdot \gamma).c\{\gamma/\beta\} \| \alpha \rangle && \mu \\
&= \langle \mu[(x \cdot \gamma).c\{\mathtt{cdr}(x \cdot \gamma)/\beta\} \| \alpha \rangle && \text{Fact 1} \\
&= \langle \mu[(x \cdot \gamma).\langle \mu\epsilon.c\{\mathtt{cdr}(\epsilon)/\beta\} \| x \cdot \gamma \rangle \| \alpha \rangle && \mu \\
&= \langle \mu\epsilon.c\{\mathtt{cdr}(\epsilon)/\beta\} \| \alpha \rangle && \eta \\
&= c\{\mathtt{cdr}(\alpha)/\beta\} && \mu
\end{aligned}$$

□

Theorem 2. *$\mu\tilde{\mu}^{\rightarrow}_{\eta}$ and $\mu\tilde{\mu}^{\rightarrow}_{cons}$ are in equational correspondence.*

Proof. The translation from $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ into $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ is syntactic inclusion, and the translation from $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ to $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ is the full macro expansion of `car` and `cdr` as given in Figure 15.

The only significant obstacle in showing that macro expansion maps equalities of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ to equalities of $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ is that the expansion of a $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ co-value, E°, is not always a $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ co-value due to the fact that the expansion of `cdr` is never a co-value. Thus, the μ-axiom of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ does not map directly to the μ-axiom of $\mu\tilde{\mu}^{\rightarrow}_{\eta}$. However, it turns out that every $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ co-value E still behaves like a co-value in $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ as justified by the extended μ-rule:

$$\mu\tilde{\mu}^{\rightarrow}_{\eta} \vdash \langle \mu\alpha.c \| E^\circ \rangle = c\{E^\circ/\alpha\} \tag{3.4}$$

which can be shown by induction on E, (using Lemma 11 in the `cdr` case) and noting that the case of α is trivial:

–

$$\begin{aligned}
\langle \mu\beta.c \| (v \cdot E)^\circ \rangle &\triangleq \langle \mu\beta.c \| v^\circ \cdot E^\circ \rangle \\
&= \langle \mu\alpha.\langle \mu\beta.c \| v^\circ \cdot \alpha \rangle \| E^\circ \rangle && \text{Inductive Hypothesis} \\
&= \langle \mu\alpha.c\{(v^\circ \cdot \alpha)/\beta\} \| E^\circ \rangle && \mu \\
&= c\{(v^\circ \cdot E^\circ)/\beta\} && \text{Inductive Hypothesis}
\end{aligned}$$

–

$$\begin{aligned}
\langle \mu\beta.c \| \mathrm{cdr}(E)^\circ\rangle &\triangleq \langle \mu\beta.c \| \mathrm{cdr}(E^\circ)\rangle \\
&= \langle \mu\alpha.\langle \mu\beta.c \| \mathrm{cdr}(\alpha)\rangle \| E^\circ\rangle && \text{Inductive Hypothesis} \\
&= \langle \mu\alpha.c\{\mathrm{cdr}(\alpha)/\beta\} \| E^\circ\rangle && \text{Lemma 11} \\
&= c\{\mathrm{cdr}(E^\circ)/\beta\} && \text{Inductive Hypothesis}
\end{aligned}$$

With this derived equality, and the fact that the translation is compositional, it is straightforward to check that the equalities of $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ and $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ are interderivable by checking each axiom of each equational theory under translation. More specifically, since many of the axioms are the same, and $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ is a syntactic subset of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$, it suffices to show that the β- and η- axioms of $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ are derivable in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$, and that the `car`, cdr, $\eta.$, `exp`, and ς-family of axioms from $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ are derivable in $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ by their macro expansions.

If $\mu\tilde{\mu}^{\rightarrow}_{\eta} \vdash t = t'$ then $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash t = t'$, where t and t' range over commands, terms and contexts. We only need to check the η- and β-axioms since all other $\mu\tilde{\mu}^{\rightarrow}_{\eta}$ axioms are $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ axioms. For η we have:

$$\begin{aligned}
\mu[(x\cdot\alpha).\langle v \| x\cdot\alpha\rangle] &=_{\texttt{exp}} \mu\beta.\langle v \| \texttt{car}(\beta)\cdot\texttt{cdr}(\beta)\rangle \\
&=_{\eta.} \mu\beta.\langle v \| \beta\rangle \\
&=_{\eta_\mu} v
\end{aligned}$$

The derivability of β follows from the exp rule together with projection operations, the underlying substitution calculus, and the derivability of the unrestricted version of the lifting rules.

$$\begin{aligned}
\langle \mu[(x\cdot\alpha).c] \| v\cdot e\rangle &= \langle \mu\beta.\{\mathrm{car}(\beta)/x, \mathrm{cdr}(\beta)/\alpha\} \| v\cdot e\rangle && \mathrm{exp}\\
&= \langle \mu\beta.c\{\mathrm{car}(\beta)/x, \mathrm{cdr}(\beta)/\alpha\} \| \tilde{\mu}y.\langle \mu\gamma.\langle y \| v\cdot\gamma\rangle \| e\rangle\rangle && \varsigma.\\
&= \langle \mu\gamma.\langle \mu\beta.c\{\mathrm{car}(\beta)/x, \mathrm{cdr}(\beta)/\alpha\} \| v\cdot\gamma\rangle \| e\rangle && \tilde{\mu}\\
&= \langle \mu\gamma.c\{\mathrm{car}(v\cdot\gamma)/x, \mathrm{cdr}(v\cdot\gamma)/\alpha\} \| e\rangle && \mu\\
&= \langle \mu\gamma.c\{v/x, \gamma/\alpha\} \| e\rangle && \texttt{car/cdr}\\
&= \langle \mu\alpha.c\{v/x\} \| e\rangle && \alpha\\
&= \langle v \| \tilde{\mu}x.\langle \mu\alpha.c \| e\rangle\rangle && \tilde{\mu}
\end{aligned}$$

The last part of the equational correspondence is to show that all roundtrip translations are equalities in their respective theories. For the roundtrip from $\mu\tilde{\mu}_{\eta}^{\rightarrow}$ to $\mu\tilde{\mu}_{\mathrm{cons}}^{\rightarrow}$ and back, this is trivial since $\mu\tilde{\mu}_{\eta}^{\rightarrow}$ is included as a syntactic subset of $\mu\tilde{\mu}_{\mathrm{cons}}^{\rightarrow}$ which is unchanged on the translation back to $\mu\tilde{\mu}_{\eta}^{\rightarrow}$. For the roundtrip from $\mu\tilde{\mu}_{\mathrm{cons}}^{\rightarrow}$ to $\mu\tilde{\mu}_{\eta}^{\rightarrow}$ and back, it suffices to observe that the macro expansions of `car` and `cdr` are provable equalities in $\mu\tilde{\mu}_{\mathrm{cons}}^{\rightarrow}$, as all other syntactic constructs roundtrip to themselves exactly.

Specifically

1. If $\mu\tilde{\mu}_{\mathrm{cons}}^{\rightarrow} \vdash t = t'$ then $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash t^{\circ} = (t')^{\circ}$. This follows by checking the axioms. The η_{μ}- and $\eta_{\tilde{\mu}}$-axioms come directly from the equivalent axioms of $\mu\tilde{\mu}_{\eta}^{\rightarrow}$. The $\tilde{\mu}$-axiom works using the $\tilde{\mu}$-axiom of $\mu\tilde{\mu}_{\eta}^{\rightarrow}$ and the compositionality of the $^{\circ}$ translation that gives $c^{\circ}\{v^{\circ}/x\} \triangleq c\{v/x\}^{\circ}$. The μ-axiom works since by Lemma 3.4 $\langle \mu\alpha.c^{\circ} \| E^{\circ}\rangle = c^{\circ}\{E^{\circ}/\alpha\}$ and because the translation is compositional that further equals $c\{E/\alpha\}^{\circ}$.

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash (\mathtt{car}(E) \cdot \mathtt{cdr}(E))^{\circ} = E^{\circ}$

$$\begin{aligned}
&(\mathtt{car}(E) \cdot \mathtt{cdr}(E))^{\circ} && \\
&\triangleq \mathtt{car}(E^{\circ}) \cdot \mathtt{cdr}(E^{\circ}) && \\
&= \tilde{\mu} f.\langle f \| \mathtt{car}(E^{\circ}) \cdot \mathtt{cdr}(E^{\circ})\rangle && \eta_{\tilde{\mu}} \\
&= \tilde{\mu} f.\langle \mu\alpha.\langle f \| \mathtt{car}(\alpha) \cdot \mathtt{cdr}(\alpha)\rangle \| E^{\circ}\rangle && \text{Equation 3.4} \\
&= \tilde{\mu} f.\langle \mu[(x \cdot \beta).\langle \mu\alpha.\langle f \| \mathtt{car}(\alpha) \cdot \mathtt{cdr}(\alpha)\rangle \| x \cdot \beta\rangle] \| E^{\circ}\rangle && \eta \\
&= \tilde{\mu} f.\langle \mu[(x \cdot \beta).\langle f \| \mathtt{car}(x \cdot \beta) \cdot \mathtt{cdr}(x \cdot \beta)\rangle] \| E^{\circ}\rangle && \mu \\
&= \tilde{\mu} f.\langle \mu[(x \cdot \beta).\langle f \| x \cdot \mathtt{cdr}(x \cdot \beta)\rangle] \| E^{\circ}\rangle && \text{Fact 1} \\
&= \tilde{\mu} f.\langle \mu[(x \cdot \beta).\langle f \| x \cdot \beta\rangle] \| E^{\circ}\rangle && \text{Fact 1} \\
&= \tilde{\mu} f.\langle f \| E^{\circ}\rangle && \eta \\
&= E^{\circ} && \eta_{\tilde{\mu}}
\end{aligned}$$

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash (\mu[(x \cdot \alpha).c])^{\circ} = (\mu\beta.c\{\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha\})^{\circ}$

$$\begin{aligned}
\mu[(x \cdot \alpha).c]^{\circ} &\triangleq \mu[(x \cdot \alpha).c^{\circ}] && \\
&= \mu\beta.\langle \mu[(x \cdot \alpha).c^{\circ}] \| \beta\rangle && \eta_{\mu} \\
&= \mu\beta.\langle \mu[(x \cdot \alpha).c^{\circ}] \| \mathtt{car}(\beta) \cdot \mathtt{cdr}(\beta)\rangle && \eta\text{. Shown above} \\
&= \mu\beta.\langle \mathtt{car}(\beta) \| \tilde{\mu} x.\langle \mu\alpha.c^{\circ} \| \mathtt{cdr}(\beta)\rangle\rangle && \beta \\
&= \mu\beta.\langle \mu\alpha.c^{\circ}\{\mathtt{car}(\beta)/x\} \| \mathtt{cdr}(\beta)\rangle && \tilde{\mu} \\
&= \mu\beta.c\{\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha\} && \text{Lemma 11} \\
&\triangleq (\mu\beta.c\{\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha\})^{\circ} &&
\end{aligned}$$

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash \mathtt{car}(v \cdot E)^{\circ} = v^{\circ}$

$$\begin{aligned}
\mathtt{car}(v \cdot E)^{\circ} &\triangleq \mu\alpha.\langle \mu[(x \cdot \_).\langle x \| \alpha\rangle] \| v^{\circ} \cdot E^{\circ}\rangle \\
&= \mu\alpha.\langle v^{\circ} \| \tilde{\mu}x.\langle E^{\circ} \| \mu\_.\langle x\|\alpha\rangle\rangle\rangle && \beta \\
&= \mu\alpha.\langle E^{\circ} \| \mu\_.\langle v^{\circ}\|\alpha\rangle\rangle && \tilde{\mu} \\
&= \mu\alpha.\langle v^{\circ} \| \alpha\rangle && \text{Equation 3.4} \\
&= v^{\circ} && \eta_{\mu}
\end{aligned}$$

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash \mathtt{cdr}(v \cdot E)^{\circ} = E^{\circ}$

$$\begin{aligned}
\mathtt{cdr}(v \cdot E)^{\circ} &\triangleq \tilde{\mu}x.\langle \mu[(\_ \cdot \alpha).\langle x \| \alpha\rangle] \| v^{\circ} \cdot E^{\circ}\rangle \\
&= \tilde{\mu}x.\langle v^{\circ} \| \tilde{\mu}\_.\langle \mu\alpha.\langle x\|\alpha\rangle \| E^{\circ}\rangle\rangle && \beta \\
&= \langle \mu\alpha.\langle x\|\alpha\rangle \| E^{\circ}\rangle && \tilde{\mu} \\
&= \tilde{\mu}x.\langle x \| E^{\circ}\rangle && \text{Equation 3.4} \\
&= E^{\circ} && \eta_{\tilde{\mu}}
\end{aligned}$$

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash \mathtt{car}(e)^{\circ} = (\mu\alpha.\langle \mu\beta.\langle \mathtt{car}(\beta)\|\alpha\rangle \| e\rangle)^{\circ}$

$$\begin{aligned}
\mathtt{car}(e) &\triangleq \mu\alpha.\langle \mu[(x \cdot \_).\langle x \| \alpha\rangle] \| e^{\circ}\rangle \\
&= \mu\alpha.\langle \mu\beta.\langle \mu[(x \cdot \_).\langle x \| \alpha\rangle] \| \beta\rangle \| e^{\circ}\rangle && \eta_{\mu} \\
&= \mu\alpha.\langle \mu\beta.\langle \mu\gamma.\langle \mu[(x \cdot \_).\langle x \| \gamma\rangle] \| \beta\rangle \| \alpha\rangle \| e^{\circ}\rangle && \mu \\
&\triangleq (\mu\alpha.\langle \mu\beta.\langle \mathtt{car}(\beta)\|\alpha\rangle \| e\rangle)^{\circ}
\end{aligned}$$

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash (\texttt{cdr}(e))^{\circ} = (\tilde{\mu}x.\langle\mu\alpha.\langle x\|\texttt{cdr}(\alpha)\rangle\|e\rangle)^{\circ}$

$$\begin{aligned}
(\texttt{cdr}(e))^{\circ} &\triangleq \tilde{\mu}x.\langle\mu[(\_\cdot\beta).\langle x\|\beta\rangle]\|e^{\circ}\rangle \\
&= \tilde{\mu}x.\langle\mu\alpha.\langle\mu[(\_\cdot\beta).\langle x\|\beta\rangle]\|\alpha\rangle\|e^{\circ}\rangle && \eta_{\mu} \\
&= \tilde{\mu}x.\langle\mu\alpha.\langle x\|\tilde{\mu}y.\langle\mu[(\_\cdot\beta).\langle y\|\beta\rangle]\|\alpha\rangle\rangle\|e^{\circ}\rangle && \tilde{\mu} \\
&\triangleq (\tilde{\mu}x.\langle\mu\alpha.\langle x\|\texttt{cdr}(\alpha)\rangle\|e\rangle)^{\circ}
\end{aligned}$$

– $\mu\tilde{\mu}_{\eta}^{\rightarrow} \vdash (v\cdot e)^{\circ} = (\tilde{\mu}x.\langle\mu\alpha.\langle x\|v\cdot\alpha\rangle\|e\rangle)^{\circ}$

$$\begin{aligned}
(v\cdot e)^{\circ} &\triangleq v^{\circ}\cdot e^{\circ} \\
&= \tilde{\mu}x.\langle x\|v^{\circ}\cdot e^{\circ}\rangle && \eta_{\tilde{\mu}} \\
&= \tilde{\mu}x.\langle\mu[(y\cdot\alpha).\langle x\|y\cdot\alpha\rangle\|v^{\circ}\cdot e^{\circ}\rangle && \eta \\
&= \tilde{\mu}x.\langle v^{\circ}\|\tilde{\mu}y.\langle\mu\alpha.\langle x\|y\cdot\alpha\rangle\|e^{\circ}\rangle\rangle && \beta \\
&= \tilde{\mu}x.\langle\mu\alpha.\langle x\|v^{\circ}\cdot\alpha\rangle\|e^{\circ}\rangle && \tilde{\mu} \\
&= (\tilde{\mu}x.\langle\mu\alpha.\langle x\|v\cdot\alpha\rangle\|e\rangle)^{\circ}
\end{aligned}$$

2. If t is in $\mu\tilde{\mu}_{\eta}^{\rightarrow}$ then $t^{\circ} \triangleq t$. That is, the $^{\circ}$ translation is the identity on everything except $\texttt{car}(e)$ and $\texttt{cdr}(e)$ which are constants that do not appear in the $\mu\tilde{\mu}_{\eta}^{\rightarrow}$ sub-syntax.

3. If t is in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ then $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \vdash t^\circ = t$. By induction on t. The only non-trivial cases to handle are $\texttt{car}(e)$ and $\texttt{cdr}(e)$. If e is a co-value, we have:

$$
\begin{aligned}
\texttt{car}(E)^\circ &\triangleq \mu\alpha.\langle \mu[(x \cdot \_).\langle x\|\alpha\rangle]\|E^\circ\rangle \\
&= \mu\alpha.\langle \mu[(x \cdot \_).\langle x\|\alpha\rangle]\|E\rangle && \text{Inductive hypothesis} \\
&= \mu\alpha.\langle \mu\beta.\langle \texttt{car}(\beta)\|\alpha\rangle\|E\rangle && exp \\
&= \mu\alpha.\langle \texttt{car}(E)\|\alpha\rangle && \mu \\
&= \texttt{car}(E) && \eta_\mu \\
\texttt{cdr}(E)^\circ &\triangleq \tilde{\mu}x.\langle \mu[(\_ \cdot \alpha).\langle x\|\alpha\rangle]\|E^\circ\rangle \\
&= \tilde{\mu}x.\langle \mu[(\_ \cdot \alpha).\langle x\|\alpha\rangle]\|E\rangle && \text{Inductive hypothesis} \\
&= \tilde{\mu}x.\langle \mu\beta.\langle x\|\texttt{cdr}(\beta)\rangle\|E\rangle && exp \\
&= \tilde{\mu}x.\langle x\|\texttt{cdr}(E)\rangle && \mu \\
&= \texttt{cdr}(E) && \eta_{\tilde{\mu}}
\end{aligned}
$$

In the case where e is not a co-value:

$$\begin{aligned}
\texttt{car}(e)^\circ &\triangleq \mu\alpha.\langle\mu[(x\cdot \_).\langle x\|\alpha\rangle]\|e^\circ\rangle \\
&= \mu\alpha.\langle\mu[(x\cdot \_).\langle x\|\alpha\rangle]\|e\rangle && \text{Inductive hypothesis} \\
&= \mu\alpha.\langle\mu\beta.\langle\texttt{car}(\beta)\|\alpha\rangle\|e\rangle && exp \\
&= \texttt{car}(e) && \varsigma_{\text{car}} \\
\texttt{cdr}(e)^\circ &\triangleq \tilde{\mu}x.\langle\mu[(\_\cdot\alpha).\langle x\|\alpha\rangle]\|e^\circ\rangle \\
&= \tilde{\mu}x.\langle\mu[(\_\cdot\alpha).\langle x\|\alpha\rangle]\|e\rangle && \text{Inductive hypothesis} \\
&= \tilde{\mu}x.\langle\mu\beta.\langle x\|\texttt{cdr}(\beta)\rangle\|e\rangle && exp \\
&= \texttt{cdr}(e) && \varsigma_{\text{cdr}}
\end{aligned}$$

□

Confluence for Extensional Call-By-Name Reduction

Now we return to our original question of confluence as it applies to the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ reduction theory. The main obstacle we must face is the fact that the surjectivity rule for call stacks, $\eta.$, is not left-linear since it requires two projections out of the *same* co-value. This is a problem because we might reduce one of the sub co-values in an $\eta.$-redex, as in:

$$\begin{tikzcd}
\texttt{car}(E)\cdot\texttt{cdr}(E) \arrow[r, "\eta."] \arrow[d] & E \\
\texttt{car}(E')\cdot\texttt{cdr}(E) \arrow[r, "\eta.", "/"{marking}] & {}
\end{tikzcd}$$

The two copies of E have gotten out of synch, so inner reductions can destroy surrounding $\eta.$ redexes. As a consequence, many standard rewriting techniques

$$\begin{aligned}
c \in \text{Commands} &::= \langle v \| e \rangle \\
v \in \text{Terms} &::= \mu\alpha.c \mid x \mid \mathtt{car}(E) \mid \mu[(x \cdot \alpha).c] \\
E \in \text{Co-Values} &::= \alpha \mid \mathtt{cdr}(E) \mid v \cdot E \\
e \in \text{Co-Terms} &::= \tilde{\mu}x.c \mid E
\end{aligned}$$

FIGURE 16. $\mu\tilde{\mu}^{F}_{\text{cons}}$: the focalized sub-syntax of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$

for establishing confluence—such as parallel reduction, full development, and commutation or reordering of reductions—do not directly apply to the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ calculus.

Instead, we look to simplify the calculus, eliminating any extraneous features and keeping only the essential kernel that is necessary for performing computation, until the non-left-linearity of $\eta.$ is no longer problematic. Then, we hoist confluence of the kernel into confluence of the full $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$-calculus. As it turns out, a minor restriction of the kernel calculus has appeared before as the stack calculus in Carraro et al. (2012), which is already known to be confluent. In repeating the proof of confluence for the extended stack calculus, we take the opportunity to emphasize what we believe to be the single key idea for confluence of both $\Lambda\mu_{\mathtt{cons}}$ and the stack calculus (Carraro et al. (2012); Nakazawa and Nagai (2014)).

In order to relate the original reduction theory of $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ with the simpler one of the stack calculus, we will use the directed analog of equational correspondence for reductions known as a Galois connection or an adjunction (Sabry and Wadler (1997)). The conditions of a Galois connection are essentially the same as the four conditions of an equational correspondence, except that we need to be careful about the direction of arrows. More specifically, given a source calculus S and target T, the translations $\flat : S \to T$ and $\sharp : T \to S$ form a *Galois connection* from S to T if and only if the following four conditions hold:

1. ($\flat$) For all terms s_1, s_2 of S, $s_1 \twoheadrightarrow S s_2$ implies $s_1^\flat \twoheadrightarrow T s_2^\flat$.

2. ($\sharp$) For all terms t_1, t_2 of T, $t_1 \twoheadrightarrow T t_2$ implies $t_1^\sharp \twoheadrightarrow S t_1^\sharp$.

3. ($\flat\sharp$) For all terms s of S, $s \twoheadrightarrow S (s^\flat)^\sharp$.

4. ($\sharp\flat$) For all terms t of T, $(t^\sharp)^\flat \twoheadrightarrow T t$.

Additionally, if the fourth condition is strengthened so that any term t of T is syntactically equal to $(t^\sharp)^\flat$, then $\flat$ and $\sharp$ form a *reflection* in S of T. Galois connections are convenient to work with because they compose: given Galois connections from S_1 to S_2 and from S_2 to S_3, we have one from S_1 to S_3. This lets us break a complex translation down into simpler steps and put them back together in the end. More crucially for our purposes, a Galois connection allows us to hoist confluence of the target reduction theory into the source.

Theorem 3. *Given a Galois connection from S to T, S is confluent whenever T is. Furthermore, given a reflection in S of T, S is confluent if and only if T is.*

Proof. Let $\flat : S \to T$ and $\sharp : T \to S$ be the translations of the Galois connection. Supposing that $s_1 \twoheadleftarrow_S s \twoheadrightarrow S s_2$, the following diagram commutes by confluence of T:

$$\begin{tikzcd}
 & & s \arrow[ddll, two heads, "S"'] \arrow[d, "\flat"] \arrow[ddrr, two heads, "S"] & & \\
 & (\flat) & s^\flat \arrow[dl, two heads, "T"'] \arrow[dr, two heads, "T"] & (\flat) & \\
s_1 \arrow[r, "\flat"] \arrow[dr, two heads, "S"'] & s_1^\flat \arrow[d, "\sharp"] \arrow[dr, two heads, "T"] & \mathit{confl.} & s_2^\flat \arrow[dl, two heads, "T"'] \arrow[d, "\sharp"'] & s_2 \arrow[l, "\flat"'] \arrow[dl, two heads, "S"] \\
(\flat\sharp) & s_1^{\flat\sharp} \arrow[dr, two heads, "S"'] \arrow[r, phantom, "(\sharp)"] & t \arrow[d, "\sharp"] \arrow[r, phantom, "(\sharp)"] & s_2^{\flat\sharp} \arrow[dl, two heads, "S"] & (\flat\sharp) \\
 & & t^\sharp & &
\end{tikzcd}$$

Additionally, a reflection gives us the fact that for any term t of T, $t \equiv (t^\sharp)^\flat$ (where $\equiv$ is syntactic equality), meaning that $t \twoheadrightarrow T(t^\sharp)^\flat$ by reflexivity. Thus, with a reflection we can swap S with T and $\flat$ with $\sharp$ in the above diagram so that it commutes again by confluence of S. □

Our proof of confluence for the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$-calculus is broken down into two separate parts:

1. We establish a reflection in the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$-calculus of an extension of the stack calculus (called the Σ^x-calculus in Figure 18). This reflection is formed as a result of normalization in two steps. First, we show that ς-normalization forms a reflection in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ of its focalized sub-syntax (called $\mu\tilde{\mu}^{F}_{\text{cons}}$ in Figure 16). Second, we show that $\tilde{\mu}\texttt{exp}$-normalization forms a reflection in $\mu\tilde{\mu}^{F}_{\text{cons}}$ of Σ^x. The full reflection comes out from composition of these two steps.

2. We elaborate the confluence proof of the Σ^x-calculus. First, we show that the Σ^x reduction theory is equivalent to one with a restricted η. surjectivity rule.

The restricted rule only applies to co-values with no other possible reductions, so it avoids the problem where a η. redex is destroyed by getting out of sync. Second, we finish the proof by showing that the reduction theory with this restricted η. rule is confluent, meaning that the original Σ^x-calculus is also confluent.

To conclude, since $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ contains a reflection of the confluent Σ^x-calculus, from the above theorem $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ must be confluent as well.

A stack calculus and its reflection

We demonstrate a reflection in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ of Σ^x by a sequence of normal forms with respect to two well-behaved subsets of the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ reduction theory. On their own, these sets of reductions are normalizing and confluent, so their normal forms can be computed all at once ahead of time to eliminate particular features of the source $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$-calculus. Furthermore, their normal forms are closed under reduction, so that further reduction does not re-introduce those eliminated features, giving a smaller target calculus. As such, these reductions can be seen as being "administrative" in nature, since they simplify some feature down to more primitive components, resulting in a simpler target calculus.

Reflection by normalization is a rather specific form of a general Galois connection, so we have some extra knowledge about how the source and target calculi are related to one another. In particular, we begin with the reduction theory for the source calculus (S) and divide it into two parts: a set of administrative reductions done upfront during normalization (A), and the remaining non-administrative reductions that are carried over into the target (T). The target calculus then becomes T-reductions over A-normal forms. So long as the A

reduction theory is confluent, a reflection in S of T via A-normalization just tells us that full A-normalization commutes across T-reduction. In the following diagrams, a dashed arrow stands for the existence of such a reduction.

Theorem 4. *Let S, T, and A be reduction theories such that A is confluent and normalizing and S is equivalent to the (reflexive, transitive) union of T and A. A-normalization forms a reflection in S of T if and only if A-normalization commutes across T reduction, i.e. for all $s_1 \twoheadrightarrow Ts_2$ and their A-normal forms t_1 and t_2, respectively:*

$$\begin{tikzcd}
s_1 \arrow[r, two heads, "T"] \arrow[d, two heads, "A"] & s_2 \arrow[d, two heads, "A"] \\
t_1 \arrow[r, two heads, dashed, "T"] & t_2
\end{tikzcd}$$

Proof. First, we note that every term s of S has a unique A-normal form (there is at least one because A is normalizing and at most one because A is confluent) which we denote s^A, so our translation functions are A-normalization ($s^\flat = s^A$) and inclusion of A-normal forms inside the original language S ($t^\sharp = t$). To show the right-to-left implication, given the above commutation, A-normalization and inclusion form a reflection in S of T:

1. ($\flat$): Suppose that $s_1 \twoheadrightarrow_S s_2$. Because S reduction is equivalent to the reflexive, transitive closure over both A and T reductions, we equivalently have that

   $$s_1 \twoheadrightarrow_A \twoheadrightarrow_T \twoheadrightarrow_A \twoheadrightarrow_T \dots \twoheadrightarrow_A \twoheadrightarrow_T s_2$$

   We can therefore show that $s_1^A \twoheadrightarrow_T s_2^A$ by induction over the reduction over n in $s_1(\twoheadrightarrow_A \twoheadrightarrow_T)^n s_2$. If $n = 0$ then the result is immediate by reflexivity of T. Otherwise, if $n = 1 + m$ we have $s_1 \twoheadrightarrow_A s_1' \twoheadrightarrow_T s_2'(\twoheadrightarrow_A \twoheadrightarrow_T)^m s_2$, and the result follows from confluence of A, commutation of T over A-normalization, and the inductive hypothesis:

$$\begin{tikzcd}
s_1 \arrow[r, two heads, "A"] \arrow[d, two heads, "A"'] \arrow[dr, phantom, "A\ confl."] & s_1' \arrow[r, two heads, "T"] \arrow[d, two heads] \arrow[dr, phantom, "A\ comm."] & s_2' \arrow[r, two heads, "(\twoheadrightarrow_A \twoheadrightarrow_T)"] \arrow[d, two heads] \arrow[dr, phantom, "A\ \ IH"] & s_2 \arrow[d, two heads, "A"] \\
s_1^A \arrow[r, equal] & s_1'^A \arrow[r, two heads, dashed, "T"] & s_2'^A \arrow[r, two heads, dashed, "T"] & s_2^A
\end{tikzcd}$$

2. ($\sharp$): For all A-normal forms t_1 and t_2, $t_1 \twoheadrightarrow T t_2$ implies $t_1 \twoheadrightarrow S t_2$ because T reduction is included in S.

3. ($\flat\sharp$): For all s, $s \twoheadrightarrow S s^A$ because A reduction is included in S.

4. ($\sharp\flat$): For all A-normal forms t, $t^A \equiv t$.

To show the left-to-right implication, given that A-normalization forms a reflection in S of T, the commutation of A-normalization across T reduction is a special case of the first property ($\flat$), since T reduction is included in S. Specifically, if $s_1 \twoheadrightarrow_T s_2$ then since T is included in S, $s_1 \twoheadrightarrow_S s_2$ and so by the property $\flat$, $t_1 \twoheadrightarrow_T t_2$ where t_1 and t_2 are the A-normal forms of s_1 and s_2 respectively. $\square$

We now build our reflection in $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ of Σ^x by $\varsigma\tilde{\mu}\texttt{exp}$-normalization in two steps. First, we fully normalize commands, terms and co-terms of the $\mu\tilde{\mu}^{\rightarrow}_{\text{cons}}$ by the ς-rules. Second, we normalize ς-normal forms by the $\tilde{\mu}$ and $\texttt{exp}$ rules. We separate these two steps due to the unnecessary complication of performing full $\varsigma\tilde{\mu}\texttt{exp}$-normalization at once: the normal forms are difficult to identify, and even showing that reduction is normalizing is not obvious. The complication is due to the fact that ς-reduction creates *new* $\tilde{\mu}$-abstractions, and thus new $\tilde{\mu}$-redexes. However, when taken separately, ς-normalization produces all the necessary extra $\tilde{\mu}$-abstractions first, so that $\tilde{\mu}$-normalization can easily eliminate them all afterward.

And since reflections compose, these two steps can be performed in sequence to build an overall reflection in $\mu\tilde{\mu}_{\text{cons}}^{\rightarrow}$ of $\varsigma\tilde{\mu}\mathtt{exp}$-normal forms.

The ς-normal forms give the focalized sub-syntax of the $\mu\tilde{\mu}_{\text{cons}}^{\rightarrow}$-calculus, called $\mu\tilde{\mu}_{\text{cons}}^{F}$ in Figure 16, where call stacks are built out of co-values, and stack projections only apply to co-values. In effect, the focalized sub-syntax limits general co-terms, so that a $\tilde{\mu}$-abstraction can only appear at the top of a co-term, and not arbitrarily nested inside call stacks. Also notice that the sub-syntax of $\mu\tilde{\mu}_{\text{cons}}^{F}$ is closed under reduction: once we have fully applied all ς-rules, they never come up again during reduction. Thus, the reduction theory of the $\mu\tilde{\mu}_{\text{cons}}^{F}$-calculus consists of all non-ς rules. This gives us a reflection in $\mu\tilde{\mu}_{\text{cons}}^{\rightarrow}$ of $\mu\tilde{\mu}_{\text{cons}}^{F}$ by ς-normalization.

In order to have an idea of how this proof works, first, note that ς-reduction is confluent (it is an orthogonal combinatory reduction system, see Klop, van Oostrom, and van Raamsdonk (1993)) and normalizing (each application of ς-reduction decreases the number of non-co-values, e, sitting in a call stack, $v \cdot e$, or projection, $\mathtt{car}(e)$ or $\mathtt{cdr}(e)$). Therefore, by Theorem 4, we only need to show that all other reductions of $\mu\tilde{\mu}_{\text{cons}}^{\rightarrow}$, namely those of $\mu\tilde{\mu}_{\text{cons}}^{F}$, commute over ς-normalization, where we denote the ς-normal form of c as c^{ς} (similarly v^{ς}, e^{ς}, and E^{ς}). In order to establish commutation, we consider commutation of a single $\mu\tilde{\mu}_{\text{cons}}^{F}$ step over ς-normalization:

$$\begin{tikzcd}
c_1 \arrow[r, "\mu\tilde{\mu}_{\text{cons}}^{F}"] \arrow[d, twoheadrightarrow, "\varsigma"] & c_2 \arrow[d, twoheadrightarrow, "\varsigma"] \\
c_1^{\varsigma} \arrow[r, dashed, twoheadrightarrow, "\mu\tilde{\mu}_{\text{cons}}^{F}"] & c_2^{\varsigma}
\end{tikzcd}$$

from which the full commutation result is obtained by composition over multiple $\mu\tilde{\mu}_{\text{cons}}^{F}$ steps. The single-step commutation can be shown by mutual induction on commands, terms, and co-terms, considering the possible reductions in each case. Most of these follow directly by the inductive hypothesis, using the additional facts that co-values are closed under reduction and ς-normalization commutes with substitution.

The interesting cases are those in which internal reductions are capable of destroying surrounding redexes. In particular, a $\eta.$ redex can be ruined by putting the co-values out of synch due to asymmetric reduction. Fortunately, because we are forced to fully reduce to the unique ς-normal form, and because co-values are closed under reduction, $\eta.$ reduction commutes:

$$\begin{tikzcd}
\mathtt{car}(E) \cdot \mathtt{cdr}(E) \arrow[r, "\eta."] \arrow[d, two heads, "\varsigma"] & E \arrow[d, two heads, "\varsigma"] \\
\mathtt{car}(E^{\varsigma}) \cdot \mathtt{cdr}(E^{\varsigma}) \arrow[r, dashed, "\eta."] & E^{\varsigma}
\end{tikzcd}$$

The other case in which an internal reduction may destroy an outer redex is in the case of ς rules themselves. This is because a non-co-value co-term may be converted into a co-value by an $\eta_{\tilde{\mu}}$-reduction, which prevents the ς-family of rules from applying and likewise changes the final shape of the ς-normal form. However, substitution by $\tilde{\mu}$ is capable of undoing such an unnecessary ς-reduction, and additional η_{μ} and $\eta_{\tilde{\mu}}$ reductions clean up the leftover μ- and $\tilde{\mu}$-abstractions:

$$\begin{tikzcd}
v \cdot e \arrow[r, "\eta_{\tilde{\mu}}"] \arrow[d, two heads, "\varsigma"] & v \cdot E \arrow[d, two heads, "\varsigma"] \\
\tilde{\mu}x.\langle \mu\alpha.\langle x \| v^{\varsigma} \cdot \alpha \rangle \| e^{\varsigma} \rangle \arrow[r, dashed, two heads, "\mu\tilde{\mu}^{F}_{\mathrm{cons}}"] & v^{\varsigma} \cdot E^{\varsigma}
\end{tikzcd}$$

Similar diagrams hold for the other ς rules for $\mathtt{car}(e)$ and $\mathtt{cdr}(e)$ when e is a non-co-value that reduces to a co-value.

The full details follow.

Lemma 12 (Closure of $\mu\tilde{\mu}^F_{\text{cons}}$). *1. Closure under substitution: If t, v, E are in the focalized sub-syntax, then $t\{v/x, E/\alpha\}$ is in the same syntactic category as t in $\mu\tilde{\mu}^F_{cons}$.*

2. Closure under reduction: If t is in the focalized sub-syntax and $t \to t'$ according to the rules of $\mu\tilde{\mu}^{\to}_{cons}$, then t' is in the focalized sub-syntax (and in the same syntactic category as t).

Proof. The first point follows by induction on t. For the second point, we proceed by cases. The μ, $\tilde{\mu}$, and exp rules hold by the closure under substitution. The η, car, and cdr rules are all immediate. Finally, the various ς-rules are simply syntactically prohibited as their left hand sides are not focalized because of the requirement that the lifted e is not a co-value. □

While as discussed previously, ς reduction is normalizing and confluent, it is easier to work with a single procedure which computes a terms ς-normal form all in one step. Such a procedure is given in Figure 17.

Lemma 13. $(-)^\varsigma$ *computes the unique ς-normal form of any command, term, or co-term in $\mu\tilde{\mu}^{\to}_{cons}$*

Proof. Observe that E^ς is always a co-value. Therefore, by cases, we know that t^ς is in $\mu\tilde{\mu}^F_{\text{cons}}$ and that it is a ς-normal form.

By induction we see that $t \twoheadrightarrow_\varsigma t^\varsigma$. x^ς and α^ς work in zero steps, $(\mu\alpha.c)^\varsigma$, $\mu[(x \cdot \alpha).c]^\varsigma$, $(\tilde{\mu}x.c)^\varsigma$, $\texttt{car}(E)^\varsigma$, $\texttt{cdr}(E)^\varsigma$, $(v \cdot E)^\varsigma$, and $\langle v \| e\rangle^\varsigma$ work by the inductive hypothesis, the remaining cases of $\texttt{car}(e)^\varsigma$, $\texttt{cdr}(e)^\varsigma$, $(v \cdot e)^\varsigma$ each correspond to a single application of a ς rule followed by additional ς reductions given by the inductive hypothesis.

$$\begin{aligned}
x^\varsigma &\triangleq x \\
\alpha^\varsigma &\triangleq \alpha \\
(\mu\alpha.c)^\varsigma &\triangleq \mu\alpha.(c^\varsigma) \\
\mu[(x\cdot\alpha).c]^\varsigma &\triangleq \mu[(x\cdot\alpha).c^\varsigma] \\
(\tilde{\mu}x.c)^\varsigma &\triangleq \tilde{\mu}x.(c^\varsigma) \\
\texttt{car}(E)^\varsigma &\triangleq \texttt{car}(E^\varsigma) \\
\texttt{car}(e)^\varsigma &\triangleq \mu\alpha.\langle\mu\beta.\langle\texttt{car}(\beta)\|\alpha\rangle\|e^\varsigma\rangle & (e \notin \text{Co-Values}) \\
\texttt{cdr}(E)^\varsigma &\triangleq \texttt{cdr}(E^\varsigma) \\
\texttt{cdr}(e)^\varsigma &\triangleq \tilde{\mu}x.\langle\mu\alpha.\langle x\|\texttt{cdr}(\alpha)\rangle\|e^\varsigma\rangle & (e \notin \text{Co-Values}) \\
(v\cdot E)^\varsigma &\triangleq v^\varsigma\cdot E^\varsigma \\
(v\cdot e)^\varsigma &\triangleq \tilde{\mu}x.\langle\mu\alpha.\langle x\|v^\varsigma\cdot\alpha\rangle\|e^\varsigma\rangle & (e \notin \text{Co-Values}) \\
\langle v\|e\rangle^\varsigma &\triangleq \langle v^\varsigma\|e^\varsigma\rangle
\end{aligned}$$

FIGURE 17. The $(-)^\varsigma : \mu\tilde{\mu}^{\rightarrow}_{\text{cons}} \to \mu\tilde{\mu}^{F}_{\text{cons}}$ translation focalizing programs

Since ς is a (linear) term rewriting system without any critical pairs it is confluent and so must have unique normal forms. Thus, t^ς is the unique ς-normal form of t. □

Lemma 14. *For any t,v,E,x, and α, $t^\varsigma\{v^\varsigma/x\} \triangleq (t\{v/x\})^\varsigma$ and $t^\varsigma\{E^\varsigma/\alpha\} \triangleq (t\{E/\alpha\})^\varsigma$.*

Proof. Induction on t. The only trick is that substitution will never change what is a co-value. □

Lemma 15. *ς-normalization commutes across the non-ς reductions of $\mu\tilde{\mu}^{\rightarrow}_{cons}$.*

Proof. By Lemma 13 it is enough to show that if $t \twoheadrightarrow t'$ then $t^\varsigma \twoheadrightarrow (t')^\varsigma$. By inducting over the reduction $t \twoheadrightarrow t'$ it is sufficient to show that $t \to t'$ implies $t^\varsigma \twoheadrightarrow (t')^\varsigma$. Examining the cases, the main issue comes when we have an internal reduction into a (co-)term subject to lifting as reduction might turn a non-co-value co-term into a co-value and thus change the translation. The main idea then is to prove the special case that if $e \to E$ then $e^\varsigma \twoheadrightarrow E^\varsigma$. This follows by induction on the derivation of the reduction $e \to E$, with only a small number of cases to consider:

– If $\tilde{\mu}x.\langle x \| E \rangle \to E$

$$\begin{aligned}
(\tilde{\mu}x.\langle x \| E \rangle)^\varsigma &\triangleq \tilde{\mu}x.\langle x \| E^\varsigma \rangle \\
&\to E^\varsigma && \eta_{\tilde{\mu}}
\end{aligned}$$

– If $v \cdot e \to v \cdot E$

$$\begin{aligned}
(v \cdot e)^\varsigma &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| v^\varsigma \cdot \alpha \rangle \| e^\varsigma \rangle \\
&\twoheadrightarrow \tilde{\mu}x.\langle \mu\alpha.\langle x \| v^\varsigma \cdot \alpha \rangle \| E^\varsigma \rangle && \text{Inductive Hypothesis} \\
&\to \tilde{\mu}x.\langle x \| v^\varsigma \cdot E^\varsigma \rangle && \mu \\
&\to v^\varsigma \cdot E^\varsigma && \eta_{\tilde{\mu}} \\
&\triangleq (v \cdot E)^\varsigma
\end{aligned}$$

– If $\mathtt{cdr}(e) \to \mathtt{cdr}(E)$

$$\begin{aligned}
\mathtt{cdr}(e)^\varsigma &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| \mathtt{cdr}(\alpha) \rangle \| e^\varsigma \rangle \\
&\twoheadrightarrow \tilde{\mu}x.\langle \mu\alpha.\langle x \| \mathtt{cdr}(\alpha) \rangle \| E^\varsigma \rangle && \text{Inductive Hypothesis} \\
&\to \tilde{\mu}x.\langle x \| \mathtt{cdr}(E^\varsigma) \rangle && \mu \\
&\to \mathtt{cdr}(E^\varsigma) && \eta_{\tilde{\mu}} \\
&\triangleq \mathtt{cdr}(E)^\varsigma
\end{aligned}$$

– If $\mathtt{car}(e) \to \mathtt{car}(E)$

$$
\begin{aligned}
\mathtt{car}(e)^{\varsigma} &\triangleq \mu\alpha.\langle \mu\beta.\langle \mathtt{car}(\beta) \| \alpha\rangle \| e^{\varsigma}\rangle \\
&\twoheadrightarrow \mu\alpha.\langle \mu\beta.\langle \mathtt{car}(\beta) \| \alpha\rangle \| E^{\varsigma}\rangle && \text{Inductive Hypothesis} \\
&\to \mu\alpha.\langle \mathtt{car}(E^{\varsigma}) \| \alpha\rangle && \mu \\
&\to \mathtt{car}(E^{\varsigma}) && \eta_\mu \\
&\triangleq \mathtt{car}(E)^{\varsigma}
\end{aligned}
$$

Otherwise, the translation is completely compositional and so the cases are direct:

– If $\langle \mu\alpha.c \| E\rangle \to_\mu c\{E/\alpha\}$ we have:

$$
\begin{aligned}
\langle \mu\alpha.c \| E\rangle^{\varsigma} &\triangleq \langle \mu\alpha.c^{\varsigma} \| E^{\varsigma}\rangle \\
&\to_\mu c^{\varsigma}\{E^{\varsigma}/\alpha\} \\
&=_\alpha c\{E/\alpha\}^{\varsigma} && \text{Lemma 14.}
\end{aligned}
$$

– If $\langle v \| \tilde\mu x.c\rangle \to_{\tilde\mu} c[v/x]$ we have:

$$
\begin{aligned}
\langle v \| \tilde\mu x.c\rangle^{\varsigma} &\triangleq \langle v^{\varsigma} \| \tilde\mu x.c^{\varsigma}\rangle \\
&\to_\mu c^{\varsigma}\{v^{\varsigma}/x\} \\
&=_\alpha c\{v/x\}^{\varsigma} && \text{Lemma 14.}
\end{aligned}
$$

– If $\mu\alpha.\langle v \| \alpha\rangle \to_{\eta_\mu} v$ we have: $(\mu\alpha.\langle v \| \alpha\rangle)^{\varsigma} \triangleq \mu\alpha.\langle v^{\varsigma} \| \alpha\rangle \to_{\eta_\mu} v^{\varsigma}$.

– If $\tilde\mu x.\langle x \| e\rangle \to_{\eta_{\tilde\mu}} e$ we have: $(\tilde\mu x.\langle x \| e\rangle)^{\varsigma} \triangleq \tilde\mu x.\langle x \| e^{\varsigma}\rangle \to_{\eta_{\tilde\mu}} e^{\varsigma}$.

$$\begin{aligned}
c \in \text{Commands} &::= \langle v \| E \rangle \\
v \in \text{Terms} &::= \mu\alpha.c \mid x \mid \mathtt{car}(E) \\
E \in \text{Co-Values} &::= \alpha \mid \mathtt{cdr}(E) \mid v \cdot E \\
e \in \text{Co-Terms} &::= \tilde{\mu}x.c \mid E
\end{aligned}$$

$$\begin{aligned}
\langle \mu\alpha.c \| E \rangle &\to_{\mu} c\{E/\alpha\} \\
\mu\alpha.\langle v \| \alpha \rangle &\to_{\eta_\mu} v \\
\tilde{\mu}x.\langle x \| e \rangle &\to_{\eta_{\tilde{\mu}}} e \\
\mathtt{car}(E) \cdot \mathtt{cdr}(E) &\to_{\eta\cdot} E \\
\mathtt{car}(v \cdot E) &\to_{\mathtt{car}} v \\
\mathtt{cdr}(v \cdot E) &\to_{\mathtt{cdr}} E
\end{aligned}$$

FIGURE 18. Stack calculus with free variables - Σ^x

– If $\mu[(x \cdot \alpha).c] \to_{\text{exp}} \mu\beta.c\{\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha\}$ we have:

$$\begin{aligned}
\mu[(x \cdot \alpha).c]^{\varsigma} &\triangleq \mu[(x \cdot \alpha).c^{\varsigma}] \\
&\to_{\text{exp}} \mu\beta.c^{\varsigma}\{\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha\} \\
&\triangleq \mu\beta.c\{\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha\}^{\varsigma} && \text{Lemma 14.}
\end{aligned}$$

– If $\mathtt{car}(v \cdot E) \to_{\text{car}} v$ we have: $\mathtt{car}(v^{\varsigma} \cdot E^{\varsigma}) \to_{\text{car}} v^{\varsigma}$.

– If $\mathtt{cdr}(v \cdot E) \to_{\text{cdr}} E$ we have: $\mathtt{cdr}(v^{\varsigma} \cdot E^{\varsigma}) \to_{\text{cdr}} E^{\varsigma}$.

– In the case of $\mathtt{car}(E) \cdot \mathtt{cdr}(E) \to_{\eta\cdot} E$ we have $(\mathtt{car}(E) \cdot \mathtt{cdr}(E))^{\varsigma} \triangleq \mathtt{car}(E^{\varsigma}) \cdot \mathtt{cdr}(E^{\varsigma}) \to_{\eta\cdot} E^{\varsigma}$, since E^{ς} is still a co-value.

□

Lemma 16. *ς-normalization forms a reflection in $\mu\tilde{\mu}^{\to}_{cons}$ of $\mu\tilde{\mu}^{F}_{cons}$.*

Proof. Follows as the composition of Lemma 15 and Theorem 4. □

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta}[id_R] \qquad \frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}[id_L]$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \vdash \Delta}[Act_R] \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}[Act_L]$$

$$\frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \mid E : A \vdash \Delta}{\langle v \| E \rangle : (\Gamma \vdash \Delta)}[cut]$$

$$\frac{\Gamma \vdash v : A \mid \Delta \qquad \Gamma \mid E : B \vdash \Delta}{\Gamma \mid v \cdot E : A \to B \vdash \Delta}[\to Intro]$$

$$\frac{\Gamma \mid E : A \to B \vdash \Delta}{\Gamma \vdash \mathtt{car}(E) : A \mid \Delta}[\to Elim_1] \qquad \frac{\Gamma \mid E : A \to B \vdash \Delta}{\Gamma \mid \mathtt{cdr}(E) : B \vdash \Delta}[\to Elim_2]$$

FIGURE 19. Simple Type Assignment for Σ^x

The second step of our reflection is to normalize the focalized $\mu\tilde{\mu}^F_{\text{cons}}$ sub-syntax by $\tilde{\mu}\mathtt{exp}$-reduction. These normal forms are exactly the stack calculus of Carraro et al. (2012) extended with free variables, Σ^x, shown in Figure 18. From the typed perspective, as in Figure 19, has only elimination rules and left introduction rules. Together with natural deduction (only right rules) and the pure sequent calculus we started with (only introduction rules on both sides) it thus represents one of a multitude of choices for a logical inference system. But, as Carraro et al. (2012) noticed, this choice is interesting computationally. In effect, $\tilde{\mu}\mathtt{exp}$-normalization eliminates all variable binders within commands, terms, and co-values, replacing them either by substitution or by a projection out of a co-variable. Therefore, commands, terms, and co-values do not contain *any* variable binders. The one technical detail is the presence of general co-terms remaining in the syntax. This is necessary to form a reflection in $\mu\tilde{\mu}^F_{\text{cons}}$ because of the fact that

$$
\begin{aligned}
\langle v \| \tilde{\mu} x.c \rangle^{\tilde{\mu}} &\triangleq c^{\tilde{\mu}}[v^{\tilde{\mu}}/x] \\
\langle v \| E \rangle^{\tilde{\mu}} &\triangleq \langle v^{\tilde{\mu}} \| E^{\tilde{\mu}} \rangle \\
x^{\tilde{\mu}} &\triangleq x \\
(\mu\alpha.c)^{\tilde{\mu}} &\triangleq \mu\alpha.(c^{\tilde{\mu}}) \\
\mu[(x \cdot \alpha).c]^{\tilde{\mu}} &\triangleq \mu\beta.c^{\tilde{\mu}}[\mathtt{car}(\beta)/x, \mathtt{cdr}(\beta)/\alpha] \\
\mathtt{car}(E)^{\tilde{\mu}} &\triangleq \mathtt{car}(E^{\tilde{\mu}}) \\
\alpha^{\tilde{\mu}} &\triangleq \alpha \\
\mathtt{cdr}(E)^{\tilde{\mu}} &\triangleq \mathtt{cdr}(E^{\tilde{\mu}}) \\
(v \cdot E)^{\tilde{\mu}} &\triangleq v^{\tilde{\mu}} \cdot E^{\tilde{\mu}}
\end{aligned}
$$

FIGURE 20. Translation from $\mu\tilde{\mu}^{F}_{\text{cons}}$ to Σ^x

a given co-term $\tilde{\mu}x.c$ will still reduce to another $\tilde{\mu}$-abstraction $\tilde{\mu}x.c'$. However, the only $\tilde{\mu}$-abstraction in the resulting co-term will be the one at the top; c' contains no other $\tilde{\mu}$-abstractions. Thus, besides the possibility of one $\tilde{\mu}$-abstraction at the very top of a co-term, the Σ^x-calculus has no variable binders. And if general co-terms are not of interest, this detail may be elided. Furthermore, like the focalized $\mu\tilde{\mu}^{F}_{\text{cons}}$ sub-syntax, the syntax of the Σ^x-calculus is also closed under reduction, so the reduction theory of the Σ^x-calculus only consists of the car, cdr, $\eta.$, μ, η_μ, and $\eta_{\tilde{\mu}}$ (at the top of a co-term only) rules. This gives us a reflection in $\mu\tilde{\mu}^{F}_{\text{cons}}$ of Σ^x by $\tilde{\mu}$exp-normalization.

The function $(-)^{\tilde{\mu}}$ translates commands and terms in the focalized sub-syntax ($\mu\tilde{\mu}^{F}_{\text{cons}}$, Figure 16) into Σ^x and is given in Figure 20.

Lemma 17 (Closure of Σ^x). *1. – Closure under substitution: If t, v, E are in Σ^x then $t[v/x, E/\alpha]$ is in the same syntactic category as t in Σ^x.*

2. – Closure under reduction: If t is in Σ^x and $t \to t'$ according to the rules of $\mu\tilde{\mu}^{\to}_{cons}$ then t' is in Σ^x (in the same syntactic category as t).

Proof. The first point follows by induction on t. The second point by cases. □

Lemma 18. *For every t, v, E in the focalized sub-syntax, $t\{v/x, E/\alpha\}^{\tilde{\mu}} \triangleq t^{\tilde{\mu}}\{v^{\tilde{\mu}}/x, E^{\tilde{\mu}}/\alpha\}$.*

Proof. By induction on t. □

Lemma 19. *$-^{\tilde{\mu}}$ computes unique normal forms of $\mu\tilde{\mu}^F_{cons}$ with respect to the $\tilde{\mu}$- and* exp-*rules.*

Proof. For every t in $\mu\tilde{\mu}^F_{\text{cons}}$, $t \twoheadrightarrow t^{\tilde{\mu}}$. This holds by induction on t. In each case we need only to perform at most one $\tilde{\mu}$- or `exp`-reduction and then utilize the inductive hypothesis. $t^{\tilde{\mu}}$ does not contains any $\tilde{\mu}$- or `exp`-redexes and is thus a normal form. $\tilde{\mu}$`exp`-reduction is confluent because it lacks critical pairs. □

Intermezzo 1. Note that the above Lemma implies that $\tilde{\mu}$`exp`-reduction is normalizing as the $(-)^{\tilde{\mu}}$ function computes a normal form with respect to it. However, it is interesting to think about why $\tilde{\mu}$-reduction is normalizing (the `exp`-rule is clearly strongly normalizing on its own as the number of $\mu[(x \cdot \alpha).c]$ forms is strictly decreasing).

As shown later in Theorem 16, our normalization proof for the typed system developed Chapter VI could be reworked to prove that μ and $\tilde{\mu}$ is strongly normalizing even in the untyped system, by constructing a "type system" with a single type. One downside of that proof though is that it seems to have quite high proof complexity, both in that we build quite a bit of machinery to make it work, and in a technical sense : the existence of a fixed-point is guaranteed by the Tarski fixed point theorem which depends on (foundationally) powerful tools such as uncountable unions or higher ordinals. That isn't much concern when our goal is a proof of strong normalization which already implies the consistency of arithmetic, but does seem a bit much if all we care about is some basic property of rewriting systems. However, we can give a simpler, and arithmetic, proof in the case of just $\tilde{\mu}$ alone which may provide some insight.

The basic idea for how we could go about showing $\tilde{\mu}$-normalization is that $\tilde{\mu}$-reduction never introduces any new redexes. In particular if v is $\tilde{\mu}$-normal then $c\{v/x\}$ has one fewer $\tilde{\mu}$-redex then $\langle v \| \tilde{\mu}x.c\rangle$. Thus, there is clearly a reduction order which is normalizing. Further, strong normalization could be additionally shown by interpreting each (co-)term or command as a polynomial with positive integer coefficients.

$$\begin{aligned}
[\![\langle v \| E\rangle]\!] &= [\![v]\!] + [\![E]\!] \\
[\![\langle v \| \tilde{\mu}x.c\rangle]\!] &= 1 + [\![v]\!] + [\![c]\!]\{[\![v]\!]/x\} \\
[\![x]\!] &= x \\
[\![\alpha]\!] &= 0 \\
[\![v \cdot E]\!] &= [\![v]\!] + [\![E]\!] \\
[\![\mathrm{cdr}(E)]\!] &= [\![E]\!] \\
[\![\mathrm{car}(E)]\!] &= [\![E]\!] \\
[\![\mu\alpha.c]\!] &= [\![c]\!] \\
[\![\mu[(x \cdot \alpha).c]]\!] &= [\![c]\!]\{0/x\} \\
[\![\tilde{\mu}x.c]\!] &= [\![c]\!]_{0/x}
\end{aligned}$$

Then, the general property we would show is that $[\![t\{v/x\}]\!] = [\![t]\!]\{[\![v]\!]/x\}$, and then given any assignment ϕ of free variables in the polynomial to integers if $t \rightarrow_{\tilde{\mu}} t'$ then $[\![t]\!](\phi) > [\![t']\!](\phi)$ and so $\tilde{\mu}$ is strongly normalizing.

That $\tilde{\mu}$ and μ are each strongly normalizing on their own is an interesting property of the two-sided sequent calculus. However, as we have given a translation function we do not actually need that property here. *End intermezzo* 1.

Lemma 20. *$\tilde{\mu}$**exp**-normalization commutes across $\mu\tilde{\mu}^F_{cons}$*

Proof. Equivalently, $t \to t'$ by a rule other than $\eta_{\tilde{\mu}}$ implies $t^{\tilde{\mu}} \twoheadrightarrow (t')^{\tilde{\mu}}$. We can prove this by induction on the reduction $t \to t'$. Reductions not at the top of t are nearly all automatic since $-^{\tilde{\mu}}$ is compositional except for the cases of $\langle v \| \mu x.c\rangle$ and $\mu[(x \cdot \alpha).c]$ which are handled by Lemma 18. That leaves only the cases where the reduction happens at the top of t:

– In the case of $\langle \mu\alpha.c \| E\rangle \to_{\mu} c[E/\alpha]$ we have $\langle \mu\alpha.c \| E\rangle^{\tilde{\mu}} \triangleq \langle \mu\alpha.c^{\tilde{\mu}} \| E^{\tilde{\mu}}\rangle \to c^{\tilde{\mu}}\{E^{\tilde{\mu}}/\alpha\}$ and by Lemma 18 $c^{\tilde{\mu}}\{E^{\tilde{\mu}}/\alpha\} =_{\alpha} c\{E/\alpha\}^{\tilde{\mu}}$.

– In the case of $\langle v \| \tilde{\mu}x.c\rangle \to_{\tilde{\mu}} c\{v/x\}$ we have $\langle v \| \tilde{\mu}x.c\rangle^{\tilde{\mu}} \triangleq c^{\tilde{\mu}}\{v^{\tilde{\mu}}/x\}$ which by Lemma 18 is $c\{v/x\}^{\tilde{\mu}}$.

– In the case of $\mu\alpha.\langle v \| \alpha\rangle \to_{\eta_\mu} v$ we have $(\mu\alpha.\langle v \| \alpha\rangle)^{\tilde{\mu}} \triangleq \mu\alpha.\langle v^{\tilde{\mu}} \| \alpha\rangle \to_{\eta_\mu} v^{\tilde{\mu}}$.

– In the case of $\langle v \| \tilde{\mu}x.\langle x \| \tilde{\mu}y.c\rangle\rangle \to_{\eta_{\tilde{\mu}}} \langle v \| \tilde{\mu}y.c\rangle$ we have $\langle v \| \tilde{\mu}x.\langle x \| \tilde{\mu}y.c\rangle\rangle^{\tilde{\mu}} \triangleq c^{\tilde{\mu}}\{v/y\}$ and $\langle v \| \tilde{\mu}y.c\rangle^{\tilde{\mu}} \triangleq c^{\tilde{\mu}}\{v/y\}$.

– In the case of $\mu[(x \cdot \alpha).c] \to_{\text{exp}} \mu\beta.c\{\texttt{car}(\beta)/x, \texttt{cdr}(\beta)/\alpha\}$ we have $\mu[(x \cdot \alpha).c]^{\tilde{\mu}} \triangleq \mu\beta.c^{\tilde{\mu}}\{\texttt{car}(\beta)/x, \texttt{cdr}(\beta)/\alpha\}$ which is $(\mu\beta.c\{\texttt{car}(\beta)/x, \texttt{cdr}(\beta)/\alpha\})^{\tilde{\mu}}$ by Lemma 18.

– In the case of $\texttt{car}(v \cdot E) \to_{\text{car}} v$ we have $\texttt{car}(v \cdot E)^{\tilde{\mu}} \triangleq \texttt{car}(v^{\tilde{\mu}} \cdot E^{\tilde{\mu}}) \to_{\text{car}} v^{\tilde{\mu}}$.

– In the case of $\texttt{cdr}(v \cdot E) \to_{\text{cdr}} E$ we have $\texttt{cdr}(v \cdot E)^{\tilde{\mu}} \triangleq \texttt{cdr}(v^{\tilde{\mu}} \cdot E^{\tilde{\mu}}) \to_{\text{cdr}} E^{\tilde{\mu}}$.

– In the case of $\mathtt{car}(E) \cdot \mathtt{cdr}(E) \to_{\eta.} E$ we have $(\mathtt{car}(E) \cdot \mathtt{cdr}(E))^{\tilde{\mu}} \triangleq \mathtt{car}(E^{\tilde{\mu}}) \cdot \mathtt{cdr}(E^{\tilde{\mu}}) \to_{\eta.} E^{\tilde{\mu}}$.

□

Lemma 21. *$\tilde{\mu}$exp-normalization forms a reflection in $\mu\tilde{\mu}^{F}_{cons}$ of Σ^x.*

Proof. By composition of Theorem 4 and Lemma 20 □

Theorem 5. *$\varsigma\tilde{\mu}$exp-normalization forms a reflection in $\mu\tilde{\mu}^{\to}_{cons}$ of Σ^x.*

Proof. By composition of the reflections in $\mu\tilde{\mu}^{\to}_{\text{cons}}$ of $\mu\tilde{\mu}^{F}_{\text{cons}}$ (Lemma 16) and in $\mu\tilde{\mu}^{F}_{\text{cons}}$ of Σ^x (Lemma 21). □

Confluence of a stack calculus

Now we focus on establishing confluence of the Σ^x-calculus. The Σ^x-calculus is simpler than the full $\mu\tilde{\mu}^{\to}_{\text{cons}}$-calculus, doing away with several constructs and reductions. However, the Σ^x reduction theory still has the complication of the $\eta.$ rule, whose non-left linearity defeats many approaches of establishing confluence.

Surprisingly, we can completely side-step the non-left-linearity problem of surjective call stacks by restricting the $\eta.$ rule. Instead of matching general co-values, we will only bring together a certain form of stuck co-values consisting of a chain of cdr projections out of a co-variable (where $\mathtt{cdr}^n(\alpha)$ means n applications of cdr to α):

$$\mathtt{car}(\mathtt{cdr}^n(\alpha)) \cdot \mathtt{cdr}(\mathtt{cdr}^n(\alpha)) \to_{\eta'.} \mathtt{cdr}^n(\alpha)$$

The restricted $\eta'.$ rule is clearly a particular instance of the more general $\eta.$ rule, and replacing $\eta.$ with $\eta'.$ gives us the simplified Σ'^x reduction theory. However, $\eta'.$

identifies an application of surjectivity that cannot get out of synch, since $\mathtt{cdr}^n(\alpha)$ is a normal form that cannot reduce further. Therefore, the *only* possible reduct of $\mathtt{car}(\mathtt{cdr}^n(\alpha)) \cdot \mathtt{cdr}(\mathtt{cdr}^n(\alpha))$ is $\mathtt{cdr}^n(\alpha)$, and so $\eta'_{\cdot}$ redexes cannot be destroyed by other reductions. This side-steps the problematic aspect of $\eta_{\cdot}$ that we began with. Unfortunately, $\eta'_{\cdot}$ brings up a different problem: $\eta'_{\cdot}$ redexes are not closed under substitution, so *surrounding* μ reductions naïvely destroy inner $\eta'_{\cdot}$ reduction. Miraculously though, the other reduction rules pick up the slack when $\eta'_{\cdot}$ fails, and so Σ^x and Σ'^x end up being equivalent reduction theories.

Lemma 22. *The Σ^x and Σ'^x reduction theories are equivalent: $c \twoheadrightarrow \Sigma^x c'$ if and only if $c \twoheadrightarrow \Sigma'^x c'$, and likewise for (co-)terms. Furthermore, the* $\mathtt{carcdr}\eta_{\cdot}$ *and* $\mathtt{carcdr}\eta'_{\cdot}$ *reduction theories are equivalent.*

Proof. The only difference between the two reduction theories is the rule for surjectivity of call stacks, $\eta_{\cdot}$ in Σ^x versus the restricted $\eta'_{\cdot}$ in Σ'^x. The $\eta'_{\cdot}$ is clearly a particular family of instances of the $\eta_{\cdot}$ rule, $\mathtt{car}(E) \cdot \mathtt{cdr}(E) \to E$, where E must have the form $\mathtt{cdr}^n(\alpha)$. So Σ'^x is simulated by Σ^x step by step.

To go the other way, we only need to show that the unrestricted $\eta_{\cdot}$ rule is simulated by $\mathtt{carcdr}\eta'_{\cdot}$ reduction. To demonstrate the simulation, we consider what happens when we substitute an arbitrary co-value E in for the co-variable of a $\eta'_{\cdot}$ redex. In particular, we demonstrate the following reduction holds:

$$\mathtt{car}(\mathtt{cdr}^n(E)) \cdot \mathtt{cdr}(\mathtt{cdr}^n(E)) \twoheadrightarrow \mathtt{carcdr}\eta'_{\cdot} \mathtt{cdr}^n(E)$$

Crucially, this reduction holds because E can only be some form of call stack and nothing else, so that the general extensionality reduction is simulated by the $\mathtt{car}$ and $\mathtt{cdr}$ computational rules for call stacks. This can be seen by induction on

n and considering cases on the possible co-values for E, where we take any `cdr` projections at the top of E to be included in the chain of projections mentioned by the rule:

- If $E = \alpha$, then we can directly apply the $\eta'_{\cdot}$ rule:

$$\mathtt{car}(\mathtt{cdr}^n(\alpha)) \cdot \mathtt{cdr}(\mathtt{cdr}^n(\alpha)) \to_{\eta'_{\cdot}} \mathtt{cdr}^n(\alpha)$$

- If $E = v \cdot E'$ and $n = 0$ then the simulation follows by `car` and `cdr` reductions:

$$\begin{aligned}\mathtt{car}(\mathtt{cdr}^0(v \cdot E')) \cdot \mathtt{cdr}(\mathtt{cdr}^0(v \cdot E')) &= \mathtt{car}(v \cdot E') \cdot \mathtt{cdr}(v \cdot E') \\ &\to_{\mathtt{car}} v \cdot \mathtt{cdr}(v \cdot E') \to_{\mathtt{cdr}} v \cdot E' = \mathtt{cdr}^0(v \cdot E')\end{aligned}$$

- If $E = v \cdot E'$ and $n = n' + 1$ then the simulation follows by two `cdr` reductions and the inductive hypothesis:

$$\begin{aligned}\mathtt{car}(\mathtt{cdr}^{n+1}(v \cdot E')) \cdot \mathtt{cdr}(\mathtt{cdr}^{n+1}(v \cdot E')) &\to_{\mathtt{cdr}}\to_{\mathtt{cdr}} \mathtt{car}(\mathtt{cdr}^n(E')) \cdot \mathtt{cdr}(\mathtt{cdr}^n(E')) \\ &\twoheadrightarrow_{IH} \mathtt{cdr}^n(E')\end{aligned}$$

- Otherwise, we cannot have $E = \mathtt{cdr}(E')$, because this `cdr` surrounding E' is joined with the chain of `cdr` projections in the rule:

$$\mathtt{car}(\mathtt{cdr}^n(\mathtt{cdr}(E'))) \cdot \mathtt{cdr}(\mathtt{cdr}^n(\mathtt{cdr}(E'))) = \mathtt{car}(\mathtt{cdr}^{n+1}(E')) \cdot \mathtt{cdr}(\mathtt{cdr}^{n+1}(E'))$$

The $\eta.$ rule is the instance of the above reduction where $n = 0$. Thus $\texttt{carcdr}\eta.$ is simulated by $\texttt{carcdr}\eta'.$, so the two are equivalent. Furthermore, Σ^x is simulated by Σ'^x, so the two are equivalent reduction theories. □

Now that we have isolated and defeated the non-left-linearity problem of surjective call stacks within the Σ'^x reduction theory, we can show that it is confluent. From this point, the proof of confluence for the simplified Σ'^x theory is entirely routine, where the only technical detail that needs to be addressed is the fact that reduction is closed under substitution, which we already saw in Lemma 22 when equating the Σ^x and Σ'^x reduction theories.

Lemma 23. *The Σ'^x reduction theory is confluent.*

Proof. We demonstrate confluence by a divide and conquer method, separating the reductions dealing with call stacks (`car`, `cdr`, and $\eta'.$) from the reductions dealing with variable binding (μ, η_μ, and $\eta_{\tilde{\mu}}$). Observe that the $\texttt{carcdr}\eta'.$ reduction theory is confluent since it is subcommutative [3] . Additionally, the $\mu\eta_\mu\eta_{\tilde{\mu}}$ reduction theory is confluent since it is an orthogonal combinatory reduction system. We also observe that these two sub theories commute:

$$\begin{tikzcd}
c_1 \arrow[r, two heads, "\texttt{carcdr}\eta'."] \arrow[d, two heads, "\mu\eta_\mu\eta_{\tilde{\mu}}"] & c_1' \arrow[d, two heads, dashed, "\mu\eta_\mu\eta_{\tilde{\mu}}"] \\
c_2 \arrow[r, two heads, dashed, "\texttt{carcdr}\eta'."] & c_2'
\end{tikzcd}$$

[3] By which we mean that critical pairs come together in zero or one step: whenever $t_1 \leftarrow t \rightarrow t_2$ there exists some t' such that $t_1 \rightarrow^= t' \leftarrow^= t_2$

and similarly for terms and co-terms, which follows by induction from the simpler one-step diagram (where $\to^{=}$ denotes zero or one steps):

$$\begin{tikzcd}
c_1 \arrow[r, "\texttt{carcdr}\eta'_{.}"] \arrow[d, "\mu\eta_\mu\eta_{\tilde{\mu}}"] & c_1' \arrow[d, dashed, "\mu\eta_\mu\eta_{\tilde{\mu}}", "="'] \\
c_2 \arrow[r, dashed, two heads, "\texttt{carcdr}\eta'_{.}"] & c_2'
\end{tikzcd}$$

Confluence of the whole Σ'^{x} reduction theory follows from the Hindley-Rosen lemma.[4] The only challenge is to show that the $\texttt{carcdr}\eta'_{.}$ reduction theory is closed under substitution: that $c \twoheadrightarrow \texttt{carcdr}\eta'_{.} c'$ implies $c\{E/\alpha\} \twoheadrightarrow \texttt{carcdr}\eta'_{.} c'\{E/\alpha\}$, and so on. This is not a trivial property because of the restricted form of the $\eta'_{.}$ rule, whose redexes are destroyed by substitution of co-values for their free co-variable: for example $\texttt{car}(\alpha) \cdot \texttt{cdr}(\alpha) \to_{\eta'_{.}} \alpha$ but $(\texttt{car}(\alpha) \cdot \texttt{cdr}(\alpha))\{x \cdot \beta/\alpha\} = \texttt{car}(x \cdot \beta) \cdot \texttt{cdr}(x \cdot \beta) \not\to_{\eta'_{.}}$. However, the $\texttt{carcdr}\eta'_{.}$ reduction theory is equivalent to the $\texttt{carcdr}\eta_{.}$ reduction theory (Lemma 22) which *is* closed under substitution, so $\texttt{carcdr}\eta'_{.}$ is closed under substitution as well. As a result, we find that $\texttt{carcdr}\eta'_{.}$ and $\mu\eta_\mu\eta_{\tilde{\mu}}$ reductions commute as shown above, and so Σ'^{x} is confluent. □

Theorem 6. *The Σ^x and $\mu\tilde{\mu}^{\to}_{cons}$ reduction theories are confluent.*

Proof. Because Σ'^{x} is confluent (Lemma 23) and Σ'^{x} and Σ^x are equivalent reduction theories (Lemma 22), then Σ^x is also confluent. Furthermore, because there is a Galois connection from $\mu\tilde{\mu}^{\to}_{\text{cons}}$ to Σ^x (Theorem 5) then $\mu\tilde{\mu}^{\to}_{\text{cons}}$ is also confluent (Theorem 3). □

[4]If $\twoheadrightarrow A$ and $\twoheadrightarrow B$ are two confluent rewriting systems that commute then $\twoheadrightarrow A \cup B$ is confluent.

CHAPTER IV

COMPUTING WITH LAZYNESS AND EXTENSIONALITY

This chapter is a version of (Johnson-Freyd et al. (2015)) revised to fit in this dissertation. I was the primary author of that paper and principle designer of the calculi and semantic artifacts appearing here. I am thankful to my coauthors Paul Downen and Zena M. Ariola for their assistance in preparing that publication.

Programming language designers are faced with multiple, sometimes contradictory, goals. On the one hand, it is important that users be able to reason about their programs. On the other hand, we want our languages to support simple and efficient implementations. For the first goal, extensionality is a particularly desirable property. We should be able to use a program without knowing how it was written, only how it behaves. In the previous chapter we explored how to accomodate extensionality in a rewriting theory. However, a rewriting theory, even a confluent one, alone is still insufficient for our second goal of actually running programs. Firstly, while rewriting theories tell us what reductions are permissible, they don't tell which of those reductions we should take. Secondly, rewriting theories aren't necessarily aimed at efficiency: the fact that two reduction sequences eventually reach the same place obscures the fact that one may be much shorter than the other. Thus, for the second goal, it is important to have normal forms, specifying the possible results of execution, that can be efficiently computed. To that end, most implementations stop execution when they encounter a λ-abstraction. This is called *weak-head normalization* and has the advantage that evaluation never encounters a free variable so long as it starts with a closed term. Only needing to deal with closed programs is a great boon for implementers.

Beyond the added simplicity that comes from knowing we won't run into a variable, in a β-reduction $(\lambda x.v)\ v' \ \rightarrow\ v\{v'/x\}$ the fact that v' is closed means that the substitution operation $\{v'/x\}$ need not rename variables in v. More generally, weak-head normalization on closed programs avoids the *variable capture problem*, which is quite convenient for implementations. Beyond reasoning and extensionality there are other desirable goals. In the previous chapter we studied a language with non-strict semantics which can be desirable in practice: programmers wanting to work with infinite data structures, which can improve modularity by separating unbounded producers from consumers (Hughes (1989)), might be more inclined to use a non-strict programming language like Haskell rather than ML.

For these reasons, (1) call-by-name (or call-by-need) evaluation, (2) weak-head normal forms and (3) functional extensionality are all desirable properties to have. However, the combination of all three is inconsistent, representing a trilemma where we can pick at most two. Switching to call-by-value evaluation respects extensionality while computing to weak-head normal forms. But, sticking with call-by-name forces us to abandon one of the other two. There is a fundamental tension between evaluation to weak-head normal form, which always finishes when it reaches a lambda, and the η axiom, which tells us that a lambda might not be done yet. For example, the η law says that $\lambda x.\Omega x$ is the same as Ω, where Ω is the non-terminating computation $(\lambda y.y\ y)(\lambda y.y\ y)$. Yet, $\lambda x.\Omega x$ is a weak-head normal form that is *done* while Ω isn't. Thus, if we want to use weak-head normal forms as our stopping point, the η law becomes suspect. This puts us in a worrisome situation: η-equivalent programs might have different termination behavior. As such, we cannot use essential properties, like our earlier example of the monad laws for state, for reasoning about our programs without the risk of changing a program that works into one that doesn't. The root of our problem is that we combined extensionality with effects, namely non-termination. This is one example of the recurrent tension that arises when we add effects to the call-by-name λ-calculus. For example, with printing as our effect, we would encounter a similar problem when combining η with evaluation to weak-head normal forms. Evaluating the term "$\texttt{print "hello"}; (\lambda y.y)$" to its weak-head normal form would print the string `"hello"`, while evaluating the η-expanded term $\lambda x.(\texttt{print "hello"}; (\lambda y.y))x$ would not.

In this chapter, we address the problem. The solution is to leverage the insights we gained from the study of confluent rewriting in the previous chapter. Namely, that functions perform pattern matching on the calling context — as this view also is the basis for an abstract machine — and that these pattern matches can be replaced with projections. This suggests how to continue computing under a λ-abstraction. We present and relate a series of operational semantics and abstract machines for head reduction motivated by this insight into the nature of λ-abstractions.

After reviewing the small-step operational semantics, big-step operational semantics and Krivine abstract machine for weak-head evaluation (Section 4.1), we turn our investigation to head reduction with the goal of providing the three different styles of semantics. We start our exploration of head reduction with the Krivine abstract machine because it fits with the sequent calculus approach we use throughout, bringing out the *negative* nature of functions (Downen and Ariola (2014)). Functions are not constructed but are *de-constructors*; it is the contexts of functions which are constructed. Replacing pattern matching with projection, we modify our Krivine machine with control to continue evaluation instead of getting

$$
\begin{aligned}
v \in \text{Terms} &::= x \mid v\ v' \mid \lambda x.v \\
E \in \text{EvaluationContexts} &::= \Box \mid E\ v \\
E[(\lambda x.v)\ v'] &\mapsto E[v\{v'/x\}]
\end{aligned}
$$

FIGURE 21. Small-step weak-head reduction

stuck on a top-level lambda. Having gathered intuition on contexts, we focus on the control-free version of this machine. This leads to our first abstract machine for head reduction, and we utilize the syntactic correspondence (Biernacka and Danvy (2007)) to derive an operational semantics for head reduction in the λ-calculus (Section 4.2). The obtained operational semantics is, however, more complicated than would be desirable, and so we define a simpler but equivalent operational semantics. By once again applying the syntactic correspondence, we derive an abstract machine for head reduction which is not based on projections and, by way of the functional correspondence (Ager, Biernacki, Danvy, and Midtgaard (2003)), we generate a big-step semantics which is shown to be equivalent to the big step semantics for head reduction from Sestoft (2002). Finally, we conclude with a more efficient implementation of the projection based machine that coalesces multiple projections into one (Section 4.3).

Weak-Head Evaluation

The semantics of a programming language can come in different flavors. It can be given by creating a mapping from a syntactic domain of programs into an abstract domain of mathematical structures (denotational semantics), or it can be given only in terms of syntactic manipulations of programs (operational semantics). Operational semantics can be further divided into *small-step* or *big-step*. A small-

$$\begin{aligned} \mathrm{N} \in \mathrm{Neutral} &::= x \mid \mathrm{N}\ v \\ \mathrm{WHNF} \in \mathrm{WeakHeadNormalForms} &::= \mathrm{N} \mid \lambda x.v \end{aligned}$$

FIGURE 22. Weak-head normal forms

$$\begin{array}{lll} c \in \mathrm{Commands} & ::= \langle v \| E \rangle \\ E \in \mathrm{CoTerms} & ::= \mathsf{tp} \mid v \cdot E \end{array}$$

$$\begin{array}{ll} \langle v\ v' \| E \rangle & \to \langle v \| v' \cdot E \rangle \\ \langle \lambda x.v \| v' \cdot E \rangle & \to \langle v\{v'/x\} \| E \rangle \end{array}$$

FIGURE 23. Krivine abstract machine

step operational semantics shows step-by-step how a program transitions to the final result. A big-step operational semantics, contrarily, only shows the relation between a program and its final result with no intermediate steps shown, as in an evaluation function. We first start in Figure 21 with a small-step call-by-name semantics for λ-calculus. As before, the *evaluation context*, denoted by E, is simply a term with a hole, written as $\Box$, which specifies where work occurs in a term. The semantics is then given by a single transition rule which specifies how to handle application. Unlike the case of the rewriting theory in the previous chapter, this rule is interpreted as applying *only* at the top of a program. According to this semantics, terms of the form $x\ ((\lambda x.x)y)$ or $\lambda x.x\ ((\lambda x.x)y)$ are not reducible. The final answer obtained is called *weak-head normal form* (whnf for short); a λ-abstraction is in whnf and an application of the form $x\ v_1 \cdots v_n$ is in whnf. We can give a grammar defining a whnf by using the notion of "neutral" from Girard, Taylor, and Lafont (1989a) for λ-calculus terms other than λ-abstractions (see Figure 22).

Note that decomposing a program into an evaluation context and a redex is a meta-level operation in the small-step operational semantics. We can instead make this operation an explicit part of the formalization in an *abstract machine.* In Figure 23 we give the Krivine abstract machine (Krivine (2007)) which can be derived directly from the operational semantics by reifying the evaluation context into a *co-term* (Biernacka and Danvy (2007)). As shown in Chapter II co-values can be understood as standing in for contexts, with tp understood as corresponding to the empty context $\Box$, and the call-stack $v \cdot E$ can be thought of as the context $E[\Box\ v]$, and the formation of a command $\langle v \| E\rangle$ corresponded to plugging v into E to obtain $E[v]$. The reduction rules of the Krivine machine are justified by this correspondence: we have a rule that recognizes that $E[v\ v'] = (E[\Box\ v'])[v]$ and a rule for actually performing β-reduction inside an evaluation context. We can further describe how to run a λ-calculus term in the Krivine machine by plugging the term into an empty context

$$v \rightsquigarrow \langle v \| \mathsf{tp}\rangle$$

then after performing as many evaluation steps as possible, we "readback" a λ-term using the rules

$$\langle v \| v' \cdot E\rangle \hookrightarrow \langle v\ v' \| E\rangle \qquad\qquad \langle v \| \mathsf{tp}\rangle \hookrightarrow v$$

Note that the first rule is only needed if we want to interpret open programs, because execution only terminates in commands of the form $\langle \lambda x.v \| \mathsf{tp}\rangle$ and $\langle x \| E\rangle$, and only the first of these can be closed.

The syntactic correspondence of Biernacka and Danvy (2007) derives the Krivine machine from the small-step operational semantics of weak-head reduction by considering them both as functional programs and applying a series of correctness-preserving program transformations. The interpreter corresponding to the small-step semantics "decomposes" a term into an evaluation context and redex, reduces the redex, "recompose" the resulting term back into its context, and repeats this process until an answer is reached. Recomposing and decomposing always happen in turns, and decomposing always undoes recomposing to arrive again at the same place, so they can be merged into a single "refocus" function that searches for the next redex in-place. This non-tail recursive interpreter can then be made tail-recursive by inlining and fusing refocusing and reduction together. Further simplifications and compression of intermediate transitions in the tail-recursive interpreter yields an implementation of the abstract machine. Equivalence of the two semantic artifacts follows from the equivalence of the associated interpreters, which is guaranteed by construction due to the correctness of each program transformation used. Note that we use $\twoheadrightarrow$ and $\hookrightarrow$ as the reflexive-transitive closures of $\to$ and $\hookrightarrow$ respectively.

Theorem 7 (Equivalence of Krivine machine and small-step operational semantics)**.** *For any λ-calculus terms v, v' the following conditions are equivalent:*

1. *$v \mapsto\!\!\!\!\twoheadrightarrow v'$ such that there is no v'' where $v' \mapsto v''$;*
2. *there exists a command c such that $\langle v \| \mathsf{tp} \rangle \twoheadrightarrow c \hookrightarrow v'$ where there is no c' such that $c \to c'$.*

Let us now turn to the big-step weak-head semantics (see Figure 24). It is not obvious that this semantics corresponds to the small step semantics, a proof of correctness is required. Interestingly, Reynolds's functional correspondence (Ager et al. (2003); Reynolds (1972)) links this semantics to the Krivine abstract

$$\frac{}{x \Downarrow_{wh} x} \qquad \frac{}{\lambda x.v \Downarrow_{wh} \lambda x.v}$$

$$\frac{v_1 \Downarrow_{wh} \lambda x.v_1' \qquad v_1'\{v_2/x\} \Downarrow_{wh} v}{v_1\ v_2 \Downarrow_{wh} v}$$

$$\frac{v_1 \Downarrow_{wh} v_1' \qquad v_1' \not\equiv \lambda x.v}{v_1\ v_2 \Downarrow_{wh} v_1'\ v_2}$$

FIGURE 24. Big-step semantics for weak-head reduction

machine by way of program transformations, similar to the connection between the small-step semantics and abstract machine. The big-step semantics is represented as a compositional interpreter which is then converted into continuation-passing style and defunctionalized (where higher-order functions are replaced with data structures which correspond to co-terms), yielding an interpreter representing the Krivine machine. Correctness follows from construction and is expressed analogously to Theorem 7 by replacing the small-step reduction (i.e. $\mapsto\!\!\!\!\twoheadrightarrow$) with the big-step (i.e. $\Downarrow$).

Theorem 8 (Equivalence of Krivine machine and big-step operational semantics)**.** *For any λ-calculus terms v, v' the following conditions are equivalent:*

1. *$v \Downarrow_{wh} v'$;*

2. *there exists a command c such that $\langle v \| \mathsf{tp} \rangle \twoheadrightarrow c \hookrightarrow v'$ where there is no c' such that $c \to c'$.*

In conclusion, we have seen three different semantics artifacts, small-step, big-step and an abstract machine, which thanks to the syntactic and functional correspondence define the *same* language. Our goal is to provide the three different styles of semantics for a different notion of final result: *head normal forms* (hnf for

$$\begin{aligned} \text{N} \in \text{Neutral} &::= x \mid \text{N}\ v \\ \text{HNF} \in \text{HeadNormalForms} &::= \text{N} \mid \lambda x.\text{HNF} \end{aligned}$$

FIGURE 25. Head Normal Forms

$$\begin{array}{lll} c \in \text{Commands} & ::= \langle v \| E \rangle & \\ v \in \text{Terms} & ::= x \mid v\ v' \mid \mu[(x \cdot \alpha).c] \mid \mu\alpha.c & \\ E \in \text{CoTerms} & ::= \alpha \mid v \cdot E \mid \mathsf{tp} & \end{array}$$

$$\begin{array}{lll} \langle v\ v' \| E \rangle & \to & \langle v \| v' \cdot E \rangle \\ \langle \mu\alpha.c \| E \rangle & \to & c\{E/\alpha\} \\ \langle \mu[(x \cdot \alpha).c] \| v \cdot E \rangle & \to & c\{E/\alpha, v/x\} \end{array}$$

FIGURE 26. Krivine abstract machine for λ-calculus with control

short) (see Figure 25). Note that head reduction unlike weak-head reduction allows execution under a λ-abstraction, e.g. $\lambda x.(\lambda z.z)x$ is a whnf but is not a hnf, whereas $\lambda x.(x(\lambda z.z)x)$ is both a whnf and a hnf.

We have seen how, in the Krivine machine, evaluation contexts take the form of a call-stack consisting of a list of arguments to the function being evaluated. As we have seen in previous chapters, from here abstraction over co-terms is the only necessary ingredient to realizing control operations. We thus arrive at an extension of the Krivine machine with control given in Figure 26.

Surjective Call Stacks

There are two different ways to take apart tuples in programming languages. The first, as we've seen, is to provide functionality to decompose a tuple by matching on its structure, as in the pattern-matching let-term $\texttt{let}\ (x, y) = v'\ \texttt{in}\ v$. By pattern-matching, $\texttt{let}\ (x, y) = (v_1, v_2)\ \texttt{in}\ v$ evaluates to v with v_1 and v_2 substituted for x and y, respectively. The second is to provide primitive projection operations for accessing the components of the tuple, as in $\mathtt{fst}(v)$ and $\mathtt{snd}(v)$. The operation $\mathtt{fst}(v_1, v_2)$ evaluates to v_1 and $\mathtt{snd}(v_1, v_2)$ evaluates to v_2. These two different views on tuples are equivalent in a sense. The `fst` and `snd` projections can be written in terms of pattern-matching

$$\mathtt{fst}(z) = (\mathtt{let}\ (x, y) = z\ \mathtt{in}\ x) \qquad\qquad \mathtt{snd}(z) = (\mathtt{let}\ (x, y) = z\ \mathtt{in}\ y)$$

and likewise, "lazy" pattern-matching can be implemented in terms of projection operations

$$\mathtt{let}\ (x, y) = v'\ \mathtt{in}\ v \to v\{\mathtt{fst}(v')/x, \mathtt{snd}(v')/y\}$$

The projective view of tuples has the advantage of a simple interpretation of extensionality, that the tuple made from the parts of another tuple is the same, by the *surjectivity* law for pairs: $v = (\mathtt{fst}(v), \mathtt{snd}(v))$.

In the previous chapter we saw that by viewing functions as pattern-matching constructs in a programming language, analogous to pattern-matching on tuples, we likewise have another interpretation of functions based on projection. That is, we can replace λ-abstractions or pattern-matching with projection operations, $\mathtt{car}(E)$ and $\mathtt{cdr}(E)$, for accessing the components of a calling context. The operation $\mathtt{car}(v \cdot E)$ evaluates to the argument v and $\mathtt{cdr}(v \cdot E)$ evaluates to the return context E. Analogously to the different views on tuples, this projective view on functional contexts can be used to implement pattern-matching:

$$\langle \mu[(x \cdot \alpha).c] \| E \rangle \to c\{\mathtt{car}(E)/x, \mathtt{cdr}(E)/\alpha\}$$

And since λ-abstractions can be written in terms of pattern-matching, they can also be implemented in terms of $\mathtt{car}(-)$ and $\mathtt{cdr}(-)$:

$$\langle \lambda x.v \| E \rangle = \langle \mu[(x \cdot \alpha).\langle v \| \alpha \rangle] \| E \rangle \to \langle v\{\mathtt{car}(E)/x\} \| \mathtt{cdr}(E) \rangle$$

Therefore, the evaluation contexts of extensional functions are *surjective call-stacks*.

In the case where the co-term is $v \cdot E$, the reduction of pattern-matching into projection justifies our existing reduction rule by performing reduction inside of a command:

$$\langle \mu[(x \cdot \alpha).c] \| v \cdot E \rangle \to c\{\mathtt{car}(v \cdot E)/x, \mathtt{cdr}(v \cdot E)/\alpha\} \twoheadrightarrow c\{v/x, E/\alpha\}$$

However, when working with abstract machines we want to keep reduction at the top of a program, therefore we opt to keep using the rule which combines both steps into one. Instead, rewriting λ-abstractions as projection suggests what to do in the case where E is *not* a stack extension. Specifically, in the case where E is the top-level constant $\mathtt{tp}$ we have the following rule

$$\langle \mu[(x \cdot \alpha).c] \| \mathtt{tp} \rangle \to c\{\mathtt{car}(\mathtt{tp})/x, \mathtt{cdr}(\mathtt{tp})/\alpha\}$$

which has no equivalent in the Krivine machine where the left-hand side of this reduction is stuck.

We thus arrive at another abstract machine in Figure 27 which works just like the Krivine machine with control except that now we have a new syntactic sort of stuck co-terms. The idea behind stuck co-terms is that E is stuck if $\mathtt{cdr}(E)$ does not evaluate further. For example, the co-term $\mathtt{cdr}(\mathtt{cdr}(\mathtt{tp}))$ is stuck, but $\mathtt{cdr}(v \cdot \mathtt{tp})$

$$\begin{array}{lll}
c \in \text{Commands} & ::= & \langle v \| E \rangle \\
v \in \text{Terms} & ::= & x \mid v\ v' \mid \mu\alpha.c \mid \mu[(x \cdot \alpha).c] \mid \mathtt{car}(S) \\
E \in \text{CoTerms} & ::= & \alpha \mid v \cdot E \mid S \\
S \in \text{StuckCoTerms} & ::= & \mathtt{tp} \mid \mathtt{cdr}(S)
\end{array}$$

$$\begin{array}{lll}
\langle v\ v' \| E \rangle & \rightarrow & \langle v \| v' \cdot E \rangle \\
\langle \mu\alpha.c \| E \rangle & \rightarrow & c\{E/\alpha\} \\
\langle \mu[(x \cdot \alpha).c] \| v \cdot E \rangle & \rightarrow & c\{E/\alpha, v/x\} \\
\langle \mu[(x \cdot \alpha).c] \| S \rangle & \rightarrow & c\{\mathtt{car}(S)/x, \mathtt{cdr}(S)/\alpha\}
\end{array}$$

FIGURE 27. Abstract machine for λ-calculus with control (projection based)

is not. Note, however, that we do not consider co-variables to be stuck co-terms since they may not continue to be stuck after substitution. For instance, $\mathtt{cdr}(\alpha)\{v \cdot \mathtt{tp}/\alpha\} = \mathtt{cdr}(v \cdot \mathtt{tp})$ is not stuck, so neither is $\mathtt{cdr}(\alpha)$. This means that we have no possible reduction for the command $\langle \mu[(x \cdot \beta).c] \| \alpha \rangle$, but this is not a problem since we assume to work only with programs which do not have free co-variables as our source language is a λ-calculus. Further, because we syntactically restrict the use of the projection to stuck co-terms, we do not need reduction rules like $\mathtt{car}(v \cdot E) \rightarrow v$ since $\mathtt{car}(v \cdot E)$ is not syntactically well formed in our machine. Intuitively, anytime we would have generated $\mathtt{car}(v \cdot E)$ or $\mathtt{cdr}(v \cdot E)$ we instead eagerly perform the projection reduction in the other rules.

With the projection based approach, we are in a situation where there is always a reduction rule which can fire at the top of a command, except for commands of the form $\langle x \| E \rangle$ or $\langle \mathtt{car}(S) \| E \rangle$, which are done, or $\langle \mu[(x \cdot \beta).c] \| \alpha \rangle$ which is stuck on a free co-variable. Specifically, we are no longer stuck when evaluating a pattern-matching function term at the top-level, i.e. $\langle \mu[(x \cdot \alpha).c] \| \mathtt{tp} \rangle$. As a consequence of this, reduction of co-variable closed commands now respects η. If we have a command of the form $\langle \mu[(x \cdot \alpha).\langle v \| x \cdot \alpha \rangle] \| E \rangle$ with no free co-variables and where x and α do not appear free in v, there are two possibilities depending on

$$
\begin{aligned}
c \in \text{Command} &::= \langle v \| E \rangle \\
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \mid \texttt{car}(S) \\
E \in \text{CoTerms} &::= v \cdot E \mid S \\
S \in \text{StuckCoTerms} &::= \texttt{tp} \mid \texttt{cdr}(S)
\end{aligned}
$$

$$
\begin{array}{lcl}
\langle v\ v' \| E \rangle & \to & \langle v \| v' \cdot E \rangle \\
\langle \lambda x.v \| v' \cdot E \rangle & \to & \langle v\{v'/x\} \| E \rangle \\
\langle \lambda x.v \| S \rangle & \to & \langle v\{\texttt{car}(S)/x\} \| \texttt{cdr}(S) \rangle
\end{array}
$$

FIGURE 28. Head reduction abstract machine for λ-calculus (projection based)

the value of E. Either E is a call-stack $v' \cdot E'$, so we β-reduce

$$\langle \mu[(x \cdot \alpha).\langle v \| x \cdot \alpha \rangle] \| v' \cdot E' \rangle \to \langle v \| v' \cdot E' \rangle$$

or E must be a stuck co-term, so we reduce by splitting it with projections

$$\langle \mu[(x \cdot \alpha).\langle v \| x \cdot \alpha \rangle] \| S \rangle \to \langle v \| \texttt{car}(S) \cdot \texttt{cdr}(S) \rangle$$

meaning that we can continue to evaluate v. Thus, the use of projection out of surjective call-stacks offers a way of implementing call-by-name reduction while also respecting η.

Head Evaluation

We have considered the Krivine machine with control to get an intuition about dealing with co-terms, however, our primary interest in this chapter is to work with the pure λ-calculus. Therefore, from the Krivine machine with control and projection, we derive an abstract machine for the pure λ-calculus which performs head reduction (see Figure 28). Observe that the only co-variable is `tp`, which represents the "top-level" of the program. Here we use $\texttt{car}(-)$ and $\texttt{cdr}(-)$ (as well as the top-level context $\texttt{tp}$) as part of the implementation since they can appear in intermediate states of the machine. However, we still assume that the programs being evaluated are pure λ-terms. Observe that the projection based approach works just like the original Krivine machine (see Figure 23), except in the case where we need to reduce a lambda in a context that is not manifestly a call-stack ($\langle \lambda x.v \| S\rangle$), and in that situation we continue evaluating the body of the lambda. To avoid the problem of having free variables, we replace a variable (x) with the projection into the context ($\texttt{car}(S)$) and indicate that we are now evaluating under the binder with the new context ($\texttt{cdr}(S)$).

Not all commands derivable from the grammar of Figure 28 are sensible; for example, $\langle \texttt{car}(\texttt{tp}) \| \texttt{tp}\rangle$ and $\langle x \| \texttt{car}(\texttt{cdr}(\texttt{tp})).\texttt{cdr}(\texttt{tp})\rangle$ are not. Intuitively, the presence of $\texttt{car}(\texttt{tp})$ means that reduction has gone under a λ-abstraction so the co-term needs to witness that fact by terminating in $\texttt{cdr}(\texttt{tp})$, making $\langle \texttt{car}(\texttt{tp}) \| \texttt{cdr}(\texttt{tp})\rangle$ a legal command. Analogously, the presence of $\texttt{car}(\texttt{cdr}(\texttt{tp}))$ indicates that reduction has gone under two λ-abstractions and therefore the co-term needs to terminate in $\texttt{cdr}(\texttt{cdr}(\texttt{tp}))$ making $\langle x \| \texttt{car}(\texttt{cdr}(\texttt{tp})).\texttt{cdr}(\texttt{cdr}(\texttt{tp}))\rangle$ a legal command. To formally define the notion of a legal command, we make use of the notation $\texttt{cd}^n\texttt{r}(-)$ for n applications of the $\texttt{cdr}$ operation, and we write $\texttt{cd}^n\texttt{r}(\texttt{tp}) \leq \texttt{cd}^m\texttt{r}(\texttt{tp})$ if $n \leq m$ and similarly $\texttt{cd}^n\texttt{r}(\texttt{tp}) < \texttt{cd}^m\texttt{r}(\texttt{tp})$ if $n < m$. We will only consider legal commands, and obviously reduction preserves legal commands.

Definition 1. *A command of the form $\langle v_1 \| v_2 \cdots v_n \cdot S\rangle$ is legal if and only if for $1 \leq i \leq n$ and for every $\texttt{car}(S')$ occurring in v_i, $S' < S$.*

As we saw for the Krivine machine, we have an associated readback function with an additional rule

$$\langle v \| \mathtt{cdr}(S)\rangle \hookrightarrow \langle \lambda x.v\{x/\mathtt{car}(S)\} \| S\rangle$$

for extracting resulting λ-calculus terms after reduction has terminated, where $v\{x/\mathtt{car}(S)\}$ is understood as replacing every occurrence of $\mathtt{car}(S)$ in v with x (which is assumed to be fresh). The readback relation is justified by reversing the direction of the reduction $\langle \lambda x.v \| S\rangle \to \langle v\{\mathtt{car}(S)/x\} \| \mathtt{cdr}(S)\rangle$.

Example 1. If we start with the term $\lambda x.(\lambda y.y)\ x$, we get an evaluation trace

$$\begin{aligned}
\langle \lambda x.(\lambda y.y)\ x \| \mathtt{tp}\rangle &\to \langle (\lambda y.y)\ \mathtt{car}(\mathtt{tp}) \| \mathtt{cdr}(\mathtt{tp})\rangle \\
&\to \langle \lambda y.y \| \mathtt{car}(\mathtt{tp}) \cdot \mathtt{cdr}(\mathtt{tp})\rangle \\
&\to \langle \mathtt{car}(\mathtt{tp}) \| \mathtt{cdr}(\mathtt{tp})\rangle \\
&\hookrightarrow \langle \lambda x.x \| \mathtt{tp}\rangle \\
&\hookrightarrow \lambda x.x
\end{aligned}$$

which reduces $\lambda x.(\lambda y.y)\ x$ to $\lambda x.x$. This corresponds to the intuitive idea that head reduction performs weak-head reduction until it encounters a lambda, at which point it recursively performs head reduction on the body of that lambda. *End example* 1.

Applying Biernacka and Danvy's syntactic correspondence, we reconstruct the small-step operational semantics of Figure 29. Because we want to treat contexts and terms separately, we create a new syntactic category of indices which replaces the appearance of $\mathtt{car}(S)$ in terms, since stuck co-terms correspond to top-level

$$\begin{aligned} t \in \text{TopTerms} &::= v \mid \lambda.t \\ v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \mid i \\ i \in \text{Indices} &:= \text{zero} \mid \text{succ}(i) \\ E \in \text{Contexts} &::= E\ v \mid \Box \\ S \in \text{TopLevelContexts} &::= \lambda.S \mid \Box \end{aligned} \qquad \begin{aligned} \mathit{Count} &: \text{TopLevelContexts} \to \text{Indices} \\ \mathit{Count}(\Box) &= \text{zero} \\ \mathit{Count}(\lambda.S) &= \text{succ}(\mathit{Count}(S)) \end{aligned}$$

$$\begin{aligned} S[E[(\lambda x.v)\ v']] &\mapsto S[E[v\{v'/x\}]] \\ S[\lambda x.v] &\mapsto S[\lambda.v\{\mathit{Count}(S)/x\}] \end{aligned}$$

FIGURE 29. Small-step head reduction corresponding to projection machine

contexts. The function $\mathit{Count}(-)$ is used for converting between top-level contexts and indices. Note that, as we now include additional syntactic objects beyond the pure λ-calculus, we need a readback relation just like in the abstract machine:

$$S[\lambda.v] \hookrightarrow S[\lambda x.v\{x/\mathit{Count}(S)\}]$$

Example 2. Corresponding to the abstract machine execution in Example 1 we have:

$$\begin{aligned} \lambda x.(\lambda y.y)\ x \mapsto & \qquad \text{where } S = \Box \\ \lambda.(\lambda y.y)\ \text{zero} \mapsto & \qquad \text{where } S = \lambda.\Box \text{ and } E = \Box \\ \lambda.\text{zero} \hookrightarrow & \\ \lambda x.x & \end{aligned}$$

Here, the context $\lambda.\Box$ zero corresponds to the co-term `car(tp)` $\cdot$ `cdr(tp)`. *End example* 2.

Because abstract machine co-terms correspond to inside out contexts, we can not just define evaluation contexts as

$$E \in \text{Contexts} ::= E\ v \mid S$$

which would make $((\lambda.S)\ v)$ a context which does not correspond to any co-term (and would allow for reduction under binders). Indeed, the variable-free version of λ-abstraction is only allowed to occur at the top of the program. Thus, composed contexts of the form $S[E[\Box]]$ serve as the exact equivalent of co-terms. Analogously to the notion of legal commands, we have the notion of legal top-level terms, and we will only consider legal terms.

Definition 2. *A top-level term t of the form $S[v]$ is legal if and only if for every index i occurring in v, $i < Count(S)$.*

This operational semantics has the virtue of being reconstructed directly from the abstract machine, which automatically gives a correctness result analogous to Theorem 7.

Theorem 9 (Equivalence of small-step semantics and abstract machine based on projections)**.** *For any index free terms v, v' the following conditions are equivalent:*

1. *$v \mapsto\!\!\!\!\twoheadrightarrow t \hookrightarrow v'$ such that there is no t' where $t \mapsto t'$;*
2. *there exists a command c such that $\langle v \| \mathsf{tp} \rangle \twoheadrightarrow c \hookrightarrow v'$ where there is no c' such that $c \to c'$.*

However, while, in our opinion, the associated abstract machine was extremely elegant, this operational semantics seems unnecessarily complicated. Fortunately, we can slightly modify it to achieve a much simpler presentation.

$$\begin{aligned} v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \\ E \in \text{Contexts} &::= E\ v \mid \Box \\ S \in \text{TopLevelContexts} &::= E \mid \lambda x.S \end{aligned}$$

$$S[(\lambda x.v)\ v'] \mapsto S[v\{v'/x\}]$$

FIGURE 30. Small-step head reduction

At its core, the cause of the complexity of the operational semantics is the use of indices for top-level lambdas. A simpler operational semantics would only use named variables (see Figure 30). To show that the two semantics are indeed equivalent we make use of Sabry and Wadler's *reduction correspondence* (Sabry and Wadler (1997)). For readability, we write $\mapsto_S$ for the evaluation according to Figure 29 and $\mapsto_T$ for the evaluation according to Figure 30. We define the translations $*$ and $\#$ from top-level terms, possibly containing indices, to pure λ-calculus terms, and from pure λ-terms to top-level terms, respectively. More specifically, $*$ corresponds to the normal form of the readback rule and $\#$ to the normal form with respect to non-β $\mapsto_S$ reductions.

Theorem 10. *The reduction systems $\mapsto\!\!\!\!\twoheadrightarrow_S$ and $\mapsto\!\!\!\!\twoheadrightarrow_T$ are sound and complete with respect to each other:*

1. *$t \mapsto\!\!\!\!\twoheadrightarrow_S t'$ implies $t^* \mapsto\!\!\!\!\twoheadrightarrow_T t'^*$;*
2. *$v \mapsto\!\!\!\!\twoheadrightarrow_T v'$ implies $v^\# \mapsto\!\!\!\!\twoheadrightarrow_S v'^\#$;*
3. *$t \mapsto\!\!\!\!\twoheadrightarrow_S t^{*\#}$ and $v^{\#*} = v$.*

Proof. First, we observe that the non-β-reduction of $\mapsto_S$ is inverse to the readback rule, so that for any top-level terms t and t', $t \mapsto_S t'$ by a non-β-reduction if and only if $t' \hookrightarrow t$. Second, we note that for any top-level context $S = \lambda \ldots \lambda.\Box$ and term v of Figure 29, there is a top-level context $S' = \lambda x_0 \dots \lambda x_n.\square$ of Figure 30 and substitution $\sigma = \{x_0/0, \dots, x_n/n\}$ such that $(S[v])^* = S'\{v^\sigma\}$ by induction on S. Similarly, for any top-level context S of Figure 30 and neutral term N, there is a top-level context S' and context E' of Figure 29 such that $(S[N])^\# = S'[E'[N^\sigma]]$ by induction on S.

1. Follows from the fact that each step of $\mapsto_S$ corresponds to zero or one step of $\mapsto_T$:

$$\begin{tikzcd}
S[E[(\lambda x.v)v']] \arrow[r, mapsto, "S"] \arrow[d, "*"] & S[E[v\{v'/x\}]] \arrow[d, "*"] & S[\lambda x.v] \arrow[r, mapsto, "S"] \arrow[d, "*"] & S[\lambda.v\{Count(S)/x\}] \arrow[d, "*"] \\
S'[E^\sigma[(\lambda x.v^\sigma)v'^\sigma]] \arrow[r, mapsto, dashed, "T"] & S'[E^\sigma[v^\sigma\{v'^\sigma/x\}]] & S'[\lambda x.v^\sigma] \arrow[r, equal, "T"] & S'[\lambda x.v^\sigma]
\end{tikzcd}$$

2. Follows from the fact that each step of $\mapsto_T$ corresponds to one step of $\mapsto_S$, similar to the above.

3. Follows from induction on the reductions of the $*$ and $\#$ translations along with the facts that the non-β-reduction of $\mapsto_S$ is inverse to $\hookrightarrow$, $t^{*\#}$ is the normal form of t with respect to non-β-reductions of $\mapsto_S$, and the top-level term v is the normal form of the readback. $\square$

This semantics corresponds to the semantics in Figure 29. where we quotient out by the equivalence on top-level terms

$$S[\lambda x.v] \equiv S[\lambda.v\{Count(S)/x\}]$$

which relates named and index based λ-abstractions. Further we have already defined how to normalize a top term into its underlying term according to this equivalence: we simply use the readback relation $\hookrightarrow$. However, because the simplified semantics does not contain any indices, to make the connection precise we need to only work with those top-level terms in the derived semantics with are *legal* in the sense of not having any free indices.

Definition 3. *A top-level term t is legal if it is of the form $S[v]$ where for every index i which occurs in S, $i < Count(S)$.*

Observe that for an ordinary term v, v legal implies that v is index free and thus a pure λ-term.

Lemma 24. *For every legal top term t there exists a unique legal term v such that $t \hookrightarrow v$. We denote this v as $readback(t)$.*

Proof. $t = S[v]$ for some S and v, so the lemma follows by induction on the length of S. $\square$

Lemma 25. *For legal $S[E[(\lambda x.v)v']]$, $readback(S[E[(\lambda x.v)\ v']]) \mapsto readback(S[E[v\{v'/x\}]])$ according to the rules of the simplified OS.*

Proof. First observe that for all S in the derived operational semantics and S' in the simplified semantics, there exists a S'' in the simplified semantics and ϕ which is a substitution mapping from indices to variables such that $readback(S[S'[E[(\lambda x.v)\ v']]]) = S''[\phi(E[(\lambda x.v)\ v'])]$ and $readback(S[S'[E[v\{v'/x\}]]]) = S''[\phi(E[v\{v'/x\}])]$ as this follows by induction on S. We get the main result by using $\square$ for S' as $S''[\phi(E[(\lambda x.v)\ v])] \mapsto S''[\phi(E[v\{v'/x\}a])]$. $\square$

Lemma 26. *If $S[v]$ is legal and $readback(S[v]) \mapsto v'$ according to the simplified semantics then there exists some t such that $S[v] \mapsto^+ t$ according to the derived semantics and $readback(t) = v'$.*

Proof. Specifically, $S[v] \mapsto\!\!\!\!\rightarrow S'[E[(\lambda x.v'')\ v''']] \mapsto S'[E[v''\{v'''/x\}]]$ with $\mathit{readback}(S'[E[v''\{v'''/x\}]]) = v'$, which we show induction on the length of S.

- In the base case $\mathit{readback}(v) = v$ and so $v \mapsto v'$ means that $v = S''[E[(\lambda x.v'')\ v''']]$ and $v' = S''[E[v''\{v'''/x\}]]$ for some S'' taken from the syntax of the simplified operational semantics. By induction on S'' that means there exists S' and ϕ a substitution mapping variables to indices such that $\mathit{readback}(S'[\phi(E[v''\{v'''/x\}])]) = v'$ with $S''[E[(\lambda x.v'')\ v''']] \mapsto\!\!\!\!\rightarrow S'[\phi(E[(\lambda x.v'')\ v'''])] \mapsto S'[\phi(E[v''\{v'''/x\}])]$.
- In the inductive case $\mathit{readback}(S[\lambda.v]) = \mathit{readback}(S[\lambda x.v\{x/\mathit{Count}(S)\}])$ and we know by the inductive hypothesis that $S[\lambda x.v\{x/\mathit{Count}(S)\}] \mapsto\!\!\!\!\rightarrow S'[E[(\lambda x.v'')\ v''']] \mapsto S'[E[v''\{v'''/x\}]]$ but then, $S[\lambda x.v\{x/\mathit{Count}(S)\}] \mapsto S[\lambda.v]$ and since reduction is deterministic that means that $S[\lambda.v] \mapsto\!\!\!\!\rightarrow S'[E[(\lambda x.v'')\ v''']] \mapsto S'[E[v''\{v'''/x\}]]$ with $\mathit{readback}(S'[E[v''\{v'''/x\}]]) = v'$.

$\square$

Theorem 11 (Equivalence of small-step semantics for head reduction)**.** *For any legal terms v, v' the following conditions are equivalent:*

1. *$v \mapsto\!\!\!\!\rightarrow v'$ such that there is no v'' where $v' \mapsto v''$ (by Figure 30)*
2. *$v \mapsto\!\!\!\!\rightarrow t \hookrightarrow v'$ such that there is no t' where $t \mapsto t'$ (by Figure 29);*

Proof.
- To show that 1. implies 2. suppose $v \mapsto\!\!\!\!\rightarrow v'$ and there is no v'' such that $v' \mapsto v''$. By induction of the reduction sequence and Lemma 26 we know that for all t'' if $\mathit{readback}(t'') \mapsto\!\!\!\!\rightarrow v'$ then there exists some t such that $t'' \mapsto\!\!\!\!\rightarrow t$ and $\mathit{readback}(t) = v'$. Specifically, since $\mathit{readback}(v) = v$, there exists some t such that $v \mapsto\!\!\!\!\rightarrow t$ and $\mathit{readback}(t) = v'$ so $t \hookrightarrow v'$. Now suppose for contradiction

$$\begin{aligned}
c \in \text{Command} &::= \langle v \| E \rangle \\
v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \\
E \in \text{CoTerms} &::= v \cdot E \mid S \\
S \in \text{StuckCoTerms} &::= \mathtt{tp} \mid \mathtt{Abs}(x, S)
\end{aligned}$$

$$\begin{array}{lcl}
\langle v\ v' \| E \rangle & \to & \langle v \| v' \cdot E \rangle \\
\langle \lambda x.v \| v' \cdot E \rangle & \to & \langle v\{v'/x\} \| E \rangle \\
\langle \lambda x.v \| S \rangle & \to & \langle v \| \mathtt{Abs}(x, S) \rangle
\end{array}$$

FIGURE 31. Head reduction abstract machine

that $t \mapsto t'$, then, by Lemma 25, $v' \mapsto \mathit{readback}(t')$ but that contradicts the assumption.

– To show that 2. implies 1. suppose that $v \mapsto\!\!\!\!\to t \hookrightarrow v'$ and there is no t' such that $t \mapsto t'$. Then, by induction on the reduction on $v \mapsto\!\!\!\!\to t$ and by Lemma 25 we know that $\mathit{readback}(v) \mapsto\!\!\!\!\to \mathit{readback}(t)$. But, $t \hookrightarrow v'$ so $\mathit{readback}(t) = \mathit{readback}(v') = v'$. Thus, $v \mapsto\!\!\!\!\to v'$. Now suppose for contradiction that $v' \mapsto v''$. By 26 that means there exists t' such that $\mathit{readback}(v') = v' \mapsto t'$ which contradicts the assumption.

$\square$

Remark 2. The small-step semantics of Figure 30 captures the definition of head reduction given by Barendregt (1984) (Definition 8.3.10) who defines a head redex as follows: If M is of the form

$$\lambda x_1 \cdots x_n.(\lambda x.M_0)M_1 \cdots M_m\ \ ,$$

for $n \geq 0$, $m \geq 1$, then $(\lambda x.M_0)M_1$ is called the *head redex* of M. Note that the context $\lambda x_1 \cdots x_n.\square \cdots M_m$ is broken down into $\lambda x_1 \cdots x_n.\square$ which corresponds

$$\frac{v \Downarrow_{wh} \lambda x.v' \qquad v' \Downarrow_h v''}{v \Downarrow_h \lambda x.v''} \qquad \frac{v \Downarrow_{wh} v' \qquad v' \not\equiv \lambda x.v''}{v \Downarrow_h v'}$$

FIGURE 32. Big-step semantics for head reduction

to a top-level context S and $\Box \cdots M_n$ which corresponds to a context E. Thus, Barendregt's notion of one-step head reduction:

$$M \to_h N \text{ if } M \to^{\Delta} N \quad ,$$

i.e. N results from M by contracting the head redex Δ, corresponds to the evaluation rule of Figure 30. *End remark* 2.

By once again applying the syntactic correspondence, this time to the simplified operational semantics in Figure 30, we derive a new abstract machine (Figure 31) which works by going under lambdas without performing any substitutions in the process. To extract computed λ-calculus terms we add one rule to the readback relation for the Krivine machine.

$$\langle v \| \texttt{Abs}(x, S) \rangle \hookrightarrow \langle \lambda x.v \| S \rangle$$

The equivalence of the abstract machine in Figure 31 and the operational semantics of Figure 30 follows by construction, and is expressed analogously to Theorem 7.

From the abstract machine in Figure 31 we again apply the functional correspondence Ager et al. (2003) to derive the big-step semantics in Figure 32. This semantics utilizes the big-step semantics for weak-head reduction ($\Downarrow_{wh}$) from Figure 24 as part of its definition of head reduction ($\Downarrow_h$). This semantics tells us that the way to evaluate a term to head-normal form is to first evaluate it to weak-

$$\frac{}{x \Downarrow_{sf} x} \qquad \frac{v \Downarrow_{sf} v'}{\lambda x.v \Downarrow_{sf} \lambda x.v'}$$

$$\frac{v_1 \Downarrow_{wh} v_1' \qquad v_1' \not\equiv (\lambda x.v)}{v_1\ v_2 \Downarrow_{sf} v_1'\ v_2} \qquad \frac{v_1 \Downarrow_{wh} \lambda x.v_1' \qquad v_1'\{v_2/x\} \Downarrow_{sf} v}{v_1\ v_2 \Downarrow_{sf} v}$$

FIGURE 33. Sestoft's big-step semantics for head reduction

head normal form, and if the resulting term is a lambda to recursively evaluate the body of that lambda. This corresponds to the behavior of the abstract machine which works just like the Krivine machine (which only reduces to weak head) except in the situation where we have a command consisting of a λ-abstraction and a context which is not a call-stack. Again, the big-step semantics corresponds to the abstract machine by construction: the equivalence can be expressed analogously to Theorem 8.

Interestingly, Sestoft gave a different big-step semantics for head reduction (Sestoft (2002), Figure 33). The only difference between the two is in the way they search for β-redexes. Sestoft's semantics is more redundant by repeating the same logic for function application from the underlying weak-head evaluation, whereas the semantics of Figure 32 only adds a loop for descending under the top-level lambdas produced by weak-head evaluation. However, besides this difference in the search for β-redexes, they are the same: they eventually find and perform β-redexes in exactly the same order. By structural induction one can indeed verify that they generate identical operational semantics.

Theorem 12. *$v \Downarrow_h v'$ if and only if $v \Downarrow_{sf} v'$.*

Coalesced Projection

The projection based machine in Figure 28 has the desirable feature that we will never encounter a variable when we start with a closed program. Additionally, unlike the machine in Figure 31, machine states do not have co-terms that bind variables in their opposing term. On the other hand, it has a certain undesirable property in that when we substitute a projection in for a variable

$$\langle \lambda x.v \| S\rangle \to \langle v\{\texttt{car}(S)/x\} \| \texttt{cdr}(S)\rangle$$

we may significantly increase the size of the term as the stuck co-term has size linear in the number of λ-abstractions we have previously eliminated in this way. The story is even worse in the other direction: the readback relation for the projection machine (and associated operational semantics) depends on performing a deep pattern-match to replace projections with variables

$$\langle v \| \texttt{cdr}(S)\rangle \hookrightarrow \langle \lambda x.v\{x/\texttt{car}(S)\} \| S\rangle$$

which requires matching *all the way* against S.

We now show that we can improve the abstract machine for head evaluation by combining multiple projections into one. We will utilize the macro projection operations $\texttt{pick}^n(-)$ and $\texttt{drop}^n(-)$ which, from the perspective of $\texttt{car}(-)$ and $\texttt{cdr}(-)$, coalesce sequences of projections into a single operation:

$$\texttt{drop}^0(E) \triangleq E \qquad\qquad \texttt{pick}^n(E) \triangleq \texttt{car}(\texttt{drop}^n(E))$$
$$\texttt{drop}^{n+1}(E) \triangleq \texttt{cdr}(\texttt{drop}^n(E))$$

$$\begin{aligned} c \in \text{Command} &::= \langle v \| E \rangle \\ v \in \text{Terms} &::= x \mid v\ v \mid \lambda x.v \mid \mathtt{pick}^n(\mathtt{tp}) \\ E \in \text{CoTerms} &::= v \cdot E \mid \mathtt{drop}^n(\mathtt{tp}) \end{aligned}$$

$$\begin{array}{lcl} \langle v\ v' \| E \rangle & \to & \langle v \| v' \cdot E \rangle \\ \langle \lambda x.v \| v' \cdot E \rangle & \to & \langle v\{v'/x\} \| E \rangle \\ \langle \lambda x.v \| \mathtt{drop}^n(\mathtt{tp}) \rangle & \to & \langle v\{\mathtt{pick}^n(\mathtt{tp})/x\} \| \mathtt{drop}^{n+1}(\mathtt{tp}) \rangle \end{array}$$

FIGURE 34. Coalesced projection abstract machine

Given that our operational semantics is for the pure λ- calculus, $\mathtt{pick}^n(\mathtt{tp})$ and $\mathtt{drop}^n(\mathtt{tp})$ are the only call-stack projections we need in the syntax.

We now construct an abstract machine (Figure 34) for head evaluation of the λ-calculus which is exactly like the Krivine machine with one additional rule utilizing the coalesced projections. As in the machine of Figure 28, the coalesced machine "splits" the top-level into a call-stack and continues. Analogously to Example 1, one has:

$$\begin{aligned} \langle \lambda x.(\lambda y.y)x \| \mathtt{drop}^0(\mathtt{tp}) \rangle &\to \langle (\lambda y.y)(\mathtt{pick}^0(\mathtt{tp})) \| \mathtt{drop}^1(\mathtt{tp}) \rangle \\ &\twoheadrightarrow \langle \mathtt{pick}^0(\mathtt{tp}) \| \mathtt{drop}^1(\mathtt{tp}) \rangle \end{aligned}$$

If the machine terminates with a result like $\langle \mathtt{pick}^n(\mathtt{tp}) \| E \rangle$, it is straightforward to read back the corresponding head normal form:

$$\langle v \| v' \cdot E \rangle \hookrightarrow \langle v\ v' \| E \rangle \qquad \langle v \| \mathtt{tp} \rangle \hookrightarrow v$$

$$\langle v \| \mathtt{drop}^{n+1}(\mathtt{tp}) \rangle \hookrightarrow \langle \lambda x.v\{x/\mathtt{pick}^n(\mathtt{tp})\} \| \mathtt{drop}^n(\mathtt{tp}) \rangle$$

where x is not free in v in the third rule. Note that the substitution $v\{x/\mathtt{pick}^n(\mathtt{tp})\}$ replaces all occurrences of terms of the form $\mathtt{pick}^n(\mathtt{tp})$ inside v

with x. Correctness is ensured by the fact that reduction in the machine with coalesced projections corresponds to reduction in the machine with non-coalesced projections, which in turn corresponds to the operational semantics of head reduction.

Theorem 13 (Equivalence of Coalesced and Non Coalesced Projection Machines)**.**

1. *$c \to c'$ in the coalesced machine if and only if $c \to c'$ in the non-coalesced machine and*

2. *$c \hookrightarrow c'$ in the coalesced machine if and only if $c \hookrightarrow c'$ in the non-coalesced machine*

where conversion is achieved by interpreting $\mathbf{drop}^n(-)$ and $\mathbf{pick}^n(-)$ as macros.

Proof. By cases. Note that the interesting case in each direction is $\langle \lambda x.v\{x/\mathtt{pick}^n(\mathtt{tp})\} \| \mathtt{drop}^n(\mathtt{tp})\rangle \to \langle v \| \mathtt{drop}^{n+1}(\mathtt{tp})\rangle$ which follows since

$$\begin{aligned}
&\langle \lambda x.v\{x/\mathtt{pick}^n(\mathtt{tp})\} \| \mathtt{drop}^n(\mathtt{tp})\rangle \\
&\to \langle v\{x/\mathtt{pick}^n(\mathtt{tp})\}[\mathtt{car}(\mathtt{drop}^n(\mathtt{tp}))/x] \| \mathtt{cdr}(\mathtt{drop}^n(\mathtt{tp}))\rangle \\
&= \langle v\{x/\mathtt{pick}^n(\mathtt{tp})\}[\mathtt{pick}^n(\mathtt{tp})] \| \mathtt{drop}^{n+1}(\mathtt{tp})\rangle \\
&= \langle v \| \mathtt{drop}^{n+1}(\mathtt{tp})\rangle
\end{aligned}$$

□

Our abstract machine, in replacing variables with coalesced sequences of projections, exhibits a striking resemblance to implementations of the λ-calculus based on de Bruijn indices (de Bruijn (1972)). Indeed, our abstract machine can be seen as given a semantic justification for de Bruijn indices as offsets into the call-stack. However, our approach differs from de Bruijn indices in that, in

$$\begin{array}{rl} t \in \text{TopTerms} ::= & \lambda^n.v \\ v \in \text{Terms} ::= & x \mid n \mid v\ v \mid \lambda x.v \\ E \in \text{Contexts} ::= & \Box \mid E\ v \end{array}$$

$$\begin{array}{lcl} \lambda^n.\lambda x.v & \mapsto & \lambda^{n+1}.v\{n/x\} \\ \lambda^n.E[(\lambda x.v)\ v'] & \mapsto & \lambda^n.E[v\{v'/x\}] \end{array}$$

FIGURE 35. Operational semantics of head reduction with partial de Bruijn

general, we still utilize named variables. Specifically, numerical indices are only ever used for representing variables bound in the leftmost branch of the λ-term viewed as a tree. As such, we avoid the complexities of renumbering during β-reduction. However, implementations which do use de Bruijn to achieve a nameless implementation of variables in general have the advantage that the substitution operations $\{\texttt{pick}^n(\texttt{tp})/x\}$ as used in the abstract machine could be replaced with a no-op. The Krivine machine is often presented using de Bruijn despite being used only for computing whnfs. With the addition of our extra rule—which in a de Bruijn setting does not require substitution—we extend it to compute hnfs.

Further, we can extract from our abstract machine an operational semantics which utilizes de Bruijn indices only for top-level lambdas by way of the syntactic correspondence. The resulting semantics, in Figure 35, keeps track of the number of top-level lambdas as part of $\lambda^n.v$ while performing head reduction on v.
As expected, the equivalence between the coalesced abstract machine and the operational semantics is by construction. The coalesced projection machine, and de Bruijn based operational semantics, are essentially equivalent to the projection machine and associated operational semantics. The difference is that by coalescing multiple projections into one, we replace the use of unary natural numbers with abstract numbers which could be implemented efficiently. In this way, we both greatly reduce the size of terms and make the pattern-matching to readback final

results efficient. Numerical de Bruijn indices are an efficient implementation of the idea that λ-abstractions use variables to denote projections out of a call stack.

Towards a Complete Implementation

The desirable combination of weak-head normal forms, call-by-name evaluation, and extensionality is not achievable. This is a fundamental, albeit easily forgettable, constraint in programming language design. However, we have seen that there is a simple way out of this trilemma: replace weak-head normal forms with head normal forms. Moreover, reducing to head normal form is motivated by an analysis of the meaning of λ-abstraction. We took a detour through control so that we could directly reason about not just terms but also their context. This detour taught us to think about the traditional β rule as a rule for pattern-matching on contexts. By analogy to the different ways we can destruct terms we recognized an alternative implementation based on projections out of call-stacks. Projection allows us to run a λ-abstraction even when we don't yet have its argument.

Returning to the pure λ-calculus we derived an abstract machine from this projection-based approach. Using the syntactic correspondence we derived from our abstract machine an operational semantics, and showed how that could be massaged into a simpler operational semantics for head reduction. With a second use of the syntactic correspondence we derived an abstract machine which implemented head reduction in what seems a more traditional way. By the functional correspondence we showed finally that our entire chain of abstract machines and operational semantics correspond to Sestoft's big-step semantics for head reduction. The use of automated techniques for deriving one form of semantics from another makes it easy to rapidly explore the ramifications that follow from a shift of strategy; in our case from weak-head to head reduction. So in the end, we arrive at a variety of different semantics for head reduction, all of which are equivalent and correspond to Barendregt's definition of head reduction for the λ-calculus.

Branching from our projection based machine we derived an efficient abstract machine which coalesces projections, so we may have our cake and eat it too. We escape the trilemma by giving up on weak-head reduction, while still retaining its virtues like avoiding the variable capture problem and keeping all reductions at the top-level. Our machine is identical to the Krivine machine, which performs weak-head evaluation, except for a single extra rule which tells us what to do when we have a λ-abstraction and no arguments to feed it. Further, these changes can be seen as a limited usage of (and application for) de Bruijn indices.

Although our motivating problem and eventual solution were in the context of the pure λ-calculus, our detour through explicit continuations taught us valuable lessons about operational semantics. Having continuations forces us to tackle many operational issues head on because it makes the contexts a first class part of the language. Thus, a meta lesson of this chapter is that adding control can be a helpful aid to designing even pure languages.

In practice, the trilemma is usually avoided by giving up on call-by-name or extensionality rather than weak-head normal forms. However, it may be that effective approaches to head evaluation such as those in this Chapter are of interest even in call-by-value languages or in settings (such as Haskell with `seq`) that lack the η axiom.

$$\begin{aligned} c \in \text{Commands} &::= \langle v|\sigma|E\rangle \\ v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \\ E \in \text{CoTerms} &::= \mathtt{tp} \mid t \cdot E \\ \sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \\ t \in \text{Closures} &::= (v, \sigma) \end{aligned}$$

$$\begin{aligned} \langle v\ v'|\sigma|E\rangle &\to \langle v|\sigma|(v',\sigma)\cdot E\rangle \\ \langle \lambda x.v|\sigma|t\cdot E\rangle &\to \langle v|(x\mapsto t) :: \sigma|E\rangle \\ \langle x|\sigma|E\rangle &\to \langle v|\sigma'|E\rangle \quad \text{where } \sigma(x) = (v,\sigma') \end{aligned}$$

FIGURE 36. Krivine abstract machine with environments

$$\begin{aligned} c \in \text{Commands} &::= \langle v|\sigma|E\rangle \\ v \in \text{Terms} &::= x \mid \lambda x.v \mid v\ v \mid \mathtt{pick}^n(\mathtt{tp}) \\ E \in \text{CoTerms} &::= \mathtt{drop}^n(\mathtt{tp}) \mid t \cdot E \\ \sigma \in \text{Environments} &::= [] \mid (x \mapsto t) :: \sigma \\ t \in \text{Closures} &::= (v, \sigma) \end{aligned}$$

$$\begin{aligned} \langle v\ v'|\sigma|E\rangle &\to \langle v|\sigma|(v',\sigma)\cdot E\rangle \\ \langle \lambda x.v|\sigma|t\cdot E\rangle &\to \langle v|(x\mapsto t) :: \sigma|E\rangle \\ \langle \lambda x.v|\sigma|\mathtt{drop}^n(\mathtt{tp})\rangle &\to \langle v|(x\mapsto(\mathtt{pick}^n(\mathtt{tp}),\sigma)) :: \sigma|\mathtt{drop}^{n+1}(\mathtt{tp})\rangle \\ \langle x|\sigma|E\rangle &\to \langle v|\sigma'|E\rangle \quad \text{where } \sigma(x) = (v,\sigma') \end{aligned}$$

FIGURE 37. Head reduction abstract machine with environments

Finally, we presented our abstract machines using substitutions and have suggested a possible alternative implementation based on de Bruijn indices, but we can also perform closure conversion to handle variables. For example, the machine in Figure 36 is an implementation of the Krivine machine using closures. We assume the existence of a data structure for maps from variable names to closures which supports at least an extension operator :: and variable lookup, which we write using list-like notation. Similarly, the machine in Figure 37 takes our efficient coalesced machine and replaces the use of substitutions with environments. This would correspond to a calculus of explicit substitutions like $\lambda\rho$ (Curien (1988)). However, in general, the problem of how to handle variables is orthogonal to questions of operational semantics, and thus environment based handling of variables can be added after the fact. What the present chapter suggests, however, is that the de Bruijn view of representing variables as numbers is semantically motivated by the desire to make reduction for the call-by-name λ-calculus consistent with extensional principles.

CHAPTER V

BEYOND LAZY FUNCTIONS

So far, we have focused on the λ-calculus and its sequent calculus analogue. In so doing we have made particular choices and limited ourselves down to a subset of features. Everything in the untyped λ-calculus is a function, and we approach those functions using a lazy call-by-name parameter passing technique. However, there are many other data types and evaluation strategies of use in real languages. In a practical language like ML we don't just have functions, but also tuples (represented by product types), booleans (a sum type), numbers, etc. Moreover, ML doesn't use call-by-name but rather call-by-value.

Indeed, even a lazy language like Haskell not only has numerous data types beyond functions, but, in practice, isn't really call-by-name. While the semantics of Haskell is call-by-name, real implementations memoize results in the style of call-by-need. Because Haskell is pure this difference is not observable, but as we are interested in control effects also we should at least take note of it. Moreover, Haskell compilers often use strictness analysis to convert parts of a program to call-by-value for efficiency, and so the actual intermediate languages must incorporate multiple different evaluation strategies.

Consequently, it seems that what is called for in calculi for intermediate languages is to go beyond just lazy functions and be able to support other types and strategies as well.

Thus, a natural question is if it is possible to apply the lessons we learned in the previous chapters to an extended setting with more types or alternative evaluation strategies. We know confluence, for example, is a very desirable property. We were able to construct a reduction theory for lazy functions with extensionality which was confluent by switching to the alternative, projection based account of their computational interpretation. Can we use the same technique with other strategies?

Unfortunately, it appears that extending our approach to other strategies and types runs into a host of challenges as we explore in the next section.

Extensionality with Other Structures and Strategies

Consider the negative form of product (&). Since negative products share the same polarity as functions, they also share the same construction/deconstruction bias: co-terms of products are constructed whereas terms pattern match on their context. This gives us two new co-terms of the form $\pi_1(e)$ and $\pi_2(e)$, which can be interpreted as building a context that requests either the first or second component of a product and sends the reply to the context e. On the other side, we have the term $\mu[\pi_1(\alpha_1).c_1 \mid \pi_2(\alpha_2).c_2]$ which expresses the fact that the product is an object waiting for a request. If the first component is requested then c_1 is executed with α_1 bound to the context inside the request message, as described by the following reduction rule:

$$\langle \mu[\pi_1(\alpha_1).c_1 \mid \pi_2(\alpha_2).c_2] \| \pi_1(e) \rangle \to_{\beta^\&} \langle \mu\alpha_1.c_1 \| e \rangle$$

We have a similar rule to handle the case if the second component is requested. The η rule captures the fact that request-forwarding terms can be simplified:

$$\mu[\pi_1(\alpha_1).\langle v \| \pi_1(\alpha_1) \rangle \mid \pi_2(\alpha_2).\langle v \| \pi_2(\alpha_2) \rangle] \to_{\eta^\&} v$$

Predictably, we see the same conflict between extensionality and control that we had with functions, as expressed by the critical pair:

$$\begin{aligned} v_0 = {} & \mu[\pi_1(\alpha).\langle \mu\beta.c \| \pi_1(\alpha)\rangle \\ & \qquad |\pi_2(\alpha).\langle \mu\beta.c \| \pi_2(\alpha)\rangle] \end{aligned}$$

$$\mu\beta.c \leftarrow_{\eta^{\&}} v_0 \twoheadrightarrow_{\mu} \mu[\pi_1(\alpha).c\{\pi_1(\alpha)/\beta\}|\pi_2(\alpha).c\{\pi_2(\alpha)/\beta\}]$$

However, it is far from obvious how to adapt the solution used for functions to work for products as well. The `exp` rule converts a function—a value that decomposes a call stack—into a μ-abstraction. Unfortunately, a product contains *two* branches instead of one, and it is not clear how to merge two arbitrary branches into a single μ-abstraction. We might be inclined to add the following reduction

$$\mu[\pi_1(\alpha).c\{\pi_1(\alpha)/\beta\}|\pi_2(\alpha).c\{\pi_2(\alpha)/\beta\}] \to \mu\beta.c$$

for α not occurring in c. However, this rule is suspicious since the pattern $\pi_i(\alpha)$ can easily be destroyed. In fact, a similar rule for functions (which corresponds to the backward ν-rule):

$$\mu[(x \cdot \alpha).c\{x \cdot \alpha/\beta\} \to \mu\beta.c$$

gives rise to a simple counterexample:

$$
\begin{array}{l}
v_0 = \mu[x \cdot \alpha].\langle \mu[\_ \cdot \_].\langle \mu[\_ \cdot \_].\langle z \| \delta \rangle \\
\qquad\qquad\qquad \| x \cdot \alpha \rangle \\
\qquad\qquad \| \delta \rangle \\
\mu\beta.\langle \mu[\_ \cdot \_].\langle \mu[\_ \cdot \_].\langle z \| \delta \rangle \leftarrow v_0 \twoheadrightarrow \mu[x \cdot \alpha].\langle \mu[\_ \cdot \_].\langle z \| \delta \rangle \| \delta \rangle \\
\qquad \| \beta \rangle \\
\quad \| \delta \rangle
\end{array}
$$

Adding a positive notion of product, the tensor $\otimes$, is also problematic. On the term side, a positive product is constructed by putting together two terms in the pair (v, v') and deconstructed via the co-term $\tilde{\mu}[(x, y).c]$ which pattern matches on its input term. The β and η rules are :

$$\langle (v, v') \| \tilde{\mu}[(x, y).c] \rangle \to_{\beta\otimes} \langle v \| \tilde{\mu}x.\langle v' \| \tilde{\mu}y.c \rangle \rangle \qquad \tilde{\mu}[(x, y).\langle (x, y) \| E \rangle] \to_{\eta\otimes} E$$

(Since $\tilde{\mu}[(x, y).c]$ is a value in call-by-name, the restriction on the η rule guarantees that a value is not turned into a non-value, similar to the restriction on η for call-by-value functions.) Unfortunately, our proof of confluence relies on co-values always being reducible to simple, normalized structures, containing no reducible sub-commands. However, decomposition of tuples, $\tilde{\mu}[(x, y).c]$, is a co-value which contains arbitrary sub-commands and therefore our proof of confluence does not apply. We conjecture that confluence is lost because in the $\eta.$ redex

$$\texttt{car}(\tilde{\mu}[(x, y).c]) \cdot \texttt{cdr}(\tilde{\mu}[(x, y).c])$$

the two occurrences of command c can get out of synch, destroying the $\eta.$ redex, similar to what is happening in Klop's counterexample of confluence for λ-calculus extended with surjective pairing (Klop and de Vrijer (1989)).

Since call-by-value is dual to call-by-name, similar problems and solutions arise in call-by-value calculi along with similar limitations. Consider the tensor product, which has a stronger η-rule (since any co-term is a co-value in call-by-value):

$$\tilde{\mu}[(x,y).\langle (x,y) \| e\rangle] \to_{\eta^{\otimes}} e$$

whereas the β-rule remains the same. As pairs have dual properties to functions, the dual of the counterexample shown in Equation (3.2) unsurprisingly arises in call-by-value:

$$\tilde{\mu}z.\langle y \| \beta\rangle \leftarrow_{\eta^{\otimes}} \tilde{\mu}[(x,y).\langle (x,y) \| \tilde{\mu}z.\langle y \| \beta\rangle\rangle] \to_{\tilde{\mu}} \tilde{\mu}[(x,y).\langle y \| \beta\rangle]$$

It might help to consider this example in a more natural style from functional programming, where we have the following β- and η-rules for decomposing pairs:[1]

$$\begin{aligned}
\texttt{case } (v_1, v_2) \texttt{ of } (x_1, x_2) \Rightarrow v &\to_{\beta^{\otimes}} \texttt{let } x_1 = v_1 \texttt{ in let } x_2 = v_2 \texttt{ in } v \\
\texttt{case } v \texttt{ of } (x,y) \Rightarrow E[(x,y)] &\to_{\eta^{\otimes}} E[v] \\
\texttt{let } x = v \texttt{ in } v' &\to_{\texttt{let}} v'\{v/x\}
\end{aligned}$$

[1] Note that the stronger reduction $\texttt{case } v \texttt{ of } (x,y) \Rightarrow v'[(x,y)/z] \to v'[v/z]$ is not valid in an untyped call-by-value calculus including non-termination. For example, $\texttt{case } \Omega \texttt{ of} (x,y) \Rightarrow \lambda d.(x,y)$ loops forever, whereas the reduct $\lambda d.\Omega$ does not under call-by-value evaluation. Thus, we restrict the η-rule for pairs to only apply when the decomposed pair appears in the eye of an evaluation context.

In this notation, the critical pair appears as:

$$\texttt{let}\ z = v\ \texttt{in}\ w \leftarrow_{\eta^{\otimes}} \texttt{case}\ v\ \texttt{of}\ (x,y) \Rightarrow \texttt{let}\ z = (x,y)\ \texttt{in}\ w \rightarrow_{\texttt{let}} \texttt{case}\ v\ \texttt{of}\ (x,y) \Rightarrow w$$

We can adopt the same solution for call-by-value pairs as we did for call-by-name functions, by converting patterns to projections and adding a surjectivity reduction:

$$\tilde{\mu}[(x,y).c] \rightarrow \tilde{\mu}z.c\{\texttt{fst}(z)/x, \texttt{snd}(z)/y\}$$

$$\texttt{fst}(V_1,V_2) \rightarrow V_1 \qquad \texttt{snd}(V_1,V_2) \rightarrow V_2 \qquad (\texttt{fst}(V),\texttt{snd}(V)) \rightarrow V$$

However, this solution does not scale in similar ways. It is not obvious how to apply this solution to a disjunctive type. Consider the additive sum type $\oplus$ which comes with two new ways of forming terms, $\iota_1(v)$ and $\iota_2(v)$, and a pattern matching co-term $\tilde{\mu}[\iota_1(x).c_1|\iota_2(y).c_2]$ with the reduction rules

$$\langle \iota_i(v) \| \tilde{\mu}[\iota_1(x_1).c_1 \mid \iota_2(x_2).c_2] \rangle \rightarrow_{\beta\oplus} c_i[v/x_i]$$

$$\tilde{\mu}[\iota_1(x).\langle \iota_1(x) \| e \rangle \mid \iota_2(x).\langle \iota_2(x) \| e \rangle] \rightarrow_{\eta\oplus} e$$

We witness the same counterexample of confluence we had for & in call-by-name. Moreover, adding functions (which are a co-data type) to the calculus breaks confluence of the surjectivity rule for pairs—a previously known and well-studied problem (Klop and de Vrijer (1989))—because values can contain arbitrary reducible sub-terms that can get out of synch:

$$(\texttt{fst}(\lambda x.v), \texttt{snd}(\lambda x.v))$$

We have seen how the interpretation of functional objects through projections out of their call stack resolves the problems with confluence in a lazy λ-calculus with both control and extensionality. Further, we have shown how that interpretation arises naturally from the standpoint that functions are a co-data type, so the observations of functions deserve just as much attention as the functions themselves. Indeed, as our equational correspondence result makes clear, defining functions by projection adds nothing that wasn't already there in the theory from the beginning. The only trick is noticing that λ-abstractions are not the only way to describe functions, and that η-contraction and -expansion are not the only operational interpretations of the η-law.

The projection-based interpretation of functions can be traced back to the call-by-name continuation-passing style transformation of the λ-calculus that validates η (Hofmann and Streicher (2002)). In continuation-passing style, programs are inverted so that function types are explained in terms of a different type of surjective products. Here, we use the sequent calculus as a vehicle for studying the surjective nature of functions in a more direct style, enabled by the equal consideration given to both producers and consumers. Indeed, the sequent calculus explanation of surjective call stacks does not need to introduce other types to explain functions. As presented here, functions are defined independently in their own right without referencing products or negation, following an orthogonal approach to studying logical connectives (Pfenning (2002)). Furthermore, even though $\lambda\mu_{\text{cons}}$ (Nakazawa and Nagai (2014)) is based on the λ-calculus, its streams are logically interpreted as left rules that come from sequent calculus instead of natural deduction. Therefore, we find that in the same way a symmetric treatment for assuming and concluding facts is important in logic, a symmetric treatment for producing and consuming information is important in programming languages as well.

The effectiveness of the projection-based approach for solving the problems with lazy functions makes it enticing to try to extend it to other systems. However, we see that this technique does not extend easily.

The Need for Types

These challenges collectively suggest a limit: we can get confluence for extensional rewriting with only lazy functions with our projection trick, but once we add anything else the trick breaks down. However, perhaps this isn't such a surprise. Our intuition for the η-rule is that it tells us that a function which merely delegates to another function is the same as the function it delegates to. But, as an untyped equation, this rule tells us two *terms* are the same without any requirement they are both actually functions. Instead, it really only makes sense to apply η when we know what we are dealing with is indeed a function.

This suggests an alternative view where we only apply extensionality principles when they are in some sense well typed. Once we start thinking about typed equations though, completeness of the reduction theory can't be so high on our list of priorities as it will often become impossible. Consider the most basic extensionality principle for functions: two functions which always do the same thing on the same input are the same. In a typed setting "the same input" should only refer to well typed inputs. However, this is a problem, since, for instance, determining if two functions agree on the space of all natural numbers is undecidable.

$$\begin{aligned}
c \in \mathit{Command} &::= \langle v \| e \rangle \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \mid \mu[x \cdot \alpha.c] \mid \mu[t\#\alpha.c] \\
e \in \mathit{Co\text{-}Term} &::= \alpha \mid \tilde{\mu}x.c \mid v \cdot e \mid A\#e \\
A, B, C, D \in \mathit{Type} &::= t \mid A \to B \mid \forall t.A
\end{aligned}$$

FIGURE 38. Syntax of classical system F ($\mu\tilde{\mu}_{\mathcal{S}}^{\to\forall}$)

Instead, we can think of extensionality principles as rules for reasoning about programs which reduction should respect but perhaps not enforce. Extensionally equivalent programs should never be observably distinct, but they don't always need to be equated by the reduction theory.

Type systems let us reason safely about extensionality. More than that, they allow us to run programs with multiple data types without worrying about crashes by keeping data of different types separate. By the same token, we can use types to let us integrate different strategies into a single language, keeping items of different strategies separate.

A Typed System

As a first step towards a system using types, let us consider the classical equivalent of the polymorphic λ-calculus given in system F style in Fig. 38.

For now, we consider only the type constructors $\to$ and $\forall$. We view both as *co-data* defined by their destructors: just as a primitive function co-term is formed as a call-stack $v \cdot e$ where v can be thought of as the argument and e the return continuation. Similarly, a primitive co-term of the universal type is an alternative kind of call-stack built by packing a type A together with a co-term e in the form $A\#e$. The term $\mu[x \cdot \alpha.c]$ of the function type, corresponding to a λ-abstraction, pattern-matches on its associated co-term, seeking out a call stack to destruct and

$$\langle \mu\alpha.c \| E\rangle \to_{\mu_{\mathcal{S}}} c\{E/\alpha\} \qquad \langle V \| \tilde{\mu}x.c\rangle \to_{\tilde{\mu}_{\mathcal{S}}} c\{V/x\}$$
$$\mu\alpha.\langle v\|\alpha\rangle \to_{\eta_\mu} v \qquad \tilde{\mu}x.\langle x\|e\rangle \to_{\eta_{\tilde{\mu}}} e$$
$$\langle \mu[x\cdot\alpha.c] \| V\cdot E\rangle \to_{\beta^{\to}_{\mathcal{S}}} c\{V/x, E/\alpha\}$$
$$\langle \mu[t\#\alpha.c] \| A\#E\rangle \to_{\beta^{\forall}_{\mathcal{S}}} c\{A/t, E/\alpha\}$$
$$\begin{aligned}
v\cdot e &\to_{\varsigma^{\to}_{\mathcal{S}}} \tilde{\mu}x.\langle v\|\tilde{\mu}y.\langle x\|y\cdot e\rangle\rangle && \textbf{where}\, v \notin Values_{\mathcal{S}}\\
V\cdot e &\to_{\varsigma^{\to}_{\mathcal{S}}} \tilde{\mu}x.\langle \mu\beta.\langle x\|V\cdot\beta\rangle\|e\rangle && \textbf{where}\, e \notin Co\text{-}Values_{\mathcal{S}}\\
A\#e &\to_{\varsigma^{\forall}_{\mathcal{S}}} \tilde{\mu}x.\langle \mu\beta.\langle x\|A\#\beta\rangle\|e\rangle && \textbf{where}\, e \notin Co\text{-}Values_{\mathcal{S}}
\end{aligned}$$

FIGURE 39. Strategy parameterized rewriting for classical system F ($\mu\tilde{\mu}^{\to\forall}_{\mathcal{S}}$)

$$\begin{aligned}
V \in Values_{\mathcal{N}} &::= v\\
E \in Co\text{-}Values_{\mathcal{N}} &::= \alpha \mid v\cdot E \mid A\#E
\end{aligned}$$

FIGURE 40. (Co-)values in the call-by-name strategy $\mathcal{N}$

bind to x and α. Similarly, the term $\mu[t\#\alpha.c]$ of the universal type, akin to a Λ-abstraction in system F, pattern matches on type-specializing co-terms.

Parameterized Reduction Theory

Following Downen and Ariola (2014) our reduction rules shown in Fig. 39 are parameterized by a *strategy* consisting of a set of terms (denoted with the meta-variable V) and a set of co-terms (denoted with E). For example, the reduction rule

$$\langle V\|\tilde{\mu}x.c\rangle \to_{\tilde{\mu}_{\mathcal{S}}} c\{V/x\}$$

substitutes a value V of the strategy $\mathcal{S}$ into the variable of an input abstraction. Conversely, the reduction rule

$$\langle \mu\alpha.c\|E\rangle \to_{\mu_{\mathcal{S}}} c\{E/\alpha\}$$

$$\begin{aligned} V \in \mathit{Values}_{\mathcal{V}} &::= x \mid \mu[x \cdot \alpha.c] \mid \mu[t\#\alpha.c] \\ E \in \mathit{Co\text{-}Values}_{\mathcal{V}} &::= e \end{aligned}$$

FIGURE 41. (Co-)values in the call-by-value strategy $\mathcal{V}$

$$\begin{aligned} V \in \mathit{Values}_{\mathcal{LV}} &::= x \mid \mu[x \cdot \alpha.c] \mid \mu[t\#\alpha.c] \\ E \in \mathit{Co\text{-}Values}_{\mathcal{LV}} &::= \alpha \mid \tilde{\mu}x.C[\langle x \| E \rangle] \mid v \cdot E \mid A\#E \\ C \in \mathit{Contexts}_{\mathcal{LV}} &::= \Box \mid \langle v \| \tilde{\mu}y.C \rangle \end{aligned}$$

FIGURE 42. (Co-)values in the call-by-need strategy $\mathcal{LV}$

substitutes a co-value E of $\mathcal{S}$ into the co-variable of an output abstraction. Restricting these rules to only fire with (co-)values allows us to avoid the essential critical pair of classical logic:

$$c_1\{\tilde{\mu}x.c_2/\alpha\} \leftarrow_{\mu_{\mathcal{S}}} \langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle \rightarrow_{\tilde{\mu}_{\mathcal{S}}} c_2\{\mu\alpha.c_1/x\} \ .$$

Different choices for the sets of values and co-values yield different evaluation strategies. The call-by-name strategy in Fig. 40 treats *all* terms as values (giving it the strongest possible notion of substitution for terms) and restricts the notion of co-values down to just (hereditary) call-stacks. Conversely, the call-by-value strategy in Fig. 41 is formed by taking the largest possible set of co-values (giving it the strongest possible notion of substitution for co-terms) and restricting the set of values down to only variables and pattern-matching abstractions.

Note that the notion of (co-)value appears not just in the substitution rules: the lifting ς rules—originally due to Wadler (2003)—appear in our reduction theory to lift unevaluated components out of call-stacks so that they can be computed first, much as the components of a (call-by-value) tuple are computed down to values one at a time before the tuple is constructed and returned. These rules are essential to ensure that computation does not get stuck, and *must* be restricted to only fire with non-(co-)values for us to have any hope of strong normalization, avoiding spurious loops which simply lift out part of a structure and substitute it back in again.

While call-by-value and call-by-name represent extremes in the design space, they are not the only possible confluent strategies. Another is the call-by-need "lazy value" strategy in Fig. 42 which starts with the same, restricted, notion of values as in call-by-value, but has a much smaller set of co-values: call-by-name co-values together with those input abstractions which *demand* their argument. Thus, it corresponds to an evaluator where in $\langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle$ execution initially proceeds like call-by-name by executing c_2 and treating $\mu\alpha.c_1$ as a delayed computation, until such a time as x is needed, at which point control jumps to c_1. Then, if $\mu\alpha.c_1$ ever reduces to a value, that value could be substituted for x back into c_2.

Compared to call-by-name and call-by-value, call-by-need's nuanced approach comes at a cost: delayed non-values stick around indefinitely when they are never needed. However, call-by-need is of deep interest in understanding lazy languages with effects—such as R, or Haskell extended with `unsafePerformIO`—and has clear efficiency benefits.

While a strategy can be used to resolve the critical pair and bring about confluence, it is important to also consider non-confluent strategies. One of these, which we call $\mathcal{U}$, simply treats every (co-)term as a (co-)value and is the most basic strategy for reducing classical proofs as used in a number of calculi (Barbanera and Berardi (1994); Curien and Herbelin (2000)), as well as Gentzen's original

$$\frac{}{\Gamma, x : A \vdash_\Theta x : A \mid \Delta}\,Var \quad \frac{}{\Gamma \mid \alpha : A \vdash_\Theta \alpha : A, \Delta}\,Co\text{-}Var$$

$$\frac{c : (\Gamma \vdash_\Theta \alpha : A, \Delta)}{\Gamma \vdash_\Theta \mu\alpha.c : A \mid \Delta}\,Act \quad \frac{c : (\Gamma, x : A \vdash_\Theta \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash_\Theta \Delta}\,Co\text{-}Act$$

$$\frac{\Gamma \vdash_\Theta v : A \mid \Delta \quad FV(A) \subseteq \Theta \quad \Gamma \mid e : A \vdash_\Theta \Delta}{\langle v \| e \rangle : (\Gamma \vdash_\Theta \Delta)}\,Cut$$

$$\frac{\Gamma \vdash_\Theta v : A \mid \Delta \quad \Gamma \mid e : B \vdash_\Theta \Delta}{\Gamma \mid v \cdot e : A \to B \vdash_\Theta \Delta}\,FunLeft$$

$$\frac{c : (\Gamma, x : A \vdash_\Theta \alpha : B, \Delta)}{\Gamma \vdash_\Theta \mu[x \cdot \alpha.c] : A \to B \mid \Delta}\,FunRight$$

$$\frac{\Gamma \mid e : B\{A/t\} \vdash_\Theta \Delta \quad FV(A) \subseteq \Theta}{\Gamma \mid A\#e : \forall t.B \vdash_\Theta \Delta}\,ForallLeft$$

$$\frac{c : (\Gamma, \vdash_{\Theta,t} \alpha : B, \Delta)}{\Gamma \vdash_\Theta \mu[t\#\alpha.c] : \forall t.B \mid \Delta}\,ForallRight$$

FIGURE 43. The type system for classical system F

formulation of cut elimination (Gentzen (1935)). While useless as an equational theory, rewriting with $\mathcal{U}$ is interesting since it allows *any* substitution and is unbiased. Moreover, we may use it when it does not matter *what* normal form we get so long as we get one. However, we caution that this non-confluent strategy should not be thought of as the most general since it completely forecloses the ς reductions which are essential to all other strategies. Indeed, there is no most-general strategy as any time the set of possible μ and $\tilde{\mu}$ reductions are extended the set of possible ς reductions must shrink, and vice versa.

Typing

The typing rules given in Fig. 43 are essentially the rules of system F but presented in a sequent calculus style, giving a term assignment to the second-order classical sequent calculus.

Since we assume the Barendregt convention, we treat contexts as sets rather than lists. When we write $\Delta, x : A$ (and similar) we require that x not already appear in Δ. Moreover we consider a sequent to be well formed only if all of the free type variables on both the left and right side of the turnstile come from Θ. Only derivations where all sequents are well formed are considered proofs. This imposes the standard side condition on the $ForallRight$ rule that the new type variable introduced above the line not appear free below the line. Our rules for Cut and $ForallLeft$, which are the only places where new types can appear going up the proof, include the condition that these new types only use free variables in Θ. This ensures that any derivation whose last line is well formed is a proof.

Adding Additional Type Formers

The classical System F system described above includes functions and type abstraction, but no other features. However, now that we have a typed system it is easy to incorporate additional type formers. Those include not just negative co-data types, like $\forall$ and $\rightarrow$, but also positive data types.

Positive Sum

For example, the positive sum type is given by extending the syntax with two additional term formers

$$v ::= \ldots \mid \iota_1(v) \mid \iota_2(v)$$

as well as a pattern matching co-term construct.

$$e ::= \ldots \mid \tilde{\mu}[\iota_1(x).c_1; \iota_2(y).c_2]$$

The corresponding typing rules would be

$$\frac{\Gamma \vdash_\Theta v : T \mid \Delta}{\Gamma \vdash_\Theta \iota_1(v) : T \oplus S \mid \Delta} \quad \frac{\Gamma \vdash_\Theta v : S \mid \Delta}{\Gamma \vdash_\Theta \iota_2(v) : T \oplus S \mid \Delta}$$

$$\frac{c_1 : \Gamma, x : T \vdash_\Theta \Delta \quad c_2 : \Gamma, y : S \vdash_\Theta \Delta}{\Gamma \mid \tilde{\mu}[\iota_1(x).c_1; \iota_2(y).c_2] : T \oplus S \vdash_\Theta \Delta}$$

with a parmetric computational inerpretation given by a pair of β rules

$$\langle \iota_1(V) \| \tilde{\mu}[\iota_1(x).c_1; \iota_2(y).c_2] \rangle \to_{\beta_{\iota_1}} c_1\{V/x\}$$

$$\langle \iota_2(V) \| \tilde{\mu}[\iota_1(x).c_1; \iota_2(y).c_2] \rangle \to_{\beta_{\iota_2}} c_2\{V/y\}$$

as well as lifting rules which can fire whenever v is not a value

$$\iota_1(v) \to_\varsigma \mu\alpha.\langle v \| \tilde{\mu}x.\langle \iota_1(x) \| \alpha \rangle \rangle$$

$$\iota_2(v) \to_\varsigma \mu\alpha.\langle v \| \tilde{\mu}x.\langle \iota_2(x) \| \alpha \rangle \rangle$$

Positive Sum

Another data type we might consider adding would be the positive product, $\otimes$, representing a pair of objects in memory.

$$v ::= \ldots \mid (v_1, v_2)$$

$$e ::= \ldots \mid \tilde{\mu}[(x, y).c]$$

$$\frac{\Gamma \vdash_\Theta v_1 : T \mid \Delta \quad \Gamma \vdash_\Theta v_2 : S \mid \Delta}{\Gamma \vdash_\Theta (v_1, v_2) : T \otimes S \mid \Delta}$$

$$\frac{c : \Gamma, x : T, y : S \vdash_\Theta \Delta}{\Gamma \mid \tilde{\mu}[(x, y).c] : T \otimes S \vdash_\Theta \Delta}$$

$$\begin{aligned}
\langle (V_1, V_2) \| \tilde{\mu}[(x, y).c] \rangle &\to_\beta c\{V_1/x, V_2/y\} \\
(v_1, v_2) &\to_\varsigma \mu\alpha.\langle v_1 \| \tilde{\mu}x.\langle (x, v_2) \| \alpha \rangle \rangle \\
(V_1, v_2) &\to_\varsigma \mu\alpha.\langle v_2 \| \tilde{\mu}x.\langle (V_1, x) \| \alpha \rangle \rangle
\end{aligned}$$

Negative Product

$\otimes$ and $\oplus$ are both positive data types, defined by constructors and observed by pattern matching. Conversly, $\to$ and $\forall$ are negative co-data types, defined by destructors and constructed by pattern matching on the calling context. There are other co-data types too. For example, we can take the exact dual of the $\oplus$ type which is an alternative kind of product formed not as a pair in memory (as in $\otimes$) but rather as an object able to respond to two different kinds of messages.

$$\begin{aligned}
e &::= \ldots \mid \mathtt{Fst}(e) \mid \mathtt{Snd}(e) \\
v &::= \ldots \mid \mu[\mathtt{Fst}(\alpha).c_1; \mathtt{Snd}(\beta).c_2]
\end{aligned}$$

$$\frac{\Gamma \mid e : T \vdash_\Theta \Delta}{\Gamma \mid \mathtt{Fst}(e) : T \& S \vdash_\Theta \Delta} \quad \frac{\Gamma \mid e : S \vdash_\Theta \Delta}{\Gamma \mid \mathtt{Snd}(e) : T \& S \vdash_\Theta \Delta}$$

$$\frac{c_1 : \Gamma \vdash_\Theta \alpha : T, \Delta \quad c_2 : \Gamma \vdash_\Theta \beta : S, \Delta}{\Gamma \vdash_\Theta \mu[\mathtt{Fst}(\alpha).c_1; \mathtt{Snd}(\beta).c_2] : t \& S \mid \Delta}$$

$$\langle \mu[\mathtt{Fst}(\alpha).c_1; \mathtt{Snd}(\beta).c_2] \| \mathtt{Fst}(E) \rangle \to_\beta c_1\{E/\alpha\}$$

$$\langle \mu[\mathtt{Fst}(\alpha).c_1; \mathtt{Snd}(\beta).c_2] \| \mathtt{Snd}(E) \rangle \to_\beta c_2\{E/\beta\}$$

$$\mathtt{Fst}(e) \to_\varsigma \tilde{\mu}x.\langle \mu\alpha.\langle x \| \mathtt{Fst}(\alpha) \rangle \| e \rangle$$

$$\mathtt{Snd}(e) \to_\varsigma \tilde{\mu}x.\langle \mu\alpha.\langle x \| \mathtt{Snd}(\alpha) \rangle \| e \rangle$$

Negative Sums

Less familiar is the dual to $\otimes$ which is an alternative kind of sum, formed not as a tagged union, but instead as an object taking a single message with two different options as to what to do next.

$$e ::= \ldots \mid (e_1, e_2)$$

$$v ::= \ldots \mid \mu[(\alpha, \beta).c]$$

$$\frac{\Gamma \mid e_1 : T \vdash_\Theta \Delta \quad \Gamma \mid e_2 : S \vdash_\Theta \Delta}{\Gamma \mid (e_1, e_2) : T \parr S \vdash_\Theta \Delta}$$

$$\frac{c : \Gamma \vdash_\Theta \alpha : S, \beta : T, \Delta}{\Gamma \vdash_\Theta \mu[(\alpha, \beta).c] : T \parr S \mid \Delta}$$

$$\langle \mu[(\alpha, \beta).c] \| (E_1, E_2) \rangle \to_\beta c\{E_1/\alpha, E_2/\beta\}$$

$$(e_1, e_2) \to_\varsigma \tilde{\mu}x.\langle \mu\alpha.\langle x \| (\alpha, e_2) \rangle \| e_1 \rangle \qquad e_1 \notin \text{Co-Values}$$

$$(E_1, e_2) \to_\varsigma \tilde{\mu}x.\langle \mu\alpha.\langle x \| (E_1, \alpha) \rangle \| e_e \rangle \qquad e_2 \notin \text{Co-Values}$$

Combining Strategies

Not only can we leverage the type system to have multiple kinds of things beyond just functions, but also multiple different strategies. To make this work, we must prohibit the direct interaction of (co-)terms of different strategies. However, we want to enable the mixing of strategies in different parts of the program. Moreover, as much as possible we would like the reduction rules to be untyped. We thus partition the syntax given in Fig. 44 into sorts parameterized by a set of strategy symbols (metavariable $\mathcal{S}$), such that a command $\langle v_S \| e_S \rangle$ is formed from (co-)terms from the same strategy. Here, the different v_S and $v_{S'}$ are the meta variables refering to different syntactic sorts (assuming S and S' are distinct). (Co-)variables and stack constructors $(\cdot)$ are always tagged with their strategy ($x^{\mathcal{S}}$ and $\alpha^{\mathcal{S}}$), and the strategy of a call stack might differ from its components.

The reduction rules shown in Fig. 45 are parameterized by a family of strategies. As a parameter to the rewriting theory, the family of strategies must define a set of values and a set of co-values for each of the strategy symbols. We use unsubscripted metavariables to refer to the union over the scripted versions, so for example we simply write $\langle \mu\alpha^{\mathcal{S}}.c \| E \rangle \to_\mu c\{E/\alpha^{\mathcal{S}}\}$ and $\langle V \| \tilde{\mu}x^{\mathcal{S}}.c \rangle \to_{\tilde{\mu}} c\{V/x^{\mathcal{S}}\}$ which, by the syntactic requirement that the two parts of the command are from the same strategy, also restricts the (co-)term to the same strategy as the (co-)variable. The strategy not only restricts what rules may fire, but also what new names are created in the ς rules. Call-by-value, call-by-name and call-by-need (here called “lazy value”) computations are thus characterized by the strategies $\mathcal{V}$, $\mathcal{N}$, and $\mathcal{LV}$ of Fig. 46. In addition, the strategy $\mathcal{U}$ treats every (co-)term as a (co-)value. As a consequence of lifting, $\mathcal{U}$ should not be thought of as the most general strategy since it completely forecloses ς reductions which are essential to

$$\begin{array}{rl} c \in \mathit{Command} ::= \langle v_{\mathcal{S}} \| e_{\mathcal{S}} \rangle & A_{\mathcal{S}}, B_{\mathcal{S}} \in \mathit{Types}_{\mathcal{S}} ::= t^{\mathcal{S}} \mid A_{\mathcal{S}_1} \to^{\mathcal{S}} B_{\mathcal{S}_2} \mid \forall^{\mathcal{S}} t^{\mathcal{S}_1}.A_{\mathcal{S}_2} \\ v_{\mathcal{S}} \in \mathit{Terms}_{\mathcal{S}} ::= x^{\mathcal{S}} \mid \mu\alpha^{\mathcal{S}}.c \mid \mu[x^{\mathcal{S}_1} \circledS \alpha^{\mathcal{S}_2}.c] \mid \mu[t^{\mathcal{S}_1} \circledS \alpha^{\mathcal{S}_2}.c] & \\ e_{\mathcal{S}} \in \mathit{Co\text{-}Terms}_{\mathcal{S}} ::= \alpha^{\mathcal{S}} \mid \tilde{\mu}x^{\mathcal{S}}.c \mid v \circledS e \mid A \circledS e & \end{array}$$

FIGURE 44. Syntax of classical system F with multiple strategies ($\mu\tilde{\mu}^{\to\forall}_{\mathcal{S}^\star}$)

$$\begin{array}{ll} \langle \mu[x^{\mathcal{S}_1} \circledS \alpha^{\mathcal{S}_2}.c] \| V_{\mathcal{S}_1} \circledS E_{\mathcal{S}_2} \rangle \to_{\beta^{\to}} c\{V_{\mathcal{S}_1}/x^{\mathcal{S}_1}, E_{\mathcal{S}_2}/\alpha^{\mathcal{S}_2}\} & \langle \mu\alpha^{\mathcal{S}}.c \| E \rangle \to_{\mu} c\{E/\alpha^{\mathcal{S}}\} \\ \langle \mu[t^{\mathcal{S}_1} \circledS \alpha^{\mathcal{S}_2}.c] \| A_{\mathcal{S}_1} \circledS E_{\mathcal{S}_2} \rangle \to_{\beta^{\forall}} c\{A_{\mathcal{S}_1}/t^{\mathcal{S}_1}, E_{\mathcal{S}_2}/\alpha^{\mathcal{S}_2}\} & \langle V \| \tilde{\mu}x^{\mathcal{S}}.c \rangle \to_{\tilde{\mu}} c\{V/x^{\mathcal{S}}\} \end{array}$$

$$\begin{array}{l} \mu\alpha^{\mathcal{S}}.\langle v \| \alpha^{\mathcal{S}} \rangle \to_{\eta_\mu} v \\ \tilde{\mu}x^{\mathcal{S}}.\langle x^{\mathcal{S}} \| e \rangle \to_{\eta_{\tilde{\mu}}} e \end{array} \quad \begin{array}{rll} v_{\mathcal{S}_1} \circledS e & \to_{\varsigma^{\to}} \tilde{\mu}x^{\mathcal{S}}.\langle v_{\mathcal{S}_1} \| \tilde{\mu}y^{\mathcal{S}_1}.\langle x^{\mathcal{S}} \| y^{\mathcal{S}_1} \circledS e \rangle\rangle & \textbf{where}\, v_{\mathcal{S}_1} \notin \mathit{Value}_{\mathcal{S}_1} \\ V \circledS e_{\mathcal{S}_2} & \to_{\varsigma^{\to}} \tilde{\mu}x^{\mathcal{S}}.\langle \mu\beta^{\mathcal{S}_2}.\langle x^{\mathcal{S}} \| V \circledS \beta^{\mathcal{S}_2} \rangle \| e_{\mathcal{S}_2} \rangle & \textbf{where}\, e_{\mathcal{S}_2} \notin \mathit{Co\text{-}Value}_{\mathcal{S}_2} \\ A \circledS e_{\mathcal{S}_2} & \to_{\varsigma^{\forall}} \tilde{\mu}x^{\mathcal{S}}.\langle \mu\beta^{\mathcal{S}_2}.\langle x^{\mathcal{S}} \| A \circledS \beta^{\mathcal{S}_\in} \rangle \| e_{\mathcal{S}_2} \rangle & \textbf{where}\, e_{\mathcal{S}_2} \notin \mathit{Co\text{-}Value}_{\mathcal{S}_2} \end{array}$$

FIGURE 45. Rewriting theory parameterized by multiple strategies ($\mu\tilde{\mu}^{\to\forall}_{\mathcal{S}^\star}$)

all other strategies. There is no most-general strategy: any time the set of possible μ and $\tilde{\mu}$ reductions grows the set of possible ς reductions must shrink, and vice versa. Additionally, the call-by-push-value (Levy (2001)) function space (a.k.a. the polarized "primordial" function space in Zeilberger (2009)) is characterized as $A_{\mathcal{V}} \to^{\mathcal{N}} B_{\mathcal{N}}$, the function and its result are lazy but the argument is eagerly evaluated.

Finally the type system Fig. 47 is much the same as the type system before, only now it keeps track of strategies. Consequently, there are multiple kinds of type variables, however, the strategy of type variables is largely handled syntactically.

We could extend this system with all of the other type formers given earlier, but in strategy parametric forms. Doing so would be routine. However, we prefer to focus on just the multi-strategy system F type system as an object of study, as it reduces the number of concepts. We must assure ourselves, however, that all our proofs extend to the other type formers if we want to add them. In this way, we

$$V_{\mathcal{V}} \in Values_{\mathcal{V}} ::= x^{\mathcal{V}} \mid \mu[x^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{V}$}} \alpha^{\mathcal{S}_2}.c] \mid \mu[t^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{V}$}} \alpha^{\mathcal{S}_2}.c] \qquad V_{\mathcal{N}} \in Values_{\mathcal{N}} ::= v_{\mathcal{N}}$$

$$E_{\mathcal{N}} \in Co\text{-}Values_{\mathcal{N}} ::= \alpha^{\mathcal{N}} \mid V \mathbin{\textcircled{$\scriptstyle\mathcal{N}$}} E \mid A \mathbin{\textcircled{$\scriptstyle\mathcal{N}$}} E \qquad E_{\mathcal{V}} \in Co\text{-}Values_{\mathcal{V}} ::= e_{\mathcal{V}}$$

$$V_{\mathcal{LV}} \in Values_{\mathcal{LV}} ::= x^{\mathcal{LV}} \mid \mu[x^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{LV}$}} \alpha^{\mathcal{S}_2}.c] \mid \mu[t^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{LV}$}} \alpha^{\mathcal{S}_2}.c]$$

$$E_{\mathcal{LV}} \in Co\text{-}Values_{\mathcal{LV}} ::= \alpha^{\mathcal{LV}} \mid \tilde{\mu}x^{\mathcal{LV}}.C[\langle x^{\mathcal{LV}} \| E_{\mathcal{LV}} \rangle] \mid v \mathbin{\textcircled{$\scriptstyle\mathcal{LV}$}} E \mid A \mathbin{\textcircled{$\scriptstyle\mathcal{LV}$}} E$$

$$C \in Contexts_{\mathcal{LV}} ::= \square \mid \langle v_{\mathcal{LV}} \| \tilde{\mu}y^{\mathcal{LV}}.C \rangle$$

FIGURE 46. (Co-)values in call-by-name ($\mathcal{N}$), -value ($\mathcal{V}$), and -need ($\mathcal{LV}$)

$$\frac{}{\Gamma, x^{\mathcal{S}} : A_{\mathcal{S}} \vdash_{\Theta} x^{\mathcal{S}} : A_{\mathcal{S}} \mid \Delta}\ Var \qquad \frac{}{\Gamma \mid \alpha^{\mathcal{S}} : A_{\mathcal{S}} \vdash_{\Theta} \alpha^{\mathcal{S}} : A_{\mathcal{S}}, \Delta}\ Co\text{-}Var$$

$$\frac{c : (\Gamma \vdash_{\Theta} \alpha^{\mathcal{S}} : A_{\mathcal{S}}, \Delta)}{\Gamma \vdash_{\Theta} \mu\alpha^{\mathcal{S}}.c : A_{\mathcal{S}} \mid \Delta}\ Act \qquad \frac{c : (\Gamma, x^{\mathcal{S}} : A_{\mathcal{S}} \vdash_{\Theta} \Delta)}{\Gamma \mid \tilde{\mu}x^{\mathcal{S}}.c : A_{\mathcal{S}} \vdash_{\Theta} \Delta}\ Co\text{-}Act$$

$$\frac{\Gamma \vdash_{\Theta} v_{\mathcal{S}} : A_{\mathcal{S}} \mid \Delta \quad FV(A_{\mathcal{S}}) \subseteq \Theta \quad \Gamma \mid e_{\mathcal{S}} : A_{\mathcal{S}} \vdash_{\Theta} \Delta}{\langle v_{\mathcal{S}} \| e_{\mathcal{S}} \rangle : (\Gamma \vdash_{\Theta} \Delta)}\ Cut$$

$$\frac{\Gamma \vdash_{\Theta} v_{\mathcal{S}_1} : A_{\mathcal{S}_1} \mid \Delta \quad \Gamma \mid e_{\mathcal{S}_2} : B_{\mathcal{S}_2} \vdash_{\Theta} \Delta}{\Gamma \mid v_{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{S}$}} e_{\mathcal{S}_2} : A_{\mathcal{S}_1} \to^{\mathcal{S}} B_{\mathcal{S}_2} \vdash_{\Theta} \Delta}\ {\to}L \qquad \frac{c : (\Gamma, x^{\mathcal{S}_1} : A_{\mathcal{S}_1} \vdash_{\Theta} \alpha^{\mathcal{S}_2} : B_{\mathcal{S}_2}, \Delta)}{\Gamma \vdash_{\Theta} \mu[x^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{S}$}} \alpha^{\mathcal{S}_2}.c] : A_{\mathcal{S}_1} \to^{\mathcal{S}} B_{\mathcal{S}_2} \mid \Delta}\ {\to}R$$

$$\frac{\Gamma \mid e_{\mathcal{S}_2} : B_{\mathcal{S}_2}\{A_{\mathcal{S}_1}/t^{\mathcal{S}_1}\} \vdash_{\Theta} \Delta \quad FV(A_{\mathcal{S}_1}) \subseteq \Theta}{\Gamma \mid A^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{S}$}} e_{\mathcal{S}_2} : \forall^{\mathcal{S}} t^{\mathcal{S}_1}.B_{\mathcal{S}_2} \vdash_{\Theta} \Delta}\ \forall L \qquad \frac{c : (\Gamma, \vdash_{\Theta, t^{\mathcal{S}_1}} \alpha^{\mathcal{S}_2} : B_{\mathcal{S}_2}, \Delta)}{\Gamma \vdash_{\Theta} \mu[t^{\mathcal{S}_1} \mathbin{\textcircled{$\scriptstyle\mathcal{S}$}} \alpha^{\mathcal{S}_2}.c] : \forall^{\mathcal{S}} t^{\mathcal{S}_1}.B_{\mathcal{S}_2} \mid \Delta}\ \forall R$$

FIGURE 47. Type system for classical system F with multiple strategies ($\mu\tilde{\mu}_{\mathcal{S}\star}^{\to\forall}$)

can focus on a small core calculus that has enough to reveal crucial complexities, but not enough for us to be caught up in uninteresting details.

CHAPTER VI

STRONG NORMALIZATION

The model in this chapter is a ultimately derived from an appendix appearing in (Downen et al. (2015)) which was my main contribution to that paper I coauthored with Paul Downen and Zena M. Ariola. I initially came up with the idea for the fixed-point construction used in this chapter independently, later realizing that it could be seen as a refinement of the technique of (Barbanera and Berardi (1994)). Paul Downen suggested using variations on orthogonality combined with restriction operators which I developed here into the theory of generalized orthogonality, as well as the notion of "charge" in the behavioral characterization of head reduction.

Now that we have a type language, what can we say about it? We saw in Chapter V that the approach to confluence with η from Chapter III doesn't scale to other types and other strategies. Indeed, our typed reduction theory does not fully capture η, *and* admits non-confluent strategies.

But there are other properties of interest. One is "strong normalization" which is the property that every reduction path is finite, meaning that there are no infinite loops, even allowing reduction inside of a program. Strong normalization is closely related to confluence: in a strongly normalizing system it is enough to prove only "local confluence", that any single-step critical pair eventually comes back together. Assuming strong normalization local confluence implies confluence, while that is not true in non-strongly-normalizing settings. Moreover, strong normalization is of importance directly for compiler writers, as if we have a strongly normalizing reduction system the compiler can freely apply rewrites without getting into infinite loops. Thus, even when we have some non-terminating reductions, it is often useful to know that the remainder of the reduction system is strongly normalizing.

Untyped programming languages are rarely strongly normalizing. However, the situation is different with types where it may be possible to have large strongly normalizing fragments where interesting programs can be written.

Consequently there are well developed proof techniques for proving strong normalization for certain classes of typed languages. In pure typed languages, strong normalization is often shown with the reducibility candidates technique (Girard, Taylor, and Lafont (1989b)) which models types as sets of terms. This picture fails once effects are added: with effects different evaluation strategies can result in different answers for the same program. Consequently, different strategies also require different equational and reduction theories specifying the rules for what program transformations are even *possible*. In the λ-calculus, for example, β reduction is sufficient for deciding equality only with the call-by-name evaluation strategy. For call-by-value evaluation, other rules are needed to make a complete equational theory (Herbelin and Zimmermann (2007); Sabry and Felleisen (1992)), which must be included in any reduction theory that hopes to decide equality of terms.

In this vein, while there are many published proofs of strong normalization for call-by-name and call-by-value λ-calculi, we know of no existing proofs for the call-by-need λ-calculus. The proofs for call-by-value and call-by-name are insufficient since neither strategy fully captures call-by-need's reduction theory. Further, call-by-need is just one of many possible alternative strategies. How can a language designer be sure their newly invented evaluation strategy is strongly normalizing?

The main component of this chapter is a proof of strong normalization which is strategy *parametric*, working uniformly for any strategy satisfying some minimal conditions. Our focus is on explaining the proof. Thus, we initially use the single strategy sequent calculus $\mu\tilde{\mu}_{\mathcal{S}}^{\rightarrow\forall}$ whose only type formers are implication and universal quantification. Later in the chapter we discuss how the proof can be amended to handle larger systems with other type formers or those combining multiple strategies (such as $\mu\tilde{\mu}_{\mathcal{S}^\star}^{\rightarrow\forall}$). As with the rest of this dissertation, our setting is the sequent calculus. However, our technique can be applied directly to the λ-calculus as well.

An early version of the technique in this chapter was already put to use to show strong normalization for a language with well founded (co-)induction in Downen et al. (2015). We do not consider recursive types here, but do consider features, such as mixed strategies, not considered in that work. Moreover, compared to that earlier paper, the proof in this chapter is significantly refined. As a result of that refinement, we are now able to establish some additional theorems showing the relation of our technique to other methods.

Strong normalization proofs for languages similar to ours have a rich heritage tracing back decades. Orthogonality provides elegant proofs for call-by-name and -value languages, and gives direct constructions for modeling types (Krivine (2009)). Orthogonality has been further refined for polarized languages, exposing how types are generated by their values (Munch-Maccagnoni (2009)). On the other hand, the symmetric candidates technique (Barbanera and Berardi (1994)) works for non-confluent strategies where orthogonality is not applicable (Lengrand and Miquel (2008)) by interpreting types with an infinite fixed-point construction. Our proof does not overthrow these previous methods, but is instead a capstone combining ideas from each and subsuming them all. In our view, the key insight of all of these techniques is that types are intrinsically two-sided, accounting for how elements of a type are both constructed and used. The meaning of a type must track both its valid terms and valid contexts, and "candidates" of types are defined by how these two sets relate to each other. By studying the algebra of "pre-types," we hope to demystify that essential relationship.

Admissible Strategies

Our proof of strong normalization is parametric in the strategy $\mathcal{S}$. Moreover, in the case of the multi-strategy system, our proof is parametric over the entire collection of strategies. However, there are two important properties for each $\mathcal{S}$ which we need in order for our proof to hold.

Definition 4. *First, a strategy is* stable *if and only if:*

1. *(co-)values are closed under reduction and substitution, and*
2. *non-(co-)values are closed under substitution and ς reduction.*

Second, a strategy is focalizing *if and only if all:*

1. *(co-)variables,*
2. *structures built from (co-)values ($V \cdot E$ and $A\#E$), and*
3. *case abstractions*

are considered (co-)values.

The focalizing property of strategies corresponds to focalization in logic (Munch-Maccagnoni (2009))—each criterion for a focalizing strategy comes from an inference rule for typing a (co-)value in focus.

In general, for the multi-strategy system we need to account for all of the strategies when we prove stability and focalization as the notion of co-value in the other strategies may influence the notion of co-value in any given strategy.

Proposition 2. *The call-by-value ($\mathcal{V}$), call-by-name ($\mathcal{N}$), call-by-need ($\mathcal{LV}$), and non-deterministic strategy ($\mathcal{U}$) are collectively stable and focalizing.*

Proof. For all the strategies, focalization is immediate. And, stability is trivial for all but call-by-name. To show stability for call-by-name, observe that the decomposition of a command c into $C[\langle x^{\mathcal{LV}} \| E_{\mathcal{LV}}\rangle]$ is, if it exists, unique so long as $x^{\mathcal{LV}}$ is not bound in c (by induction on C in the solution). Moreover, both (co-)values and non-(co-)values are closed under substitution (immediate). Moreover, $\mathcal{LV}$ contexts are closed under substitution of (co-)variables by (co-)values.

Now, observe that the lifting rules do not change values into non values and vice versa.

– $v_{\mathcal{S}} \textcircled{$\scriptscriptstyle\mathcal{LV}$}\, e \to_{\varsigma\to} \tilde{\mu} x^{\mathcal{LV}}.\langle v_{\mathcal{S}} \| \tilde{\mu} y^{\mathcal{S}}.\langle x^{\mathcal{LV}} \| y^{\mathcal{S}} \textcircled{$\scriptscriptstyle\mathcal{LV}$}\, e\rangle\rangle$ if e is not a co-value, then neither side are (co-)values. If e is a co-value then both sides are.

– $V \textcircled{$\scriptscriptstyle\mathcal{LV}$}\, e_{\mathcal{S}} \to_{\varsigma\to} \tilde{\mu} x^{\mathcal{LV}}.\langle \mu\beta^{\mathcal{S}}.\langle x^{\mathcal{LV}} \| V \textcircled{$\scriptscriptstyle\mathcal{LV}$}\, \beta^{\mathcal{S}}\rangle \| e_{\mathcal{S}}\rangle$ for this rule to fire $e_{\mathcal{S}}$ must not be a co-value, in which case neither side are (co-)values. Notably, in the right hand side the reason it is not a co-value is that $\langle \mu\beta^{\mathcal{S}}.C \| e_{\mathcal{S}}\rangle$ is not a production of C.

– $A \textcircled{$\scriptscriptstyle\mathcal{LV}$}\, e_{\mathcal{S}} \to_{\varsigma\forall} \tilde{\mu} x^{\mathcal{LV}}.\langle \mu\beta^{\mathcal{S}}.\langle x^{\mathcal{LV}} \| A \textcircled{$\scriptscriptstyle\mathcal{LV}$}\, \beta^{\mathcal{S}}\rangle \| e_{\mathcal{S}}\rangle$, as before, $e_{\mathcal{S}}$ must not be a co-value. So neither side are (co-)values.

All that remains is showing that the additional reduction rules do not turn (co-)values into non-(co-)values. To do so, all at once we show three things:

1. if $E \to e'$ then e' is a co-value,
2. if $V \to v'$ then v' is a value and
3. if $C_{\mathcal{LV}}[\langle x^{\mathcal{LV}} \| E_{\mathcal{LV}} \rangle] \to c$ then there exists $C'_{\mathcal{LV}}, E'_{\mathcal{LV}}$ such that $c = C'_{\mathcal{LV}}[\langle x^{\mathcal{LV}} \| E'_{\mathcal{LV}} \rangle]$.

which follows by mutual induction on the size of the syntax. The interesting case being

$$\begin{aligned}\langle V \| \tilde{\mu} y^{\mathcal{LV}}.C[\langle x^{\mathcal{LV}} \| E \rangle] \rangle &\to C[\langle x^{\mathcal{LV}} \| E \rangle]\{y^{\mathcal{LV}}/V\} \\ &= C\{y^{\mathcal{LV}}/V\}[\langle x^{\mathcal{LV}} \| E\{y^{\mathcal{LV}}/V\} \rangle]\end{aligned}$$

which follows since, by the structure of the syntax $x^{\mathcal{LV}} \neq y^{\mathcal{LV}}$. Note also that because $x^{\mathcal{LV}} \neq y^{\mathcal{LV}}$ we know $\tilde{\mu} y^{\mathcal{LV}}.C[\langle x^{\mathcal{LV}} \| E \rangle]$ is not a co-value, as such, there is no μ reduction case for reducing $C[\langle x^{\mathcal{LV}} \| E \rangle]$. $\square$

Our proof of strong normalization works uniformly for any stable and focalizing collection of strategies. It could be any combination of the four strategies mentioned or any other collection of stable and focalizing strategies.

A Strategy Parametric Proof

Here we will show that reduction is strongly normalizing for *any* stable and focalizing strategy $\mathcal{S}$. Our argument is based on a logical relation: types and typing judgments are interpreted semantically as unary relations on untyped syntax. This follows the usual idea of factoring normalization through logical relations so that (1) the typing rules are sound (all well-typed expressions are in the relation), and (2) everything in the relation is strongly normalizing. More specifically, our method is a combination and refinement of techniques based on *orthogonality* (Krivine (2009); Munch-Maccagnoni (2009)) and *symmetric candidates* (Barbanera and Berardi (1994); Lengrand and Miquel (2008)).

The first challenge is to find a suitable domain for modeling types which can accommodate the control effects in the language. Sequent calculus emphasizes a fact already latent in λ-calculus: types classify both the production (terms) and consumption (co-terms) of information. As such, we take a two-sided view of types modeled as a *pair* of a set of terms and a set of co-terms, both strongly normalizing, as opposed to the more common orthogonality-based models that only pay attention to one side (i.e. terms) at a time. This domain of term-and-co-term pairs, which we call *pre-types* since only some behave as types, becomes a central object of study.

(Co-)terms aren't the only syntactic objects, however. We must also ensure that the commands, formed by the cut rule, are also strongly normalizing. Thus, the set of strongly normalizing commands, denoted by $\perp\!\!\!\perp$, becomes the other central object of study. The set $\perp\!\!\!\perp$ lets us speak about the namesake *orthogonality* operation on pre-types, and check when a pre-type is *orthogonally sound* so that all the commands formed by its (co-)terms are strongly normalizing. Orthogonal soundness is a crucial property that must hold for all *reducibility candidates*—that is, pre-types that might actually model types—guaranteeing that the type disallows any bad interactions and justifying the cut rule for that type.

Soundness is only one of the criteria for reducibility candidates. We must also force them to be "complete" in the appropriate sense so that they carry all the (co-)terms promised by the typing rules. For instance, for the (*Co-*)*Act* rules for forming μ and $\tilde{\mu}$ abstractions to be sound, we must ensure that the interpretation of every type contains at least those abstractions meeting the premise of the typing rules. And thus we meet our second challenge. Usually, orthogonality-based models will rely on a *closure under head expansion* property of the set $\perp\!\!\!\perp$—that $\langle v \| e \rangle \in \perp\!\!\!\perp$ if $\langle v \| e \rangle \mapsto c$ and c, v, e are strongly normalizing—to ensure the necessary completeness. Unfortunately, this property fails in the general setting, because non-confluent strategies like $\mathcal{U}$ allow for the essential critical pair of classical logic, and it can happen that $c_1 \mapsfrom \langle v \| e \rangle \mapsto c_2$ where v, e, c_1 is strongly normalizing but c_2 is not (Lengrand and Miquel (2008)).

To sidestep the critical pair, we give up on closure under head expansion for the set $\perp\!\!\!\perp$, and instead construct pre-types that come with an appropriate head expansion closure property in themselves. Our approach derives from the symmetric candidates method, constructing the models for types as fixed-point solutions of an appropriate *saturation* operation on pre-types. And here our two-sided presentation of pre-types bears fruit, because having both sides yields two natural orderings.

One ordering of pre-types is by straightforward inclusion, which we call *refinement*, and uses subset ordering of their (co-)terms in the *same* direction. This ordering is useful in the model for framing soundness and completeness properties of pre-types by putting upper and lower bounds on their permitted and mandatory (co-)terms. Refinement effectively corresponds to the usual subset order used in orthogonality models, and shares the same basic properties like the fact that orthogonality (and other similar operations like saturation) *reverses* refinement ordering. The other ordering of pre-types, which we call *sub-typing* because it corresponds to behavioral sub-typing, does not commonly appear in the literature on orthogonality and uses subset ordering of their (co-)terms in *opposite* directions.

While we do not include sub-typing in our language, it is still useful in the model because of the way orthogonality-like operations *preserve* sub-type ordering. The fact that the domain of pre-types *can* speak about sub-typing guarantees that we have fixed-point solutions to our saturation operation for constructing models of types.

With the two orderings on pre-types, we have a general method for constructing models for types that are fixed-points of *any* orthogonality-like saturation operation, thereby baking in the necessary completeness properties. The final challenge is then to show that these fixed-points are indeed sound. In order to frame saturation in a generic and strategy-agnostic way, we define a special head reduction relation that details the reductions can happen at the top of a program according to a *charge*, and allows us to characterize (co-)terms by how they behave, rather than how they are written, which was the approach of previous symmetric candidates proofs. *Positive* and *negative* head reductions allow the term and co-term, respectively, to take control without consideration for the other side, whereas *neutral* head reductions operate on both sides of a command in tandem. Each of the three charged reduction sub-relations are always deterministic, regardless of the chosen strategy, which cuts the Gordian knot of the classical critical pair. The pre-types we get from one step of saturation may not be sound, but it happens that the fixed-point solutions to saturation *are* sound only because both sides of the fixed-point must be aware of each other so that no ill-behaved head reductions are possible. Strong normalization then follows by commuting head reduction with other arbitrary reductions.

Pre-Types

Types classify both terms and co-terms, so pre-types are a pair of (1) a set of terms and (2) a set of co-terms. We require the (co-)terms of a pre-type to be strongly normalizing on their own. However, in general we don't require *anything* about how a pre-type's terms and co-terms interact. Developing such requirements is the crux of the notion of "reducibility candidates" which model actual types; however, having the more general notion of pre-type available to us is very useful. Firstly, we are able to define candidates as pre-types satisfying certain conditions, and so being able to state those conditions as conditions on pre-types is very convenient. Secondly, pre-types which *are not* well-behaved types necessarily appear in the interim as we *construct* types, since types will be modeled as fixed-points of functions over the domain of pre-types.

Definition 5. *A pre-type in $\mathcal{S}$ is a pair of a set of strongly normalizing terms from $\mathcal{S}$ and a set of strongly normalizing co-terms in $\mathcal{S}$. For a pre-type $\mathcal{A}$, we write $v \in \mathcal{A}$ or $e \in \mathcal{A}$ to mean that (co-)term is an element of the respective set underlying $\mathcal{A}$.*

Making pre-types explicit in our construction reveals latent structure that might otherwise be missed. As pairs of sets, pre-types have two fundamental orderings arising from the subset relation on their underlying sets. One we call *refinement* which is just the product of the two underlying subset partial orders. The other we call *sub-typing* comes as the product of the subset order for the set of terms with the *opposite* order for the set of co-terms. It is the interplay of these orderings which makes pre-types a productive domain for modeling types.

Refinement makes it easy to talk about when a type *contains* the right (co-)terms since it treats the two sides of the type symmetrically and so works much like the ordinary subset operation. As such, we use $\sqsubseteq$ pervasively in our definitions.

On the other hand, we use the term "sub-type" for the alternative order because it corresponds to behavioral sub-typing (Liskov (1987)): a type $\mathcal{A}$ is a sub-type of $\mathcal{B}$ if every object of $\mathcal{A}$ is an object of $\mathcal{B}$ *and also* the valid observations that can be made about $\mathcal{B}$ are a subset of the observations of $\mathcal{A}$. Although less useful for constraining types, sub-typing is crucial for us because the operations we use for building types *preserve* sub-type order but *reverse* refinement order, and thus the sub-type ordering is used to find fixed-points to those operations.

Definition 6. *Given two pre-types $\mathcal{A}$ and $\mathcal{B}$ in $\mathcal{S}$, $\mathcal{A}$* refines *$\mathcal{B}$, written $\mathcal{A} \sqsubseteq \mathcal{B}$, if and only if*

$$v \in \mathcal{A} \implies v \in \mathcal{B} \qquad \textit{and} \qquad e \in \mathcal{A} \implies e \in \mathcal{B}$$

and $\mathcal{A}$ is a sub-type *of $\mathcal{B}$, written $\mathcal{A} \leq \mathcal{B}$, if and only if*

$$v \in \mathcal{A} \implies v \in \mathcal{B} \qquad \textit{and} \qquad e \in \mathcal{B} \implies e \in \mathcal{A} \; .$$

Notice that both these two orderings form complete lattices on the set of strongly-normalizing pre-types which is crucial to constructing our models of types as fixed-point solutions.

Lemma 27 (Pre-type lattice)**.** *Each of the $\leq$ and $\sqsubseteq$ orderings on the set of pre-types in a strategy $\mathcal{S}$ form a complete lattice: that is, they have all joins and meets.*

Proof. The set of subsets of a set is a complete lattice ordered by $\subseteq$ with the usual $\bigcup$ and $\bigcap$ operations. Further, the dual of a complete lattice is itself a complete lattice, and the product of two complete lattices is a complete lattice. The case of $\sqsubseteq$ is the product of the two subset lattices; the case of $\leq$ is the product of the two subset lattices where one is dualized. $\square$

In particular, given the pre-types $\mathcal{A} = (\mathcal{A}^+, \mathcal{A}^-)$ and $\mathcal{B} = (\mathcal{B}^+, \mathcal{B}^-)$, where $\mathcal{A}^+$ and $\mathcal{B}^+$ contain the terms and $\mathcal{A}^-$ and $\mathcal{B}^-$ contain the co-terms of $\mathcal{A}$ and $\mathcal{B}$, respectively, we have the following binary joins and meets for both orders:

$$
\begin{aligned}
(\mathcal{A}^+, \mathcal{A}^-) \sqcup (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cup \mathcal{B}^+, \mathcal{A}^- \cup \mathcal{B}^-) \\
(\mathcal{A}^+, \mathcal{A}^-) \sqcap (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cap \mathcal{B}^+, \mathcal{A}^- \cap \mathcal{B}^-) \\
(\mathcal{A}^+, \mathcal{A}^-) \vee (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cup \mathcal{B}^+, \mathcal{A}^- \cap \mathcal{B}^-) \\
(\mathcal{A}^+, \mathcal{A}^-) \wedge (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cap \mathcal{B}^+, \mathcal{A}^- \cup \mathcal{B}^-)
\end{aligned}
$$

Note that we use square symbols like $\sqsubseteq$, $\sqcap$, and $\sqcup$ for refinement and triangular ones like $\leq$, $\wedge$, and $\vee$ for sub-typing.

Lemma 28. *$\sqcup$ is monotonic (in both arguments) with respect to $\leq$ and is commutative and associative.*

Proof. That $\sqcup$ is commutative and associative follows immediately from its definition. The interesting fact is that, for any pre-types $\mathcal{A} = (\mathcal{A}^+, \mathcal{A}^-)$, $\mathcal{B} = (\mathcal{B}^+, \mathcal{B}^-)$, and $\mathcal{C} = (\mathcal{C}^+, \mathcal{C}^-)$ if $\mathcal{A} \leq \mathcal{B}$ then we have

$$
\begin{aligned}
\mathcal{A} \sqcup \mathcal{C} &= (\mathcal{A}^+ \cup \mathcal{C}^+, \mathcal{A}^- \cup \mathcal{C}^-) \\
&\leq (\mathcal{B}^+ \cup \mathcal{C}^+, \mathcal{B}^+ \cup \mathcal{C}^-) \\
&= \mathcal{B} \sqcup \mathcal{C}
\end{aligned}
$$

since $\mathcal{A}^+ \subseteq \mathcal{B}^+$ means $\mathcal{A}^+ \cup \mathcal{C}^+ \subseteq \mathcal{B}^+ \cup \mathcal{C}^+$ and $\mathcal{B}^- \subseteq \mathcal{A}^-$ means $\mathcal{B}^- \cup \mathcal{C}^- \subseteq \mathcal{A}^- \cup \mathcal{C}^-$. $\square$

We can now talk about the largest pre-type SN with respect to refinement ordering, which is just the pre-type containing all strongly normalizing (co-)terms, as well as the complementary set $\perp\!\!\!\perp$ of strongly normalizing commands.

Definition 7. *1. $SN_{\mathcal{S}}$ is the largest pre-type (w.r.t. $\sqsubseteq$) containing exactly the strongly normalizing (co-)terms in $\mathcal{S}$.*

2. $\perp\!\!\!\perp$ is the set containing exactly the strongly normalizing commands.

Note also that $\perp\!\!\!\perp$ and $SN_{\mathcal{S}}$ are closed under reduction since all reducts of strongly normalizing expressions are themselves strongly normalizing.

The most basic operation on pre-types is the orthogonal $(-)^{\perp}$ which collects all the (co-)terms which are strongly normalizing with everything in that pre-type.

Definition 8. *For any pre-type $\mathcal{A}$, the pre-type $\mathcal{A}^{\perp}$ is:*

$$v \in \mathcal{A}^{\perp} \iff v \in SN \land \forall e \in \mathcal{A}.\langle v \| e \rangle \in \perp\!\!\!\perp$$
$$e \in \mathcal{A}^{\perp} \iff e \in SN \land \forall v \in \mathcal{A}.\langle v \| e \rangle \in \perp\!\!\!\perp .$$

Furthermore, $\mathcal{A}$ is orthogonally sound *if it refines its own orthogonal ($\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$) and* orthogonally complete *if it is refined by its own orthogonal ($\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}$).*

Expanding the definition, $\mathcal{A}$ is orthogonally sound if for all $v, e \in \mathcal{A}$, $\langle v \| e \rangle \in \perp\!\!\!\perp$. It is orthogonally complete if *every* $v \in SN$ such that $\langle v \| e \rangle \in \perp\!\!\!\perp$ for all $e \in \mathcal{A}$ is also in $\mathcal{A}$, and the dual statement for co-terms. However, we prefer the refinement based definitions to cut down on nested quantifiers. Because refinement is a partial order, an orthogonally sound and complete pre-type $\mathcal{A}$ is one where $\mathcal{A} = \mathcal{A}^{\perp}$.

In terms of sub-typing, the orthogonal is a monotonic function. Given $\mathcal{A} \leq \mathcal{B}$, if $v \in \mathcal{A}^{\perp}$ then $v \in \mathcal{B}^{\perp}$ as well because the co-terms of $\mathcal{B}$ are included in $\mathcal{A}$, and vice versa. Since in most proofs the orders on pre-types are glossed over, the monotonicity with respect to sub-typing is not immediately noted. However, for us it is central. On the other hand, viewed from the perspective of refinement, the orthogonal operation is order *inverting* and is self adjoint. We summarize the relationship between orthogonality and the two orderings as follows, where all but the first result are standard.

Lemma 29. *For all pre-types $\mathcal{A}$ and $\mathcal{B}$, the following hold:*

– *Monotonicity: if $\mathcal{A} \leq \mathcal{B}$ then $\mathcal{A}^{\perp} \leq \mathcal{B}^{\perp}$*

– *Contrapositive: if $\mathcal{A} \sqsubseteq \mathcal{B}$ then $\mathcal{B}^{\perp} \sqsubseteq \mathcal{A}^{\perp}$*

– *Double negation introduction: $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp\perp}$*

– *Triple negation elimination: $\mathcal{A}^{\perp} = \mathcal{A}^{\perp\perp\perp}$*

Proof. If the set of terms of $\mathcal{A}$ is a subset of the set of terms of $\mathcal{B}$ then any co-term which is strongly normalizing when paired with all the terms in $\mathcal{B}$ must be strongly normalizing when paired with the smaller set of terms in $\mathcal{A}$. Thus, in that case, the set of co-terms of $\mathcal{B}^{\perp}$ must be a subset of the co-terms of $\mathcal{A}^{\perp}$. Conversely, if the co-terms of $\mathcal{A}$ is a subset of the co-terms of $\mathcal{B}$ then the terms of $\mathcal{B}^{\perp}$ are a subset of the terms of $\mathcal{A}^{\perp}$. Thus, by the definition of $\leq$, $(-)^{\perp}$ must be monotonic with respect to it, and by the definition of $\sqsubseteq$ it must be order inverting.

If v is in $\mathcal{A}$ then by the definition of $(-)^{\perp}$ we know that for every $e \in \mathcal{A}^{\perp}, \langle v \| e \rangle \in \perp\!\!\!\perp$. Therefore, it must be that $v \in \mathcal{A}^{\perp\perp}$. As the case of co-terms is symmetric.

By the above we know that $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp\perp}$. Thus, by contrapositive $\mathcal{A}^{\perp\perp\perp} \sqsubseteq \mathcal{A}^{\perp}$. However, the double negation elimination also gives us the inclusion in the other direction, $\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}^{\perp\perp\perp}$ and so they must be equal. □

A (co-)term is strongly normalizing iff it is strongly normalizing when made into a command with a (co-)variable. While seemingly obvious, this is an important sanity check since some potential reductions could invalidate that property by allowing reductions only on commands which should really be thought of as reductions internal to the (co-)term (like ς). Because our rewriting theory passes that test, the largest pre-type SN can be rephrased in terms of orthogonality and the pre-type Var of (co-)variables. This ensures that any orthogonally complete pre-type $\mathcal{A}$ must contain (co-)variables, since $\mathcal{A} \sqsubseteq SN$ by definition and thus $SN^{\perp} \sqsubseteq \mathcal{A}^{\perp} \sqsubseteq \mathcal{A}$ by contrapositive. This will be important, because deriving strong normalization of open programs from soundness requires that the models of all types be inhabited with some (co-)values.

Lemma 30. *1. $v_{\mathcal{S}}$ is strongly normalizing iff $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$ is.*

2. $e_{\mathcal{S}}$ is strongly normalizing iff $\langle x_{\mathcal{S}} \| e^{\mathcal{S}} \rangle$ is.

Proof. 1. Since $v_{\mathcal{S}}$ is a sub-term of $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$, strong normalization of $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$ implies strong normalization of $v_{\mathcal{S}}$.

Going the other way, we show that every reduction of $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$, except for possibly one top-level μ reduction, can be traced by $v_{\mathcal{S}}$ as well. We proceed to show that $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$ is strongly normalizing because all of its reducts are by well-founded induction on $|v_{\mathcal{S}}|$:

– Suppose $v_{\mathcal{S}} = \mu\beta^{\mathcal{S}}.c$, so that we have the top-level $\mu_{\mathcal{S}}$ reduction:

$$\langle \mu\beta^{\mathcal{S}}.c \| \alpha^{\mathcal{S}} \rangle \to_{\mu} c\{\alpha^{\mathcal{S}}/\beta^{\mathcal{S}}\}$$

Furthermore, we know $\mu\alpha^{\mathcal{S}}.c\{\alpha^{\mathcal{S}}/\beta^{\mathcal{S}}\}$ is strongly normalizing since it is $\alpha^{\mathcal{S}}$-equivalent to the strongly normalizing $\mu\beta^{\mathcal{S}}.c$, which means that $c\{\alpha^{\mathcal{S}}/\beta^{\mathcal{S}}\}$ is also strongly normalizing since it is a sub-command of $\mu\alpha^{\mathcal{S}}.c\{\alpha^{\mathcal{S}}/\beta^{\mathcal{S}}\}$.

– Suppose we have some other reduction internal to $v_{\mathcal{S}}$, so that:

$$\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle \to \langle v'_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$$

Then we know that $v_{\mathcal{S}} \to v'_{\mathcal{S}}$ so $|v'_{\mathcal{S}}| < |v_{\mathcal{S}}|$. Therefore, by the inductive hypothesis, we get that $\langle v'_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$ is strongly normalizing.

Since every reduct of $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$ is strongly normalizing, then $\langle v_{\mathcal{S}} \| \alpha^{\mathcal{S}} \rangle$ is also strongly normalizing.

2. Analogous to the above by duality. □

Lemma 31. $SN = Var^{\perp}$.

Proof. Note that $Var^{\perp}_{\mathcal{S}}$ is the pre-type:

$$v \in Var^{\perp}_{\mathcal{S}} \iff v \in SN_{\mathcal{S}} \land \forall \alpha^{\mathcal{S}} \in Var_{\mathcal{S}}. \langle v \| \alpha^{\mathcal{S}} \rangle \in \perp\!\!\!\perp$$

$$e \in Var^{\perp}_{\mathcal{S}} \iff e \in SN_{\mathcal{S}} \land \forall x^{\mathcal{S}} \in Var_{\mathcal{S}}. \langle x \| e_{\mathcal{S}} \rangle \in \perp\!\!\!\perp$$

And so $v, e \in SN_{\mathcal{S}}$ if and only if $v, e \in Var_{\mathcal{S}}^{\perp}$ by the above Lemma 30. $\square$

Corollary 1. *If $\mathcal{A}^{\perp} \sqsubseteq \mathcal{A} \sqsubseteq SN$ then $Var \sqsubseteq \mathcal{A}$.*

Proof. Using the above Lemma 31, we conclude by double negation introduction (Lemma 29) and contrapositive (Lemma 29):

$$Var_{\mathcal{S}} \sqsubseteq Var_{\mathcal{S}}^{\perp\perp} = SN_{\mathcal{S}}^{\perp} \sqsubseteq \mathcal{A}^{\perp} \sqsubseteq A \qquad \square$$

Beyond orthogonality, there are other interesting operations on pre-types. We write $\mathcal{A}^{v}$ for the pre-type containing exactly the (co-)values in $\mathcal{A}$. It is immediate that $\mathcal{A}^{v} \sqsubseteq \mathcal{A}$ and that $(-)^{v}$ is monotonic with respect to both orders. Extending this further, it is often fruitful to consider the orthogonal of just the values of a pre-type $(\mathcal{A}^{v})^{\perp}$, which we usually write $\mathcal{A}^{v\perp}$, and which behaves similarly to the plain orthogonal.

Lemma 32. *For all pre-types $\mathcal{A}$ and $\mathcal{B}$ the following hold:*

1. *if $\mathcal{A} \leq \mathcal{B}$ then $\mathcal{A}^{v\perp} \leq \mathcal{A}^{v\perp}$,*
2. *if $\mathcal{A} \sqsubseteq \mathcal{B}$ then $\mathcal{B}^{v\perp} \sqsubseteq \mathcal{A}^{v\perp}$,*
3. *$\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}^{v\perp}$, and*
4. *$\mathcal{A}^{v} \sqsubseteq \mathcal{A}^{v\perp v\perp}$.*

Proof. Properties 1 through 3 hold by composition of a property of $(-)^{v}$ (resp. monotonicity with respect to $\leq$, monotonicity with respect to $\sqsubseteq$, and that $(-)^{v}$ is decreasing) with a property of $(-)^{\perp}$ (monotonicity for 1 and contrapositive for 2 and 3). For property 4 observe that $\mathcal{A}^{v\perp v} \sqsubseteq \mathcal{A}^{v\perp}$ by monotonicity of $(-)^{v}$ and so $\mathcal{A}^{v\perp\perp} \sqsubseteq \mathcal{A}^{v\perp v}$, however, we already know that $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp\perp}$ so by transitivity the property holds. $\square$

Head Reduction

A command is strongly normalizing iff everything it reduces to is. However, *proving* that a command is strongly normalizing based on that fact can be quite challenging. The problem is that it is generally very hard to consider the set of everything that a command might reduce to, as that requires thinking about multiple possible reductions at the top of a command, as well as reductions deep inside a command. To avoid this, we consider a limited rewriting theory which only talks about what reductions can happen at the *head* of a command. Head reduction commutes with internal reduction, giving a limited standardization result—just enough standardization to show orthogonal soundness for forward closed pre-types by only looking at head reductions.

Moreover, it allows us to prove a command strongly normalizing when all we know is that it head-reduces to *a* strongly normalizing command, so long as that reduction happens to be from a deterministic part of the theory. Digging deeper into the rewriting theory, the head reduction relation given in Fig. 48 is *charged*. Neutrally charged reductions $\mapsto_0$ require cooperation of the term and co-term, like in the β rules. Positively charged reductions $\mapsto_+$ allow the term to take over the command in order to simplify itself. Negatively charged reductions $\mapsto_-$ instead allow the co-term to take over the command. There are several useful facts that are immediately apparent about this definition of head reduction.

1. Every head reduction step is simulated by the general reduction theory: $c \mapsto_{+,0,-} c'$ implies that $c \twoheadrightarrow c'$.

2. Head reduction only occurs when one side of the command is a (co-)value: if $\langle v \| e \rangle \ \mapsto_+$ then e is a co-value, if $\langle v \| e \rangle \ \mapsto_-$ then v is a value, and if

$$\langle \mu\alpha.c \| E\rangle \mapsto_{+} c\{E/\alpha\}$$

$$\begin{aligned}
\langle \mu[x\cdot\alpha.c] \| V\cdot E\rangle &\mapsto_{0} c\{V/x, E/\alpha\}\\
\langle \mu[t\#\alpha.c] \| A\#E\rangle &\mapsto_{0} c\{A/t, E/\alpha\}
\end{aligned}$$

$$\begin{aligned}
\langle V \| \tilde\mu x.c\rangle &\mapsto_{-} c\{V/x\}\\
\langle V \| v\cdot e\rangle &\mapsto_{-} \langle V \| \tilde\mu x.\langle v \| \tilde\mu y.\langle x \| y\cdot e\rangle\rangle\rangle && \textbf{where } v \notin Value\\
\langle V \| V'\cdot e\rangle &\mapsto_{-} \langle V \| \tilde\mu x.\langle \mu\beta.\langle x \| V'\cdot \beta\rangle \| e\rangle\rangle && \textbf{where } e \notin Co\text{-}Value\\
\langle V \| A\#e\rangle &\mapsto_{-} \langle V \| \tilde\mu x.\langle \mu\beta.\langle x \| A\#\beta\rangle \| e\rangle\rangle && \textbf{where } e \notin Co\text{-}Value
\end{aligned}$$

FIGURE 48. Head Reductions.

$\langle v \| e\rangle \mapsto_0$ then both v and e are (co-)values. This last point follows from the assumption that the chosen strategy $\mathcal{S}$ is focalizing.

3. When taken on their own, each of the charged head reduction relations, $\mapsto_0$, $\mapsto_+$, and $\mapsto_-$, are always deterministic regardless of the chosen strategy, even though the combined $\mapsto_{+,-}$ head reduction relation may be non-deterministic. Additionally, the combined $\mapsto_{+,0}$ and $\mapsto_{-,0}$ reduction relations are always deterministic as well.

Furthermore, the key to its utility is the observation that head reduction steps commute with reduction inside the (co-)term, and that commutation preserves charge.

Lemma 33. *1. If $v \to v'$ and $\langle v\|e\rangle \mapsto_0 c$ then there is a c' such that $\langle v'\|e\rangle \mapsto_0 c'$ and $c \twoheadrightarrow c'$.*

2. If $e \to e'$ and $\langle v\|e\rangle \mapsto_0 c$ then there is a c' such that $\langle v\|e'\rangle \mapsto_0 c'$ and $c \twoheadrightarrow c'$.

3. If $v \to v'$ and $\langle v\|e\rangle \mapsto_- c$ then there is a c' such that $\langle v'\|e\rangle \mapsto_- c'$ and $c \twoheadrightarrow c'$.

4. If $e \to e'$ and $\langle v\|e\rangle \mapsto_+ c$ then there is a c' such that $\langle v\|e'\rangle \mapsto_- c$ and $c \twoheadrightarrow c'$.

5. *If* $v \to v'$ *then either*

   (a) $\forall e, c$ *such that* $\langle v \| e \rangle \mapsto_+ c$ *we have* $c \twoheadrightarrow \langle v' \| e \rangle$, *or*

   (b) $\forall e, c$ *such that* $\langle v \| e \rangle \mapsto_+ c$ *there exists a* c' *such that* $\langle v' \| e \rangle \mapsto_+ c'$ *and* $c \twoheadrightarrow c'$.

6. *If* $e \to e'$ *then either*

   (a) $\forall v, c$ *such that* $\langle v \| e \rangle \mapsto_- c$ *we have* $c \twoheadrightarrow \langle v \| e' \rangle$ *or*

   (b) $\forall v, c$ *such that* $\langle v \| e \rangle \mapsto_- c$ *there exists a* c' *such that* $\langle v \| e' \rangle \mapsto_- c'$ *and* $c \twoheadrightarrow c'$.

Proof. By cases on the possible reductions. Note that for statements 1-4, there are no critical pairs between neutral head reductions and general reductions, because (co-)values are closed under reduction by stability. Additionally, for the other statements, the fact that (co-)values are closed under reduction cuts out many critical pairs. For statements 5 and 6 we consider all the remaining interesting cases.

– $\tilde{\mu} x^{\mathcal{S}}.\langle x^{\mathcal{S}} \| e \rangle \to e$. Statement (a) holds since the only possible negative head reduction is

$$\langle V \| \tilde{\mu} x^{\mathcal{S}}.\langle x^{\mathcal{S}} \| e \rangle \rangle \mapsto_- \langle V \| e \rangle$$

  which is identical.

– $\mu \alpha^{\mathcal{S}}.\langle v \| \alpha^{\mathcal{S}} \rangle \to v$ Statement (a) holds analogously to above.

– $\tilde{\mu} x^{\mathcal{S}}.c \to \tilde{\mu} x^{\mathcal{S}}.c'$. Statement (b) holds since the only possible head reduction is

$$\langle V \| \tilde{\mu} x^{\mathcal{S}}.c\rangle \mapsto_{-} c\{V/x^{\mathcal{S}}\}$$

and since reduction commutes with substitution, $c\{V/x^{\mathcal{S}}\} \twoheadrightarrow c'\{V/x^{\mathcal{S}}\}$

– $\mu\alpha^{\mathcal{S}}.c \to \mu\alpha^{\mathcal{S}}.c'$. Statement (b) holds analogously to above.

– $v_{\mathcal{S}_1} \circledS\, e \to \tilde{\mu} x^{\mathcal{S}}.\langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}_1}.\langle x \| y^{\mathcal{S}_1} \circledS\, e\rangle\rangle$. Case (b) holds as there is only a single possible negative head reduction.

$$\begin{aligned}
\langle V \| v_{\mathcal{S}_1} \circledS\, e\rangle &\mapsto_{-} \langle V \| \tilde{\mu} x^{\mathcal{S}}.\langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}}.\langle x^{\mathcal{S}} \| y^{\mathcal{S}_1} \circledS\, e\rangle\rangle\rangle \\
&\to \langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}_1}.\langle V \| y^{\mathcal{S}_1} \circledS\, e\rangle\rangle \\
\langle V \| \tilde{\mu} x^{\mathcal{S}}.\langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}_1}.\langle x^{\mathcal{S}} \| y^{\mathcal{S}_1} \circledS\, e\rangle\rangle\rangle &\mapsto_{-} \langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}_1}.\langle V \| y^{\mathcal{S}_1} \circledS\, e\rangle\rangle
\end{aligned}$$

– In the case of $V \circledS\, e_{\mathcal{S}_2} \to \tilde{\mu} x^{\mathcal{S}}.\langle \mu\alpha^{\mathcal{S}_2}.\langle x^{\mathcal{S}} \| V \circledS\, \alpha^{\mathcal{S}_2}\rangle \| e\rangle$ (b) holds since there is only one possible negative head reduction.

$$\begin{aligned}
\langle V' \| V \circledS\, e_{\mathcal{S}_2}\rangle &\mapsto_{-} \langle V' \| \tilde{\mu} x^{\mathcal{S}}.\langle \mu\alpha^{\mathcal{S}_2}.\langle x^{\mathcal{S}} \| V \circledS\, \alpha^{\mathcal{S}_2}\rangle \| e_{\mathcal{S}_2}\rangle\rangle \\
&\to \langle \mu\alpha^{\mathcal{S}_2}.\langle V' \| V \circledS\, \alpha^{\mathcal{S}_2}\rangle \| e_{\mathcal{S}_2}\rangle \\
\langle V' \| \tilde{\mu} x^{\mathcal{S}}.\langle \mu\alpha^{\mathcal{S}_2}.\langle x^{\mathcal{S}} \| V \circledS\, \alpha^{\mathcal{S}_2}\rangle \| e_{\mathcal{S}_2}\rangle\rangle &\mapsto_{-} \langle \mu\alpha^{\mathcal{S}_2}.\langle V' \| V \circledS\, \alpha^{\mathcal{S}_2}\rangle \| e_{\mathcal{S}_2}\rangle
\end{aligned}$$

– $\varsigma_{\forall}$ analaguous to above.

– In the case of $v_{\mathcal{S}_1} \circledS e \to V \circledS e$ where $v_{\mathcal{S}_1}$ is not a value, case (a) holds since there is only a single possible head reduction

$$\begin{aligned}
\langle V' \| v_{\mathcal{S}_1} \circledS e\rangle &\mapsto_- \langle V' \| \tilde{\mu} x^{\mathcal{S}}.\langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}_1}.\langle x^{\mathcal{S}} \| y^{\mathcal{S}_1} \circledS e\rangle\rangle\rangle \\
&\to \langle v_{\mathcal{S}_1} \| \tilde{\mu} y^{\mathcal{S}_1}.\langle V' \| y^{\mathcal{S}_1} \circledS e\rangle\rangle \\
&\to \langle V \| \tilde{\mu} y^{\mathcal{S}_1}.\langle V' \| y^{\mathcal{S}_1} \circledS e\rangle\rangle \\
&\to \langle V' \| V \circledS e\rangle
\end{aligned}$$

– In the case of $V \circledS e \to V \circledS E$ and $A \circledS e \to A \circledS E$ where e is not a co-value case (a) holds analogously to above.

$\square$

Lemma 34 (Commutation). *Given any v, e, v', e' where $v \twoheadrightarrow v'$ and $e \twoheadrightarrow e'$ and $p \in \{+, -, 0\}$, if $\langle v \| e\rangle \mapsto_p c$ then there exists some c' such that $\langle v' \| e'\rangle \mapsto_p^= c'$ and $c \twoheadrightarrow c'$. In pictures:*

$$\begin{tikzcd}
\langle v \| e\rangle \arrow[r, mapsto, "p"] \arrow[d, twoheadrightarrow] & c \arrow[d, dashed, twoheadrightarrow] & \text{or} & \langle v \| e\rangle \arrow[r, mapsto] \arrow[d, twoheadrightarrow] & c \arrow[dl, dashed, twoheadrightarrow] \\
\langle v' \| e'\rangle \arrow[r, dashed, mapsto, "p"] & c' & & \langle v' \| e'\rangle &
\end{tikzcd}$$

Proof. A special case of Lemma 33 $\square$

The main application of head reduction is in providing a tool for showing that a pre-type is orthogonally sound. Note that the proof of the next lemma is the *only* place where we need to perform induction on strongly normalizing reduction sequences: every other time we need to prove a pre-type orthogonally sound we will be able to use this lemma to avoid having to consider internal reductions.

Lemma 35 (Head orthogonality)**.** *Suppose that $\mathcal{A}$ is a pre-type forward closed under reduction and that for all $v, e \in \mathcal{A}$, $\langle v \| e \rangle \mapsto_{+,0,-} c$ implies $c \in \perp\!\!\!\perp$. Then $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$.*

Proof. Note that a command c is in $\perp\!\!\!\perp$ if and only if all reducts of c are in $\perp\!\!\!\perp$. Since $\mathcal{A} \sqsubseteq SN$, every (co-)term in $\mathcal{A}$ is strongly normalizing, so let $|v|$ and $|e|$ be the lengths of the longest reduction sequence from any $v, e \in \mathcal{A}$, respectively. We now proceed to show that for any $v, e \in \mathcal{A}$, $\langle v \| e \rangle \in \perp\!\!\!\perp$ because all of its reducts are in $\perp\!\!\!\perp$, by induction on $|v| + |e|$.

- $\langle v \| e \rangle \mapsto_{+,0,-} c$: we know $c \in \perp\!\!\!\perp$ by assumption.
- $\langle v \| e \rangle \to \langle v' \| e \rangle$ because $v \to v'$: then $v' \in \mathcal{A}$ because $\mathcal{A}$ is forward closed, and $|v'| < |v|$. Furthermore, we have that all head reducts of $\langle v' \| e \rangle$ are in $\perp\!\!\!\perp$ by Lemma 34 and the fact that $\perp\!\!\!\perp$ is closed under reduction. Therefore, $\langle v' \| e \rangle \in \perp\!\!\!\perp$ by the inductive hypothesis.
- $\langle v \| e \rangle \to \langle v \| e' \rangle$ because $e \to e'$: analogous to the previous case by duality. □

The head orthogonality lemma serves as the workhorse for the rest of our proof. One useful consequence is that if a command made of strongly normalizing (co-)terms can reduce by a neutral head reduction to something strongly normalizing then it must be strongly normalizing as well.

Lemma 36. *If $v, e \in SN$ and $\langle v \| e \rangle \mapsto_0 c \in \perp\!\!\!\perp$, then $\langle v \| e \rangle \in \perp\!\!\!\perp$.*

Proof. Consider the pre-type $\mathcal{A}$ defined as

$$v' \in \mathcal{A} \iff v \twoheadrightarrow v' \qquad\qquad e' \in \mathcal{A} \iff e \twoheadrightarrow e'$$

Given any $v', e' \in \mathcal{A}$, $\langle v' \| e' \rangle \mapsto_0^{=} c'$ such that $c \twoheadrightarrow c'$ by Lemma 34, so that $c' \in \perp\!\!\!\perp$ because $\perp\!\!\!\perp$ is closed under reduction. As such $\langle v' \| e' \rangle$ is either in $\perp\!\!\!\perp$ (meaning everything it head reduces to is in $\perp\!\!\!\perp$) or it undergoes a neutral head reduction to something in $\perp\!\!\!\perp$. However, in second case since both $\mapsto_{0,+}$ and $\mapsto_{0,-}$ are deterministic, that means *everything* it head reduces to is in $\perp\!\!\!\perp$. Thus, $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$ by Lemma 35 and so $\langle v \| e \rangle \in \perp\!\!\!\perp$. □

A second application of the head reduction relation is in classifying terms based on how they behave. For example, we define the set of *simple* (co-)terms which only cause neutral reductions. Simplicity is useful because the simple (co)terms of a type can be defined in a strategy-agnostic way.

Definition 9. *A term is* simple *if it never takes part in a positive head reduction. A co-term is simple if it never takes part in a negative head reduction. A pre-type is simple if all its (co-)terms are.*

Observe that because our chosen strategy is assumed to be focalizing, all simple (co-)terms are (co-)values: a simple (co-)term either takes a neutral head reduction step, in which case it must be stack or a pattern match, or it doesn't actively participate in any reduction, so it must be a (co-)variable. Furthermore, the set of simple (co-)terms is closed under reduction because (co-)values are closed under reduction (since the strategy is stable). Also of note is the fact that non-simple (co-)terms never participate in neutral reductions.

We define the operation $\mathcal{A}^s$ as containing all the simple (co-)terms of $\mathcal{A}$. $(-)^s$ is, by construction, monotonic with respect to both orders. There is also a simplified version of the orthogonality operation $(\mathcal{A}^{\perp})^s$, which we write as $(-)^{\perp s}$. Note that like $(-)^{v\perp}$, $(-)^{\perp s}$ satisfies the same monotonicity and contrapositive properties as the plain orthogonal $(-)^{\perp}$.

Beyond being (co-)values, simple (co-)terms have another important property which is that they are *deterministic*. That is, we can always tell if a command containing a simple (co-)term is strongly normalizing by looking at a *single* possible head reduction it takes. Thus, we also consider a generalization of simplicity in the pre-type of *deterministic values* $\mathcal{DV}$.

$$\begin{aligned}
v \in \mathcal{DV} \iff\ & v \in SN^v \wedge \\
& \forall V', e, c, c', v \twoheadrightarrow V' \Rightarrow \\
& \langle V' \| e \rangle \mapsto c \wedge \langle V' \| e \rangle \mapsto c' \wedge c \in \perp\!\!\!\perp \Rightarrow c' \in \perp\!\!\!\perp \\
e \in \mathcal{DV} \iff\ & e \in SN \twoheadrightarrow e' \wedge \\
& \forall E', v, c, c', e \twoheadrightarrow E' \Rightarrow \\
& \langle v \| E' \rangle \mapsto c \wedge \langle v \| E' \rangle \mapsto c' \wedge c \in \perp\!\!\!\perp \Rightarrow c' \in \perp\!\!\!\perp
\end{aligned}$$

We can restrict any pre-type down to just deterministic values via the operation $(-)^d$ or use a deterministic version of the orthogonal $(-)^{\perp d}$ where

$$\mathcal{A}^{\perp d} = (\mathcal{A}^{\perp})^d = \mathcal{A}^{\perp} \sqcap \mathcal{DV}$$

Observing that

$$\mathcal{A}^d \sqsubseteq \mathcal{A}^s \sqsubseteq \mathcal{A}^v \sqsubseteq \mathcal{A}$$

for all $\mathcal{A}$.

Reducibility Candidates

To define the set of reducibility candidates, we have to determine the pre-types which are sufficiently saturated so that they contain enough (co-)terms to make the typing rules sound without invalidating the *Cut* rule. For example, we need to be sure that reducibility candidates contain the necessary μ- and $\tilde{\mu}$-abstractions according to *Act* and *Co-Act*. Our trick is to characterize this saturation in terms of head reduction, using the $Head(-)$ operation on pre-types given below. $Head(\mathcal{A})$ collects all the terms which induce a positive head reduction to a strongly normalizing command when cut with any of the co-values from $\mathcal{A}$, and all the co-terms which induce a negative head reduction with all the values. As such, to show something is in $Head(\mathcal{A})$, we only need to show it produces *a* strongly normalizing reduction with all the (co-)values in $\mathcal{A}$ instead of showing that *all* its head reductions take us to strongly normalizing places. For example $\mu\alpha.c$ is in $Head(\mathcal{A})$ so long as c is strongly normalizing when we substitute in any co-value from $\mathcal{A}$ for α since $\langle \mu\alpha.c \| E\rangle \mapsto_+ c\{E/\alpha\}$ *even if* E itself is a $\tilde{\mu}$-abstraction which might induce its own head reductions. Formally, $Head(-)$ is defined as:

$$v \in Head(\mathcal{A}) \iff v \in SN \wedge \forall E \in \mathcal{A}. \langle v \| E\rangle \mapsto_+ c \in \perp\!\!\!\perp$$

$$e \in Head(\mathcal{A}) \iff e \in SN \wedge \forall V \in \mathcal{A}. \langle V \| e\rangle \mapsto_- c \in \perp\!\!\!\perp$$

Note that $Head(-)$ satisfies the same monotonicity and contrapositive properties as $(-)^{\perp}$ and $(-)^{\perp s}$.

In order to adequetly model types, pre-types need to contain there own $Head(-)$. However, $Head(-)$ might not always enough since it doesn't for instance, include all the things orthogonal to a type. We thus define the saturation function $Sat(\mathcal{A})$, which selects all the (co-)terms that are strongly normalizing with all the (co-)values of $\mathcal{A}$ either now or one step in the future:

$$\begin{aligned}
v \in Sat(\mathcal{A}) \iff{}& v \in SN \\
&\wedge \forall E \in \mathcal{A}. \langle v \| E \rangle \in \perp\!\!\!\perp \vee \langle v \| E \rangle \mapsto_{+} c \in \perp\!\!\!\perp \\
e \in Sat(\mathcal{A}) \iff{}& e \in SN \\
&\wedge \forall V \in \mathcal{A}. \langle V \| e \rangle \in \perp\!\!\!\perp \vee \langle V \| e \rangle \mapsto_{-} c \in \perp\!\!\!\perp
\end{aligned}$$

$Sat(\mathcal{A})$ includes both $\mathcal{A}$'s head and its (co-)value-orthogonal:

$$Head(\mathcal{A}) \sqcup \mathcal{A}^{v\perp} \sqsubseteq Sat(\mathcal{A}).$$

However, the converse may not be true. The definition of $Sat(\mathcal{A})$ includes all terms which when paired with any co-value of $\mathcal{A}$ produce a command which is either strongly normalizing now or in one step. That could potentially include more terms than are in $Head(\mathcal{A}) \sqcup \mathcal{A}^{v\perp}$ because of the relative order of the quantifier and the disjunction: a term is in $Head(\mathcal{A}) \sqcup \mathcal{A}^{v\perp}$ only when (paired with co-values from $\mathcal{A}$) it either always produces commands that are strongly normalizing immediately *or* always produces commands which are strongly normalizing in one step. The advantage to the larger saturation function is that it produces forward-closed pre-types (by Lemma 34) which might not be true for $Head(\mathcal{A}) \sqcup \mathcal{A}^{v\perp}$ since an internal reduction might eliminate the need to take a head reduction step.

We now define reducibility candidates as all orthogonally sound and complete pre-types that are sufficiently saturated.

Definition 10 (Reducibility candidates)**.** *A pre-type $\mathcal{A}$ is a* reducibility candidate *iff $Sat(\mathcal{A}) \sqsubseteq \mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$.*

The set of reducibility candidates in a strategy $\mathcal{S}$ is written $CR_{\mathcal{S}}$.

An immediate consequence of the definition is that any reducibility candidate contains its own $Head$, is orthogonally complete, and is generated by its values Munch-Maccagnoni (2009).

Theorem 14. *If $\mathcal{A}$ is a reducibility candidate then*

- $\mathcal{A} = \mathcal{A}^{\perp} = \mathcal{A}^{v\perp} = Sat(\mathcal{A})$
- $Head(\mathcal{A}) \sqsubseteq \mathcal{A}$

Proof. For one, we just observe that, by definition, for every pre-type $\mathcal{A}$

$$\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}^{v\perp} \sqsubseteq Sat(\mathcal{A})$$

which means that when $Sat(\mathcal{A}) \sqsubseteq \mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$ all of these must be equal. Two follows since $Head(\mathcal{A}) \sqsubseteq Sat(\mathcal{A})$. □

We know that since it contains its own head, a reducibility candidate must contain all the μ- and $\tilde{\mu}$- abstractions that are well-behaved when paired with any of its (co-)values.

Lemma 37 (Strong activation).

1. *If $\mathcal{A}$ is a reducibility candidate in $\mathcal{S}$ and for all $E \in \mathcal{A}$, $c\{E/\alpha^{\mathcal{S}}\} \in \perp\!\!\!\perp$, then $\mu\alpha^{\mathcal{S}}.c \in \mathcal{A}$.*
2. *If $\mathcal{A}$ is a reducibility candidate and for all $V \in \mathcal{A}$, $c\{V/x^{\mathcal{S}}\} \in \perp\!\!\!\perp$, then $\tilde{\mu}x^{\mathcal{S}}.c \in \mathcal{A}$.*

Proof. – Since $\mathcal{A}$ is a reducibility candidate, we know that $Head(\mathcal{A}) \sqsubseteq \mathcal{A}$. Observe that for all $E \in \mathcal{A}$,

$$\langle \mu\alpha^{\mathcal{S}}.c \| E \rangle \mapsto c\{E/\alpha^{\mathcal{S}}\} \in \perp\!\!\!\perp$$

by assumption. Therefore, $\mu\alpha^{\mathcal{S}}.c \in Head(\mathcal{A}) \sqsubseteq \mathcal{A}$.

– Analogous to the previous statement by duality. □

Additionally, if a reducibility candidate contains all the structures built from (co-)values of other reducibility candidates, then it must contain the general constructs built from (co-)terms as well. This follows from strong activation together with the fact that all the lifting rules are implemented as non-neutrally charged head reductions.

Lemma 38 (Unfocalization). *1. If $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are reducibility candidates such that $V \circledS E \in \mathcal{A}$ for every $V \in \mathcal{B}$ and $E \in \mathcal{C}$, then $v \circledS e \in \mathcal{A}$ for every $v \in \mathcal{B}$ and $e \in \mathcal{C}$.*

2. If $\mathcal{A}$ and $\mathcal{B}$ are reducibility candidates and C is a syntactic type such that $C \circledS E \in \mathcal{A}$ for every $E \in \mathcal{B}$, then $C \circledS e \in \mathcal{A}$ for every $e \in \mathcal{B}$.

Proof. First observe

1. If $\mathcal{A}$ and $\mathcal{B}$ are reducibility candidates and e a co-term such that for all $V \in \mathcal{B}$, $V \circledS e \in \mathcal{A}$ then for every $v \in \mathcal{B}$. $v \circledS e \in \mathcal{A}$.

2. If $\mathcal{A}$ and $\mathcal{B}$ are reducibility candidates and V a value such that for all $E \in \mathcal{B}$, $V \circledS E \in \mathcal{A}$ then for every $e \in \mathcal{B}$, $V \circledS e \in \mathcal{A}$.

since

1. If $\mathcal{A}$ and $\mathcal{B}$ are reducibility candidates and e a co-term such that for all $V \in \mathcal{B}$, $V \circledS e \in \mathcal{A}$ it follows by repeated application of Lemma 37 that for any $v \in \mathcal{B}$, $\tilde{\mu} y^{\mathcal{S}}.\langle v \| \tilde{\mu} x^{\mathcal{S}_1}.\langle y^{\mathcal{S}} \| x^{\mathcal{S}_1} \circledS e \rangle\rangle \in \mathcal{A}$. Thus, for any $v \in \mathcal{B}$ we have for all $V \in \mathcal{A}$ we know $\langle V \| v \circledS e \rangle \mapsto_{-} \langle V \| \tilde{\mu} y^{\mathcal{S}}.\langle v \| \tilde{\mu} x^{\mathcal{S}_1}.\langle y^{\mathcal{S}} \| x^{\mathcal{S}_1} \circledS e \rangle\rangle\rangle \in \perp\!\!\!\perp$ so $v \circledS e \in Head(\mathcal{A})$ meaning $v \circledS e \in \mathcal{A}$.

2. Analogously to statement 1 above.

Therefore we can show our two main goals

1. By statement 1 above, for every $V \in \mathcal{B}$ and $e \in \mathcal{C}$, $V \circledS e \in \mathcal{A}$. Thus, by statement 1 for every $v \in \mathcal{B}$ and $e \in \mathcal{C}$, $v \circledS e \in \mathcal{A}$.

2. Analogous to statements 1 and 2 above. □

Building Candidates

Because $Sat(\mathcal{A})$ includes $\mathcal{A}$'s head and the orthogonal of its (co-)values, we can use $Sat(-)$, to grow a full-fledged reducibility candidate by iteratively saturating it starting with an arbitrary fixed seed $\mathcal{C}$, one *Step* at a time:

$$Step_{\mathcal{C}}(\mathcal{A}) = \mathcal{C} \sqcup Sat(\mathcal{A})$$

Because $Step_{\mathcal{C}}$ is monotonic and the pre-types form a complete lattice with respect to sub-typing, it must have a fixed-point and indeed a least fixed-point Cousot and Cousot (1979). This lets us define a function that saturates any pre-type $\mathcal{C}$.

Lemma 39. $\mathcal{A} \leq \mathcal{B} \implies Step_{\mathcal{C}}(\mathcal{A}) \leq Step_{\mathcal{C}}(\mathcal{B})$

Proof. Because $\sqcup$ is monotonic in both arguments and $Sat(-)$ is monotonic with respect to sub-typing by construction. □

Corollary 2. *For any $\mathcal{C}$, there is a fixed point $\mathcal{A} = Step_{\mathcal{C}}(\mathcal{A})$.*

Lemma 40. *There exists a function on pre-types $\mathcal{R}(-)$ such that $\mathcal{R}(\mathcal{C}) = Step_{\mathcal{C}}(\mathcal{R}(\mathcal{C}))$.*

Proof. By Corollary 2 there is a fixed point to the function $Step_{\mathcal{C}}$. Indeed, there is a least fixed point which we can take for $\mathcal{R}(\mathcal{C})$ avoiding the axiom of choice. □

Note that this $\mathcal{R}(-)$ operation gives us something that *might* be a reducibility candidate. We know it is complete enough by construction; however, perhaps it is not sound. For any $Sat(\mathcal{A}) \sqsubseteq \mathcal{A} \sqsubseteq SN_{\mathcal{S}}$ we have that:

$$Head(\mathcal{A}) \sqsubseteq Sat(\mathcal{A}) \sqsubseteq \mathcal{A}$$
$$\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}^{v\perp} \sqsubseteq Sat(\mathcal{A}) \sqsubseteq \mathcal{A}$$

So all that is remaining is to show that for appropriate choices for $\mathcal{C}$, $\mathcal{R}(\mathcal{C})$ is orthogonally sound, i.e.$\mathcal{R}(\mathcal{C}) \sqsubseteq \mathcal{R}(\mathcal{C})^{\perp}$. In particular, we will find by application of Lemma 35 that $\mathcal{R}(\mathcal{C})$ is guaranteed to be orthogonally sound whenever $\mathcal{C} = \mathcal{C}^{\perp d}$.

First we observe that the fixed-point constructed from a forward closed seed must itself be forward closed because the $Sat(-)$ function always generates a forward closed pre-type.

Lemma 41. *For any pre-type $\mathcal{A}$, $Sat(\mathcal{A})$ is forward closed.*

Proof. – Suppose $v \in Sat(\mathcal{A})$ and $v \to v'$. Let $E \in \mathcal{A}$, so that $v \in Sat(\mathcal{A})$ implies that either $v \perp\!\!\!\perp E$ or $\langle v \| E \rangle \mapsto_{+} c$.

In the first case we know that $\langle v \| E \rangle \in \perp\!\!\!\perp$, so $\langle v \| E \rangle \to \langle v' \| E \rangle$ and $\langle v' \| E \rangle \in \perp\!\!\!\perp$ since $\perp\!\!\!\perp$ is forward closed.

In the second case we know that $\langle v' \| E \rangle \leftarrow \langle v \| E \rangle \mapsto_{+} c \in \perp\!\!\!\perp$. Based on the commutation of internal and head reduction (Lemma 34), and the fact that $\perp\!\!\!\perp$ is forward closed, we have one of two cases:

  * $c \twoheadrightarrow \langle v' \| E \rangle \in \perp\!\!\!\perp$, or
  * $\langle v' \| E \rangle \mapsto_{+} c' \twoheadleftarrow c \in \perp\!\!\!\perp$

Therefore, since in any case $\langle v' \| E \rangle \in \perp\!\!\!\perp$ or $\langle v' \| E \rangle \mapsto_{+} c' \in \perp\!\!\!\perp$, then $v' \in \mathit{Sat}(\mathcal{A})$.

– Suppose $e \in \mathit{Sat}(\mathcal{A})$ and $e \to e'$. Analogous to the previous case by duality.

□

Further, $\mathcal{C}^{\perp s}$ must be forward closed because orthogonal pre-types and the pre-type of deterministic (co-)values are.

Lemma 42. *Given $\mathcal{A} = \mathit{Step}_{\mathcal{C}}(\mathcal{A})$, $\mathcal{A}$ is forward closed when $\mathcal{C}$ is.*

Proof. Immediate. □

Moreover, because we start with a seed pre-type that is deterministic and orthogonally sound, we can analyze all the head reductions between (co-)terms in the fixed-point $\mathcal{A} = \mathit{Step}_{\mathcal{C}}(\mathcal{A}) = \mathit{Sat}(\mathcal{A}) \sqcup \mathcal{C}$ by cases. Since all such head reductions lead to strongly normalizing commands, $\mathcal{A}$ must be orthogonally sound so it is a reducibility candidate.

Lemma 43. *If $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp d}$ and $\mathcal{A} = \mathit{Step}_{\mathcal{C}}(\mathcal{A})$, then for all $v, e \in \mathcal{A}$, $\langle v \| e \rangle \mapsto_{+,0,-} c$ implies $c \in \perp\!\!\!\perp$.*

Proof. Let $v, e \in \mathcal{A} = \mathcal{C} \sqcup \mathit{Sat}(\mathcal{A})$. By cases, we have:

– $V, E \in \mathcal{C}$: $V \perp\!\!\!\perp E$ by the assumption that $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp d}$, so $\mathcal{C}$ is orthogonally sound.

– $V \in \mathcal{C}$, $e \in \mathit{Sat}(\mathcal{A})$: by $e \in \mathit{Sat}(\mathcal{A})$ and the fact that V is a value, we know that either $\langle V \| e \rangle \in\perp\!\!\!\perp$ or $\langle V \| e \rangle \mapsto_{-} c \in\perp\!\!\!\perp$ for some c. In the first case, every reduct of $\langle V \| e \rangle$ is in $\perp\!\!\!\perp$, including any head reductions, since $\perp\!\!\!\perp$ is forward closed. In the second case, we know V is deterministic so $\langle V \| e \rangle \mapsto_{+,-,-} c$ implies c in $\perp\!\!\!\perp$.

– $v \in \mathit{Sat}(\mathcal{A})$, $E \in \mathcal{C}$: analogous to the previous case by duality.

– $v, e \in \mathit{Sat}(\mathcal{A})$: Let us consider the possible head reductions:

  * $\langle v \| e \rangle \mapsto_0 c$: in this case, both v and e are simple (co-)values, and neither can take a positively charged head reduction step. Therefore, for $v, e \in \mathit{Sat}(\mathcal{A})$ to hold, it must be that $\langle v \| e \rangle \in\perp\!\!\!\perp$, and so $c \in\perp\!\!\!\perp$ because $\perp\!\!\!\perp$ is closed under reduction.

  * $\langle v \| e \rangle \mapsto_{+} c$: in this case, e must be a co-value, so for $v \in \mathit{Sat}(\mathcal{A})$ to hold, either $\langle v \| e \rangle \in\perp\!\!\!\perp$ already or there is a c' such that $\langle v \| e \rangle \mapsto_{+} c'$. In the former case, $c \in\perp\!\!\!\perp$ because $\perp\!\!\!\perp$ is closed under reduction, and in the latter case, $c = c' \in\perp\!\!\!\perp$ because $\mapsto_{+}$ is deterministic.

  * $\langle v \| e \rangle \mapsto_{-} c$: analogous to the previous case by duality. □

Lemma 44. *For any forward closed $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp d}$, any $\mathcal{A}$ such that $\mathcal{A} = \mathit{Step}_{\mathcal{C}}(\mathcal{A})$ is a reducibility candidate such that $\mathcal{C} \sqsubseteq \mathcal{A}$.*

As an instance of this fact, if $\mathcal{C} = \mathcal{C}^{\perp d}$ then $\mathcal{R}(\mathcal{C})$ is a reducibility candidate such that $\mathcal{C} \sqsubseteq \mathcal{R}(\mathcal{C})$

Proof. Suppose $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp d}$ and is forward closed, and that $\mathcal{A} = Step_{\mathcal{C}}(\mathcal{A})$. Clearly, $\mathcal{C} \sqsubseteq \mathcal{A}$ by the definition of $Step_{\mathcal{C}}$, so we only need to prove that it is a reducibility candidate. By Lemma 42 and Lemma 43 we know that $\mathcal{A}$ is forward closed and all head reducts are in $\perp\!\!\!\perp$, so $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$ by Lemma 35 and contains $Sat(\mathcal{A})$ by definition.

If $\mathcal{C} = \mathcal{C}^{\perp d}$ then it is forward closed since both the orthogonal and the set of deterministic values and by Lemma 40, $\mathcal{R}(\mathcal{C}) = Step_{\mathcal{C}}(\mathcal{R}(\mathcal{C}))$ and so satisfies the conditions above. □

Now, we just need constructions for forming deterministic seed pre-types corresponding to the type constructors $\to$ and $\forall$. Since both $\to$ and $\forall$ are negative types, it makes sense to define a general strategy for building the core of a negative type from a set of deterministic co-values. The idea is that for any set of deterministic co-values S, $Neg(S)$ will contain all of the deterministic values that are strongly normalizing with anything in S and all the deterministic co-values that are strongly normalizing with *those*. Consequently, starting from a set of deterministic co-values representing core observations of a type we can build up a complete set of deterministic generators for that type.

Definition 11. *For a set of co-values S, the pre-type $Neg(S)$ is given by:*

$$v \in Neg(S) \Leftrightarrow v \in \mathcal{DV} \wedge \forall E \in S, \langle v \| E \rangle \in \perp\!\!\!\perp$$

$$e \in Neg(S) \Leftrightarrow e \in \mathcal{DV} \wedge \forall v \in \mathcal{DV}, (\forall E \in S, \langle v \| E \rangle \in \perp\!\!\!\perp) \Rightarrow \langle v \| e \rangle \in \perp\!\!\!\perp$$

Note: when S is empty, the result strategy of $Neg(S)$ is taken from context.

Lemma 45. *If S is a set of deterministic strongly normalizing co-values in $\mathcal{S}$ then*

1. $\forall E \in S, E \in Neg(S)$ *and*

2. $Neg(S) = Neg(S)^{\perp s}$.

Proof. 1. Since E is assumed to be deterministic and strongly normalizing if it is in S it is in $Neg(S)$ by definition.

2. That $Neg(S) \sqsubseteq Neg(S)^{\perp d}$ follows since $Neg(S)$ is simple by definition and any two elements of $Neg(S)$ must be strongly normalizing when put together by construction. That $Neg(S)^{\perp d} \sqsubseteq Neg(S)$ follows since S is a subset of the values of $Neg(S)$ while the set of co-values already is the simplified orthogonal of the set of values. □

To define a specific negative type like $\to^{\mathcal{S}}$ we then only need to gather up a sufficiently descriptive set of deterministic co-values. For functions, that is a set of call-stacks.

Definition 12. *If $\mathcal{A}$ and $\mathcal{B}$ are pre-types then $FunCore_{\mathcal{S}}(\mathcal{A}, \mathcal{B})$ is the pre-type $Neg(\{V \circledS E \mid V \in \mathcal{A}, E \in \mathcal{B}\})$*

Lemma 46. *If $\mathcal{A}$ and $\mathcal{B}$ are reducibility candidates then $FunCore_{\mathcal{S}}(\mathcal{A}, \mathcal{B}) = FunCore(\mathcal{A}, \mathcal{B})^{\perp d}$.*

Proof. $\{V \circledS E | V \in \mathcal{A}, E \in \mathcal{B}\}$ is a set of deterministic co-values so this follows by Lemma 45. □

However, it is not enough to have a reducibility candidate which contains all the *observations* of a function type. We need it to contain the values as well. This follows since pattern-matching terms are simple values and are included in the seed so long as they are orthogonal with the initial co-values.

Lemma 47. *If $\mathcal{A}$ and $\mathcal{B}$ are both reducibility candidates and c is a command such that for any $V \in \mathcal{A}$ and $E \in \mathcal{B}$, $c\{V_{\mathcal{S}}/x^{\mathcal{S}}, E_{\mathcal{T}}/\alpha^{\mathcal{T}}\} \in \perp\!\!\!\perp$ then $\mu[x^{\mathcal{S}} \circledR \alpha^{\mathcal{T}}.c] \in FunCore_{\mathcal{R}}(\mathcal{A}, \mathcal{B})$*

Proof. We know that $x^{\mathcal{S}} \in \mathcal{A}$ and $\alpha^{\mathcal{T}} \in \mathcal{B}$ so $c = c\{x^{\mathcal{S}}/x^{\mathcal{S}}, \alpha^{\mathcal{T}}/\alpha^{\mathcal{T}}\} \in \perp\!\!\!\perp$. Thus $\mu[x^{\mathcal{S}} \circledR \alpha^{\mathcal{T}}.c] \in SN^d_{\mathcal{R}}$. Now, for any $V \in \mathcal{A}$ and $E \in \mathcal{B}$ we have

$$\langle \mu[x^{\mathcal{S}} \circledR \alpha^{\mathcal{T}}.c] \| V \circledR E\rangle \mapsto_0 c\{V/x^{\mathcal{S}}, E/\alpha^{\mathcal{T}}\} \in \perp\!\!\!\perp$$

so by Lemma 36 $\langle \mu[x^{\mathcal{S}} \circledR \alpha^{\mathcal{T}}.c] \| V \circledR E\rangle \in \perp\!\!\!\perp$. Thus, $\mu[x^{\mathcal{S}} \circledR \alpha^{\mathcal{T}}.c] \in FunCore_{\mathcal{R}}(\mathcal{A}, \mathcal{B})$ □

We can define the semantic type constructor corresponding to the universal quantifier in much the same way. Note that while syntax representing types appears in terms and is important to type checking, we do not require that the syntactic representation of types be connected in any way to their semantic representation when defining the valid elements of a type.

Definition 13. *If S is a set of pre-types in $\mathcal{S}$ then $ForallCore^{\mathcal{T}}_{\mathcal{R}}(S)$ is the pre-type in $\mathcal{R}$ $Neg(\{A_{\mathcal{T}} \circledR E \mid \exists \mathcal{B} \in S, E \in \mathcal{B}\})$*

Lemma 48. *If S is a set of reducibility candidates in $\mathcal{S}$ then $ForallCore^{\mathcal{T}}_{\mathcal{R}}(S) = ForallCore^{\mathcal{T}}_{\mathcal{R}}(S)^{\perp d}$.*

Proof. $\{A_{\mathcal{T}} \circledR E | \exists \mathcal{B} \in S, E \in \mathcal{B}\}$ is a set of deterministic co-values so this follows by lemma 45. □

Lemma 49. *If S is a inhabited set of reducibility candidates in $\mathcal{S}$ and c is a command such that given any syntactic type $A_{\mathcal{T}}$, reducibility candidate $\mathcal{B} \in S$ and co-value $E \in \mathcal{B}$ we have $c\{A_{\mathcal{T}}/t^{\mathcal{T}}, E/\alpha^{\mathcal{S}}\} \in \perp\!\!\!\perp$ then $\mu[t^{\mathcal{T}} \circledR \alpha^{\mathcal{S}}.c] \in ForallCore^{\mathcal{T}}_{\mathcal{R}}(S)$.*

Proof. Since S is inhabited there is some $\mathcal{B} \in S$ and since $\mathcal{B}$ is a reducibility candidate, $\alpha^{\mathcal{S}} \in \mathcal{B}$. Thus, $c = c\{t^{\mathcal{T}}/t^{\mathcal{T}}, \alpha^{\mathcal{S}}/\alpha^{\mathcal{S}}\} \in \perp\!\!\!\perp$ so $\mu[t^{\mathcal{T}} \circledR \alpha^{\mathcal{S}}.c] \in SN^{d}_{\mathcal{R}}$. Moreover, given any $A_{\mathcal{T}}$, $\mathcal{B} \in S$, and $E \in \mathcal{B}$ we know that

$$\langle \mu[t^{\mathcal{T}} \circledR \alpha^{\mathcal{S}}.c] \| A_{\mathcal{T}} \circledR E \rangle \mapsto_0 c\{A_{\mathcal{T}}/t^{\mathcal{T}}, E/\alpha^{\mathcal{S}}\} \in \perp\!\!\!\perp$$

and so, by Lemma 36 $\langle \mu[t^{\mathcal{T}} \circledR \alpha^{\mathcal{S}}.c] \| A_{\mathcal{T}} \circledR E \rangle \in \perp\!\!\!\perp$. $\square$

The previous lemma required an inhabited set of reducibility candidates. But, if there are no reducibility candidates then this statement is vacuous. Luckily, we have all the machinery in place to ensure that at least one reducibility candidate exists as we know every set of deterministic co-values yields a candidate, including the *empty* set of co-values.

Lemma 50. *The set of reducibility candidates in a strategy is inhabited.*

Proof. $Neg(\emptyset) = Neg(\emptyset)^{\perp d}$ by 45 and so by Lemma 44 $\mathcal{R}(Neg(\emptyset)) \in CR_{\mathcal{S}}$. $\square$

Soundness

We now have all the machinery to define the semantics of types. Our semantic function $[\![-]\!]$ will convert syntactic types to semantic reducibility candidates, given a partial function ϕ mapping syntactic type variables to reducibility candidates.

$$\begin{aligned}
[\![a]\!](\phi) &\triangleq \phi(a) \\
[\![A \to B]\!](\phi) &\triangleq \mathcal{R}(FunCore([\![A]\!](\phi), [\![B]\!](\phi))) \\
[\![\forall a.A]\!](\phi) &\triangleq \mathcal{R}(ForallCore(\{[\![A]\!](\phi, a \mapsto \mathcal{B}) | \mathcal{B} \in CR\}))
\end{aligned}$$

Lemma 51. *If ϕ is defined and yields a reducibility candidate on all free type variables in A then $[\![A]\!](\phi) \in CR$*

Proof. By induction on $A_{\mathcal{S}}$.

- The case of a type variable is immediate.
- In the case of $[\![A_{\mathcal{S}_1} \rightarrow^{\mathcal{S}} B_{\mathcal{S}_2}]\!](\phi)$ we know by the inductive hypothesis that $[\![A_{\mathcal{S}_1}]\!](\phi) \in CR_{\mathcal{S}_1}$ and $[\![B_{\mathcal{S}_2}]\!](\phi) \in CR_{\mathcal{S}_2}$. Thus by Lemma 46 that $FunCore^{\mathcal{S}}([\![A_{\mathcal{S}_1}]\!](\phi), [\![B_{\mathcal{S}_2}]\!](\phi)) = FunCore^{\mathcal{S}}([\![A_{\mathcal{S}_1}]\!](\phi), [\![B_{\mathcal{S}_2}]\!](\phi))^{\perp d}$ and so by Lemma 44 that $[\![A \rightarrow^{\mathcal{S}} B]\!](\phi) \in CR_{\mathcal{S}}$.
- In the case of $[\![\forall^{\mathcal{S}} a^{\mathcal{S}_1}.A_{\mathcal{S}_2}]\!](\phi)$ we know by the inductive hypothesis that given any $\mathcal{B} \in CR_{\mathcal{S}_1}$, $[\![A_{\mathcal{S}_2}]\!](\phi, a^{\mathcal{S}_1} \mapsto \mathcal{B}) \in CR_{\mathcal{S}_2}$. Thus $\{[\![A_{\mathcal{S}_2}]\!](\phi, a^{\mathcal{S}_1} \mapsto \mathcal{B}) | \mathcal{B} \in CR_{\mathcal{S}_1}\}$ is a set of reducibility candidates and so by Lemma 48 we know that $ForallCore^{\mathcal{S}}_{\mathcal{S}_1}(\{[\![A_{\mathcal{S}_2}]\!](\phi, a^{\mathcal{S}_1} \mapsto \mathcal{B}) | \mathcal{B} \in CR_{\mathcal{S}_1}\}) = ForallCore^{\mathcal{S}}_{\mathcal{S}_1}(\{[\![A_{\mathcal{S}_2}]\!](\phi, a^{\mathcal{S}_1} \mapsto \mathcal{B}) | \mathcal{B} \in CR_{\mathcal{S}_1}\})^{\perp d}$ and so $[\![\forall^{\mathcal{S}} a^{\mathcal{S}_1}.A_{\mathcal{S}_2}]\!](\phi) \in CR_{\mathcal{S}}$ by Lemma 44. □

We extend our meaning function to sequents. As is standard, sequents classify substitutions mapping term variables to values, co-variables to co-values, and type variables to syntactic types in such a way that if $x : A$ in a sequent then the substitution classified by that sequent under ϕ maps x to a value which has the semantic type given by $[\![A]\!](\phi)$.

$$
\begin{aligned}
[\![\vdash]\!](\phi) &\triangleq \text{the Set of Substitutions} \\
[\![\vdash_{\Theta,a}]\!](\phi) &\triangleq \{\rho \in [\![\vdash_{\Theta}]\!](\phi) | \rho(a) \text{ a syntactic type}\} \\
[\![\vdash_{\Theta} \alpha : A, \Delta]\!](\phi) &\triangleq \{\rho \in [\![\vdash_{\Theta} \Delta]\!](\phi) | \rho(\alpha) \in [\![A]\!](\phi)\} \\
[\![\Gamma, x : A \vdash_{\Theta} \Delta]\!](\phi) &\triangleq \{\rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!](\phi) | \rho(x) \in [\![A]\!](\phi)\}
\end{aligned}
$$

Lemma 52. *If $\Gamma \vdash_{\Theta} \Delta$ is well formed and ϕ is defined and yields a reducibility candidate on all type variables in Θ, then $[\![\Gamma \vdash_{\Theta} \Delta]\!](\phi)$ is well defined and contains the identity substitution (id).*

Proof. By induction over length of the sequent using 51 for the cases of variables and co-variables. □

We can now give the semantic interpretation of each of the typing judgments.

$$
\begin{aligned}
c : (\Gamma \vDash_{\Theta} \Delta) &\triangleq \forall(\phi \in \Theta \to CR), \forall \rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!](\phi), \\
&\quad c\{\rho\} \in \perp\!\!\!\perp \\
\Gamma \vDash_{\Theta} v : A | \Delta &\triangleq \forall(\phi \in \Theta \to CR), \forall \rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!](\phi), \\
&\quad v\{\rho\} \in [\![A]\!](\phi) \\
\Gamma | e : A \vDash_{\Theta} \Delta &\triangleq \forall(\phi \in \Theta \to CR), \forall \rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!](\phi), \\
&\quad e\{\rho\} \in [\![A]\!](\phi)
\end{aligned}
$$

With the definitions out of the way, we can state our primary theorem: any judgment which is provable according to the proof rules corresponds to a semantically true statement in our model.

Theorem 15 (Soundness). – *If there is a proof of* $c : (\Gamma \vdash_\Theta \Delta)$ *then* $c : (\Gamma \vDash_\Theta \Delta)$,

– *if there is a proof of* $\Gamma \vdash_\Theta v : A | \Delta$ *then* $\Gamma \vDash_\Theta v : A | \Delta$*, and*

– *if there is a proof of* $\Gamma | e : A \vdash_\Theta \Delta$ *then* $\Gamma | e : A \vDash_\Theta \Delta$.

Proof. By mutual induction on the typing derivation. The *Cut* rule follows from orthogonal soundness of candidates and Lemma 51. While Lemma 37, Lemma 38, Lemma 47, Lemma 49, and Lemma 50 are used for the other rules. □

Corollary 3 (Strong Normalization). *If* $c : (\Gamma \vdash_\Theta \Delta)$ *is provable then* c *is strongly normalizing.*

Proof. If $c : (\Gamma \vdash_\Theta \Delta)$ is provable then it is well formed. Moreover, since by Lemma 50 the set of reducibility candidates is inhabited, there must exist some $\phi : \Theta \to CR$. Further, by Lemma 52 $id \in [\![\Gamma \vdash_\Theta \Delta]\!](\phi)$. Therefore, by Theorem 15 $c\{id\} \in \perp\!\!\!\perp$. □

Extensions

Our proof can be quite readily extended to account for additional typing constructors, such as a positive sum type defined by two constructor terms $\iota_1(v)$ and $\iota_2(v)$. As observed in the previous chapter, unlike the function and universal co-data type constructors, defined by their co-value formers, sums are data types. Moreover, they represent an additive type constructor (where there are multiple options) while, so far, we have only considered multiplicative types (where there are multiple components).

Adding the sum types presents no major challenge to the proof; only a small number of modifications are needed. First, we must extend our notion of head reduction to account for the additional positive ς reductions and neutral β reductions. The central commutation lemma (Lemma 34) continues to hold. Moreover, we can extend the unfocalization lemma (Lemma 38) with the two new constructor forms: if $\mathcal{A}$ and $\mathcal{B}$ are reducibility candidates such that $\iota_1(V) \in \mathcal{B}$ for every value $V \in \mathcal{A}$, then it must be that $\iota_1(v) \in \mathcal{B}$ for all terms $v \in \mathcal{A}$.

Constructing seeds for data types is dual to co-data types: starting with a set of simple values and finding first the simple co-values which are orthogonal to those and then all the simple values which are orthogonal to the co-values. All the remaining cases of the soundness proof follow similarly.

Analogously, the system can be extended with negative sums as well as positive and negative products and a type for existential quantification. The changes are restricted to places which explicitly mention specific types and their constructors: namely the unfocalization lemma, the interaction of head and regular reduction, and in the handling of the specific typing rules for the soundness proof.

Variations and Applications

Untyped Sub-Theories

A more exotic alteration of our proof emerges from a desire to better understand the untyped $\mu\tilde{\mu}$ calculus. It turns out that the μ, $\tilde{\mu}$ and ς rules alone, without β, represent a core language of substitution and focusing which is strongly normalizing.

Theorem 16. *The $\to_{\mu_E,\tilde{\mu}_V,\eta_\mu,\eta_{\tilde{\mu}},\varsigma}$ reduction theory is strongly normalizing on untyped programs for any stable and focalizing strategy.*

That the $\to_{\mu_E,\tilde{\mu}_v}$ theory is strongly normalizing was already established in Polonovski (2004). Using our technique we can also include reductions like ς. The idea is that, in absence of β reduction, "type" stops being a useful concept. Instead, there is really just one type and so we require only a single reducibility candidate. Specifically, our proof can proceed as before, only without the β reductions in the theory. As such, there are *no* neutrally charged reductions at all. Otherwise though, the definitions and lemmas of the proof are unchanged and there are fewer cases to consider in the commutation lemma. The single reducibility candidate we need, $\mathcal{X}$, is built via the fixed-point from the seed containing *all* strongly normalizing simple (co-)terms. Then by induction on the syntax tree, it can be shown that if σ is a substitution mapping variables to values in $\mathcal{X}$ and co-variables to co-values in $\mathcal{X}$, that

1. For any term v, $v\{\sigma\} \in \mathcal{X}$
2. For any co-term e, $e\{\sigma\} \in \mathcal{X}$
3. For any command c, $c\{\sigma\} \in \perp\!\!\!\perp$.

Moreover, variables and co-variables are in $\mathcal{X}$, meaning all commands are strongly normalizing in this reduced system.

Strong normalization of the $\mu\tilde{\mu}$-calculus is unique to the two sided sequent calculus. If variables and co-variables are interchangeable we get no such result as we can encode infinite loops by double self-application. This suggests that distinguishing terms and co-terms is a sort of type system, albeit a minimal one. Strong normalizing of the $\mu\tilde{\mu}$ component of the syntax was shown in Polonovski

(2004) using symmetric candidates. The argument above extends that result to the full syntax and accounts for the ς rules. Because it is strategy parametric, it works for the non-deterministic strategy where every (co-)term is a (co-)value, which is exactly the same as the core $\mu\tilde{\mu}$ calculus without any value restriction in substitution. This goes to show that studying the models of types can teach us things even about reduction in untyped languages.

A Fixed Point Characterization of Candidates

In this chapter we defined reducibility candidates as pre-types which satisfied two conditions. First, they were required to not be too big by insisting that they be orthogonally sound. Second, they were required to not be too small by insisting that they be saturated. All properties of interest came from that definition and it seems one that works well in practice. However we might instead want a simpler definition. Namely, we proved in Theorem 14 that all reducibility candidates were fixed-points of the saturation function. Here we observe that we could take this as definition as well.

Theorem 17 (Fixed Point Characterization)**.** *A pre-type $\mathcal{A}$ is a reducibility candidate if and only if $\mathcal{A} = Sat(\mathcal{A})$*

Proof. The only if direction was proved in Theorem 14. For the if direction we observe that the empty pre-type 0 containing no terms or co-terms is forward closed and

$$0 \sqsubseteq 0^{\perp d}.$$

Thus whenever

$$\mathcal{A} = Sat(\mathcal{A}) = Step_0(\mathcal{A})$$

we have met the conditions in Lemma 44 taking 0 as the seed and so know that $\mathcal{A}$ is a candidate. □

Characterizing candidates as fixed-point solutions of the $Sat(-)$ function has a certain elegance since we also *construct* candidates as fixed-point solutions to a slightly more complicated function. Moreover, we can use it to derive some interesting facts about the space of candidates. Namely, that the set of candidates under subtyping has all meets and joins.

Theorem 18. *The set of reducibility candidates in a strategy forms a complete lattice when ordered by subtyping ($\leq$).*

Proof. Candidates are fixed-points of $Sat(-)$ by Theorem 17. And, $Sat(-)$ is a monotonic function with respect to $\leq$. So the existence of a complete lattice of fixed-points is given by Tarski's fixed-point theorem. □

Generalized Orthogonality

Through our proof we have used operations like $(-)^{\perp}$ and $(-)^{\perp d}$ which share some common structure.

We can generalize the orthogonal operation to a general class of similar operations on pre-types that all share the same properties. We say that an operation Op on pre-types in $\mathcal{S}$ is a *negation operation inside* $\mathcal{D}$ if and only if $\mathcal{D}$ is a fixed pre-type in $\mathcal{S}$ and there exists predicates P and Q on term, co-term pairs such that:

$$v \in Op(\mathcal{A}) \iff v \in \mathcal{D} \wedge \forall e \in \mathcal{A}.P(v,e)$$

$$e \in Op(\mathcal{A}) \iff e \in \mathcal{D} \wedge \forall v \in \mathcal{A}.Q(v,e)$$

We write $Op^\star$ for the *adjoint* negation operator to Op which is formed by reversing P and Q, namely:

$$v \in Op^\star(\mathcal{A}) \iff v \in \mathcal{D} \wedge \forall e \in \mathcal{A}.Q(v,e)$$
$$e \in Op^\star(\mathcal{A}) \iff e \in \mathcal{D} \wedge \forall v \in \mathcal{A}.P(v,e)$$

It follows that $Op^{\star\star} = Op$ and that $Op^\star$ is a negation operation inside $\mathcal{D}$ whenever Op is.

Furthermore, Op is a *symmetric negation operation inside* $\mathcal{D}$ when $Op = Op^\star$ which holds if and only if for all $v, e \in \mathcal{D}$, $P(v,e) \iff Q(v,e)$. Note that orthogonality is a symmetric negation operation inside $SN_{\mathcal{S}}$.

It follows that all such negation operations enjoy the standard basic properties of orthogonality.

Lemma 53 (Monotonicity)**.** *For any negation operation Op inside $\mathcal{D}$, $\mathcal{A} \leq \mathcal{B}$ implies $Op(\mathcal{A}) \leq Op(\mathcal{B})$.*

Proof. Suppose $v \in Op(\mathcal{A})$, so we know that $v \in \mathcal{D}$ and for all $e \in \mathcal{A}$, $P(v,e)$. Then, given any $e \in \mathcal{B}$, we know that $e \in \mathcal{A}$ because $\mathcal{A} \leq \mathcal{B}$, and so $P(v,e)$. Therefore, $v \in Op(\mathcal{B})$ as well.

Suppose $e \in Op(\mathcal{B})$, so we know that $e \in \mathcal{D}$ and for all $e \in \mathcal{B}$, $Q(v,e)$. Then, given any $v \in \mathcal{A}$, we know that $v \in \mathcal{B}$ because $\mathcal{A} \leq \mathcal{B}$, and so $Q(v,e)$. Therefore, $e \in Op(\mathcal{A})$ as well. □

Lemma 54 (Contrapositive)**.** *For any negation operation Op inside $\mathcal{D}$, $\mathcal{A} \sqsubseteq \mathcal{B}$ implies $Op(\mathcal{B}) \sqsubseteq Op(\mathcal{A})$.*

Proof. Suppose $v \in Op(\mathcal{B})$, so we know that $v \in \mathcal{D}$ and for all $e \in \mathcal{B}$, $P(v, e)$. Then, given any $e \in \mathcal{A}$, we know that $e \in \mathcal{B}$ because $\mathcal{A} \sqsubseteq \mathcal{B}$, and so $P(v, e)$. Therefore, $v \in Op(\mathcal{B})$ as well.

Suppose $e \in Op(\mathcal{B})$, so we know that $e \in \mathcal{D}$ and for all $v \in \mathcal{B}$, $Q(v, e)$. Then, given any $v \in \mathcal{A}$, we know that $v \in \mathcal{B}$ because $\mathcal{A} \sqsubseteq \mathcal{B}$, and so $Q(v, e)$. Therefore, $e \in Op(\mathcal{B})$ as well. □

Lemma 55 (Double Negation Introduction)**.** *For any negation operation Op inside $\mathcal{D}$, $\mathcal{A} \sqsubseteq \mathcal{D}$ implies $\mathcal{A} \sqsubseteq Op^\star(Op(\mathcal{A}))$.*

Proof. For any $v \in \mathcal{A}$ and $e \in Op(\mathcal{A})$, $Q(v, e)$ by definition of $Op(\mathcal{A})$, so $P(v, e)$ as well since Op is symmetric. Therefore, $\mathcal{A} \sqsubseteq \mathcal{D}$ implies that $v \in \mathcal{D}$, so that $v \in Op^\star(Op(\mathcal{A}))$. The case of $e \in \mathcal{A}$ implies $e \in Op^\star(Op(\mathcal{A}))$ is dual. □

Lemma 56 (Triple Negation Elimination)**.** *For any negation operation Op inside $\mathcal{D}$, $\mathcal{A} \sqsubseteq \mathcal{D}$ implies $Op(Op^\star(Op(\mathcal{A}))) = Op(\mathcal{A})$.*

Proof. Note that $Op(\mathcal{A}) \sqsubseteq \mathcal{D}$ because Op is a negation operation inside $\mathcal{D}$. Therefore, by double negation introduction (Lemma 55), $Op(\mathcal{A}) \sqsubseteq Op(Op^\star(Op(\mathcal{A})))$. Additionally, because $\mathcal{A} \sqsubseteq \mathcal{D}$, we know that $\mathcal{A} \sqsubseteq Op^\star(Op(\mathcal{A}))$ by double negation introduction (Lemma 55), and so $Op(Op^\star(Op(\mathcal{A}))) \sqsubseteq Op(\mathcal{A})$ by contrapositive (Lemma 54). Therefore, $Op(Op^\star(Op(\mathcal{A}))) = Op(\mathcal{A})$. □

These properties of a symmetric negation operation are exactly the properties of a Galois injection. Indeed, Lemmas 54 and 55 together are exactly the statement that Op and $Op^\star$ are adjoint contravariant functors with respect to $\sqsubseteq$. This suggests a fundamental connection with the theory of abstract interpretation which we will not pursue at this time. Further, it suggests the next theorem.

Lemma 57 (De Morgan Law)**.** *For any symmetric negation operation Op inside $\mathcal{D}$, if $\mathcal{A} \sqsubseteq \mathcal{D}$ and $\mathcal{B} \sqsubseteq \mathcal{D}$ then $Op(A \sqcup B) = Op(A) \sqcap Op(B)$.*

Proof. Our proof proceeds by inclusion in both directions. Since $A \sqsubseteq A \sqcup B$ we know by contrapositive (Lemma 54) that $Op(A \sqcup B) \sqsubseteq Op(A)$. Symmetrically, $Op(A \sqcup B) \sqsubseteq Op(B)$. Thus, $Op(A \sqcup B) \sqsubseteq Op(A) \sqcap Op(B)$. $Op(A) \sqcap Op(B) \sqsubseteq Op(A)$ and so by contrapositive, $Op^\star(Op(A)) \sqsubseteq Op^\star(Op(A) \sqcap Op(B))$. So, by double negation introduction (Lemma 55) $A \sqsubseteq Op^\star(Op(A)) \sqsubseteq Op^\star(Op(A) \sqcap Op(B))$. Symmetrically, $B \sqsubseteq Op^\star(Op(A) \sqcap Op(B))$. So, $A \sqcup B \sqsubseteq Op^\star(Op(A) \sqcap Op(B))$ and so by contrapositive and double negation elimination $Op(A) \sqcap Op(B) \sqsubseteq Op(Op^\star(Op(A) \sqcap Op(B))) \sqsubseteq Op(A \sqcup B)$. □

A *restriction operation* is one of the form $Res(\mathcal{A}) = \mathcal{A} \sqcap \mathcal{B}$ for some fixed pre-type $\mathcal{B}$. All such operations are monotonic with respect to both orders. Moreover, if $Op(-)$ is a negation operation inside $\mathcal{D}$ and $Res(\mathcal{A}) = \mathcal{A} \sqcap \mathcal{B}$ for all $\mathcal{A}$ then $Op(Res(-))$ is a negation operation inside $\mathcal{D}$ and $Res(Op(-))$ is a negation operation inside $\mathcal{D} \sqcap \mathcal{B}$. Moreover, if $Op(-)$ is a symmetric negation operation inside $\mathcal{D}$ then $Res(Op(Res(-)))$ is a symmetric negation operation inside $\mathcal{D} \sqcap \mathcal{B}$.

When we consider both negations and restrictions together, however, they admit some additional properties. In particular, given a symmetric negation operation $Op(-)$ and a restriction operation $Res(-)$, we have the following restricted versions of double negation introduction and triple negation elimination:

– *restricted double negation introduction*:

$$\mathcal{A} \sqsubseteq Op(Res(Op(\mathcal{A})))$$

Proof. Note that we know $\mathcal{A} \sqsubseteq Op(Op(\mathcal{A}))$ by unrestricted double negation introduction. Furthermore, we have $Res(Op(\mathcal{A})) \sqsubseteq Op(\mathcal{A})$ by inclusion and thus $Op(Op(\mathcal{A})) \sqsubseteq Op(Res(Op(\mathcal{A})))$ by contrapositive. Therefore, $\mathcal{A} \sqsubseteq Op(Op(\mathcal{A})) \sqsubseteq Op(Res(Op(\mathcal{A})))$. $\square$

– *restricted triple negation elimination*:

$$Res(Op(Res(Op(Res(Op(Res(\mathcal{A}))))))) = Res(Op(Res(\mathcal{A})))$$

Proof. First, observe that

$$Res(\mathcal{A}) \sqsubseteq Op(Res(Op(Res(\mathcal{A}))))$$

by restricted double negation introduction, and so monotonicity and idempotency of $Res(-)$,

$$Res(\mathcal{A}) \sqsubseteq Res(Op(Res(Op(Res(\mathcal{A})))))$$

therefore, by contrapositive of $Op(-)$

$$Op(Res(Op(Res(Op(Res(\mathcal{A})))))) \sqsubseteq Op(Res(\mathcal{A}))$$

and so by monotonicity

$$Res(Op(Res(Op(Res(Op(Res(\mathcal{A}))))))) \sqsubseteq Res(Op(Res(\mathcal{A})))$$

Second, observe that

$$Res(Op(Res(\mathcal{A}))) \sqsubseteq Op(Res(Op(Res(Op(Res(\mathcal{A}))))))$$

also by restricted double negation introduction, and so

$$Res(Op(Res(\mathcal{A}))) \sqsubseteq Res(Op(Res(Op(Res(Op(Res(\mathcal{A})))))))$$

by monotonicity and idempotency of $Res(-)$. Therefore,

$$Res(Op(Res(Op(Res(Op(Res(\mathcal{A}))))))) = Res(Op(Res(\mathcal{A}))) \ . \qquad \square$$

Note that the unrestricted double negation introduction and triple negation elimination principles follow from the above restricted variants, since the identity operation on pre-types is a restriction operation.

We will also use the following notation for constructing a new pre-type by "cutting" together two existing ones:

$$v \in \langle \mathcal{A} \| \mathcal{B} \rangle \iff v \in \mathcal{A} \qquad\qquad e \in \langle \mathcal{A} \| \mathcal{B} \rangle \iff e \in \mathcal{B}$$

As usual, the cut operation on pre-types has its own properties based on the two orderings of pre-types:

- *self-cut*: $\mathcal{A} = \langle \mathcal{A} \| \mathcal{A} \rangle$
- *cross-cut*: $\langle \mathcal{A} \| \mathcal{B} \rangle \sqsubseteq \langle \mathcal{A}' \| \mathcal{B}' \rangle$ and $\langle \mathcal{C} \| \mathcal{D} \rangle \sqsubseteq \langle \mathcal{C}' \| \mathcal{D}' \rangle$ implies $\langle \mathcal{A} \| \mathcal{D} \rangle \sqsubseteq \langle \mathcal{A}' \| \mathcal{D}' \rangle$
- *order exchange*: $\langle \mathcal{A} \| \mathcal{B} \rangle \sqsubseteq \langle \mathcal{C} \| \mathcal{D} \rangle$ iff $\langle \mathcal{A} \| \mathcal{D} \rangle \leq \langle \mathcal{C} \| \mathcal{B} \rangle$

Observe that as a consequence of self-cut and cross-cut for refinement, we have the additional cross-cut property that $\mathcal{A} = \langle \mathcal{B} \| \mathcal{B}' \rangle$ and $\mathcal{A} = \langle \mathcal{C} \| \mathcal{C}' \rangle$ implies that $\mathcal{A} = \langle \mathcal{B} \| \mathcal{C}' \rangle$. Also note the additional properties of how cutting semantic types interacts with arbitrary negation operations $Op(-)$ and restriction operations $Res(-)$:

- *cut negation*: $Op(\langle \mathcal{A} \| \mathcal{B} \rangle) = \langle Op(\mathcal{B}) \| Op(\mathcal{A}) \rangle$
- *cut restriction*: $Res(\langle \mathcal{A} \| \mathcal{B} \rangle) = \langle Res(\mathcal{A}) \| Res(\mathcal{B}) \rangle$

Lemma 58. *Given $\mathcal{C} = \mathcal{C}^v \sqsubseteq \mathcal{A} = \mathcal{A}^{\perp} = \mathcal{A}^{v\perp}$, then $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{C}^{\perp v \perp}$ iff $\mathcal{A} = \mathcal{C}^{\perp v \perp}$.*

Proof. First we show the "only if" direction, so assume that $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{C}^{\perp v \perp}$. We get $\mathcal{A} = \mathcal{A}^{\perp} \sqsubseteq \mathcal{C}^{\perp}$ by refinement contrapositive of $\mathcal{C} \sqsubseteq \mathcal{A}$ with the negation operation $\_^{\perp}$, and likewise $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{C}^{\perp v \perp} \sqsubseteq \mathcal{A}^{v\perp} = \mathcal{A}$ by refinement contrapositive of $\mathcal{A} \sqsubseteq \mathcal{C}^{\perp}$ with the negation operation $\_^{v\perp}$. Thus, we get $\mathcal{A} = \mathcal{A}^{\perp} \sqsubseteq \mathcal{C}^{\perp v \perp}$ by refinement contrapositive of $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{A}$ with $\_^{\perp}$ again. Therefore, $\mathcal{A} = \mathcal{C}^{\perp v \perp}$ since both $\mathcal{A} \sqsubseteq \mathcal{C}^{\perp v \perp}$ and $\mathcal{C}^{\perp v \perp} \sqsubseteq \mathcal{A}$ hold.

Second, we show the "if" direction, so assume that $\mathcal{A} = \mathcal{C}^{\perp v \perp}$. By assumption, we have that $\mathcal{A}^v = \mathcal{A}^{v \perp v} = \mathcal{C}^{\perp v \perp v \perp v}$, so $\mathcal{A}^v = \mathcal{C}^{\perp v}$ by restricted triple negation elimination with $\_^{v \perp v}$. Therefore, $\mathcal{C}^{\perp v} = \mathcal{A}^v \sqsubseteq \mathcal{A} = \mathcal{C}^{\perp v \perp}$ by refinement inclusion of $\_^v$. $\square$

Lemma 59. *For any $\mathcal{C}$, $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{C}^{\perp v \perp}$ whenever either (1) $\langle \mathcal{C}^{\perp v} \| \mathcal{C} \rangle \sqsubseteq \mathcal{C}$, or (2) $\langle \mathcal{C} \| \mathcal{C}^{\perp v} \rangle \sqsubseteq \mathcal{C}$.*

Proof.

1. We have $\mathcal{C}^{\perp v} \sqsubseteq \langle \mathcal{C}^{\perp v} \| \mathcal{C} \rangle^{\perp v} = \langle \mathcal{C}^{\perp v} \| \mathcal{C}^{\perp v \perp v} \rangle$ by contrapositive and cut negation with $\_^{\perp v}$ on the assumption $\langle \mathcal{C}^{\perp v} \| \mathcal{C} \rangle \sqsubseteq \mathcal{C}$. Thus, for any $V, E \in \mathcal{C}^{\perp v}$, we also know that $E \in \mathcal{C}^{\perp v \perp v}$, so $\langle V \| E \rangle \in \perp\!\!\!\perp$. Therefore, $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{C}^{\perp v \perp}$.
2. Analogous to the proof of part 1 by duality. $\square$

Related Work

Our strong normalization proof builds on powerful techniques already existing in the literature. By combining them into a coherent whole we find an approach suitable for proving our strategy parametric theorem. In this section, we review the development of these techniques and how we build on them.

Reducibility Candidates

The approach to strong normalization where types are interpreted as candidates traces back to Tait (1967) and Girard (1972). Moreover, such strong normalization proofs are closely related to other lines of research including logical relations (Wadler (1989)) and realizability (Kleene (1945)). The candidates-based approach is compelling in part because it easily accommodates impredicative polymorphism, as we have in this paper, by defining the set of potential candidates from the start before any particular type, allowing that set to be quantified over in the definition of polymorphic types as *elements* of that set.

Orthogonality

Tait's method as originally constituted is not suitable in our setting because we need types to classify co-terms in addition to terms. The idea of orthogonality appears in multiple places, including Girard's development of linear logic (Girard (1987)), Krivine's analysis of classical realizability (Krivine (2009)), and Pitts' $\top\top$-closed logical relations (Pitts (2000)). A pure orthogonality-based proof of strong normalization is suitable for certain strategies. For example, if we wished to prove strong normalization only for call-by-name, we could start by defining the types in terms of their observations—so for functions, valid call stacks—and then define their set of terms orthogonally as anything which runs with any of these observations, and, finally, their set of co-terms as the double orthogonal. A dual approach, starting with the constructions of values, would work for call-by-value.

Munch-Maccagnoni (2009) identified a key feature of the construction of a type from its orthogonal: all types are generated by their values. That is, the denotation of a type is the orthogonal of its (co-)values. In our language, $\mathcal{A} = \mathcal{A}^{v\perp}$. This property is not needed to make soundness go through in our proof, but it seems to hold practically "for free" in the construction. Indeed, soundness would have been provable with the weaker condition in the definition of reducibility candidates that every candidate contain its own $Head$, instead of having every candidate be Sat. However, all candidates we actually generate are Sat and so we have included this as part of the definition for clarity. Moreover, the fact that all types are generated by their values allows us to relate our fixed-point construction to orthogonality. Specifically, we observe that the reducibility candidates we produce in the case of the call-by-value and call-by-name strategies are uniquely defined similar to the direct definitions from the orthogonal construction of types. We could show this by showing that the types generated by orthogonality are the (unique) solutions to our fixed-point equations, but completeness means that it is enough to know that we have generated candidates which include their seed.

Theorem 19. *Suppose that head reduction in the chosen strategy is deterministic. Then, for any reducibility candidate $\mathcal{A}$ and $\mathcal{C} = \mathcal{C}^{\perp d} \sqsubseteq \mathcal{A}$, it must be that $\mathcal{A} = \mathcal{C}^{\perp v \perp}$.*

Proof. We prove the case where all values are simple terms as the other case is dual. We know that $\mathcal{C} = \mathcal{C}^{\perp d}$. Thus,

$$\begin{aligned}\mathcal{C} &= \langle \mathcal{C} \| \mathcal{C} \rangle \\ &= \langle \mathcal{C}^{\perp s} \| \mathcal{C} \rangle \\ &= \langle \mathcal{C}^{\perp v} \| \mathcal{C} \rangle\end{aligned}$$

The last step following from the assumption that all values are deterministic. Therefore, by Lemma 59 it must be that $\mathcal{C}^{\perp v} \sqsubseteq \mathcal{C}^{\perp v \perp}$ and so by Lemma 58 that $\mathcal{A} = \mathcal{C}^{\perp v \perp}$. □

Corollary 4. *For any pre-type $\mathcal{C} = \mathcal{C}^{v}$ of (co-)values,*

– $(\{V|V \in \mathcal{C}\}, \{E|E \in \mathcal{C}^{\perp}\})^{\perp}$ *is a reducibility candidate if $\mathcal{C}$ is in $\mathcal{V}$, and*

– $(\{V|V \in \mathcal{C}^{\perp}\}, \{E|E \in \mathcal{C}\})^{\perp}$ *is a reducibility candidate if $\mathcal{C}$ is in $\mathcal{N}$.*

Proof. 1. Note that this can be restated as saying $\langle \mathcal{C} \| \mathcal{C}^{\perp} \rangle^{\perp}$ is a reducibility candidate whenever $\mathcal{C}$ is in $\mathcal{V}$.

Let $\mathcal{A} = \langle \mathcal{C} \| \mathcal{C}^{\perp v} \rangle^{\perp v \perp}$. Because $\mathcal{V}$ head reduction is deterministic, we already know that $\mathcal{A}$ is a reducibility candidate by Theorem 19. Furthermore, because every co-term is a co-value in the $\mathcal{V}$ strategy, we also know that $\langle \mathcal{D} \| \mathcal{E}^{v} \rangle =$

$\langle \mathcal{D} \| \mathcal{E} \rangle$ for any pre-types $\mathcal{D}$ and $\mathcal{E}$ in $\mathcal{V}$. Therefore, it follows that:

$$\begin{aligned}
\mathcal{A} &= \langle \mathcal{C} \| \mathcal{C}^{\perp v} \rangle^{\perp v \perp} && \text{by definition} \\
&= \langle \mathcal{C}^{\perp v \perp} \| \mathcal{C}^{\perp v \perp v \perp} \rangle && \text{by cut negation and restriction} \\
&= \langle \mathcal{C}^{\perp v \perp} \| \mathcal{C}^{\perp v \perp v \perp v} \rangle && \text{by } \mathcal{V} \\
&= \langle \mathcal{C}^{\perp v \perp} \| \mathcal{C}^{v \perp v \perp v \perp v} \rangle && \text{by assumption } \mathcal{C} = \mathcal{C}^{v} \\
&= \langle \mathcal{C}^{\perp v \perp} \| \mathcal{C}^{v \perp v} \rangle && \text{by restricted triple negation elimination} \\
&= \langle \mathcal{C}^{\perp v \perp} \| \mathcal{C}^{\perp} \rangle && \text{by } \mathcal{V} \text{ and } \mathcal{C} = \mathcal{C}^{v} \\
&= \mathcal{A}^{\perp} = \langle \mathcal{C}^{\perp v \perp} \| \mathcal{C}^{\perp} \rangle^{\perp} && \text{by reducability candidacy of } \mathcal{A} \\
&= \langle \mathcal{C}^{\perp \perp} \| \mathcal{C}^{\perp v \perp \perp} \rangle && \text{by cut negation} \\
&= \langle \mathcal{C}^{\perp \perp} \| \mathcal{C}^{\perp} \rangle && \text{by cross cut} \\
&= \langle \mathcal{C} \| \mathcal{C}^{\perp} \rangle^{\perp} && \text{by cut negation}
\end{aligned}$$

2. Analogous to part 1 by duality. □

Thus, we see that the reducibility candidates generated by the fixed-point in our method for call-by-value and call-by-name could also be generated *purely* using repeated use of orthogonality and the restrictions to (co-)values. Moreover, the solution is *unique*: any orthogonality-based construction which ends in a candidate and includes all the expected deterministic (co-)terms from the seed is ultimately producing the same object.

We might wonder then about the purpose of the fixed-point construction. It seems orthogonality is simpler and more direct, giving an exact construction of candidates without relying on a potentially non-constructive fixed-point theorem.

However, the basic orthogonality approach is not strategy agnostic, so that the model is constrained by the chosen strategy. Worse, it appears that it alone is simply not strong enough to handle all strategies of interest. The problem is clearest in the non-deterministic strategy $\mathcal{U}$ where everything is a (co-)value. In such a setting we can try to *define* the type constructors starting with either constructions or observations and then by orthogonality. However, we run into a problem when we seek to prove the soundness of the μ and $\tilde{\mu}$ rules. In the soundness proof, we have the inductive hypothesis that for any term v in the candidate, $c\{v/x\}$ is strongly normalizing. Unfortunately, with orthogonality, it does not follow that $\tilde{\mu}x.c$ is in the candidate. That is because the co-term might contain a $\mu\alpha.c'$ where $c\{\mu\alpha.c'/x\}$ is strongly normalizing but $c'\{\tilde{\mu}x.c/\alpha\}$ isn't. This is avoided in the case of call-by-name since $\tilde{\mu}x.c$ is not a co-value. However, the problem seems fundamental in the case of the non-deterministic strategy (Lengrand and Miquel (2008)), and is difficult to tame in more complicated strategies like call-by-need.

Thus, while the basic orthogonality method corresponds to a special case of our proof technique for the call-by-value and call-by-name strategies, it is strictly less general. The call-by-need case is particularly interesting: Theorem 19 gives us a method for directly constructing call-by-need candidates from an initial set of (co-)values by using orthogonality a total of four times together with intermediate restrictions down to a set of values. Unlike in the case of call-by-name and call-by-value, we have neither that all terms are values or all co-terms are co-values, and so none of these uses appear to cancel out. We discovered and know this direct construction yields candidates because we have proven it equal to the result of our strategy agnostic method, however, on its own its behaviour seems rather inscruitable and it seems notable that it was not previously discovered.

Symmetric Candidates

The symmetric candidates method introduced by Barbanera and Berardi (Barbanera and Berardi (1994)) to prove strong normalization for the symmetric λ-calculus—a system using the non-deterministic strategy $\mathcal{U}$—avoids the problem with orthogonality by defining types via a function on sets of terms which explicitly includes the problematic syntactic constructs such as μ. Types are then interpreted as fixed-points of these functions.

We have attempted to simplify the presentation of the fixed-point construction in the present work by modeling types as pairs of sets of terms and sets of co-terms, where Barbanera and Berardi used only sets of terms. Where they used a pair of anti-monotone maps, whose composition is guaranteed to have a fixed-point, we use a single map which is monotone with respect to the sub-typing order. As a consequence their original presentation concealed the role of behavioral sub-typing in ensuring that fixed-point solutions for candidates exist.

A more fundamental difference with Barbanera and Berardi has to do with how the function whose fixed-point we take is defined. While Barbanera and Berardi's definition is syntactic, our definition is in terms of how (co-)terms behave under (head) reduction. This change is important not only for simplicity and conceptual elegance, but also for extensibility. While Barbanera and Berardi, as well as Lengrand and Miquel (2008) who use the same technique, consider only the non-deterministic strategy $\mathcal{U}$ where everything is a (co-)value, we are interested also in strategies that have non-(co-)values. Having non-(co-)values enables additional reductions, namely ς, and this presents a problem both in defining the type and in the soundness proof if a purely syntactic definition is used. The problem is that a (co-)term which is not a μ or $\tilde{\mu}$ abstraction, such as a call-stack, might reduce to one which is. Thus, we need to be sure that we only include such (co-)terms which only reduce to things which would otherwise be in the type, and we have no syntactic criteria for ensuring this. Instead, by using how terms behave under head reduction, we are able to ensure we encode all of the (co-)terms we need in a type.

However, in the case of $\mathcal{U}$ where everything is a co-value, the ς rules never fire and so the saturation in our fixed-point construction exactly corresponds to symmetric candidates. Thus, we can see our proof as generalizing that proof as well.

Ultimately, we see a trade-off, while the orthogonality method works behaviorally and so handles the lifting rules and unfocalization without issue it breaks down in handling activation in more exotic strategies. Conversely, symmetric candidates always handles activation but not unfocalization in strategies where there are non-(co-)value. As call-by-need exposes both problems, it alone demands a proof technique which gets the benefits of both. Moreover, by synthesizing these techniques into a cohesive whole we have a singular model construction and proof which works across a whole range of strategies.

CHAPTER VII

CONCLUSION

In this dissertation we have studied languages inspired by a Curry-Howard correspondence with classical sequent calculus. This setting which emphasizes the dual-sided nature of computation has been instructive in several ways. Perhaps most importantly, it has helped draw attention to the idea, which we have returned to again and again through out the dissertation, that functions are a co-data, defined by how they are used. The idea of functions as pattern matching on call-stacks provides the opening for the alternative implementation of functions in Chapter III in terms of projection operations. That alternative view allows us to wrestle with the tension between lazy extensionality and control by way of a confluent rewriting system equivalent to our desired equational theory. It also came to use in Chapter IV even without control in directing us towards an approach to closed head-reduction and thus a solution to the conflict between weak head reduction and extensionality that happens even in the pure λ-calculus.

The dual-sided approach has continued to be helpful as we turned our attention to typed systems and strategies other than just call-by-name. In particular, our reducibility candidates based proof is based on the idea of seeing types semantically as two-sided entities, classifying both terms and co-terms. Our proof suggests that it isn't enough to look at just what terms inhabit a type. Indeed, it was observed more than 40 years ago that "types are not sets" Morris (n.d.). While term models interpreting types as just sets of terms are appealing in their simplicity, they fail to tell the full story, so more precise approaches are needed. *Partial equivalence relation* (PER) models are one compelling solution which recognizes that a type tells us not only what terms it classifies but also when those terms are equal. The lesson we would like to draw from this dissertation is that orthogonality and related methods suggest yet a third perspective where we recognize types as classifying both terms *and* contexts. Our view is that this is not just a technical convenience to make proofs work, but something essential about what “types” are. It is highly suggestive that the dualization in orthogonality—that more terms means less co-terms and vice versa—corresponds so neatly to sub-typing. And, we find it even more suggestive that this notion of behavioral sub-typing is exactly what lets us solve the recursion at the heart of symmetric candidates.

Building a single, strategy-parametric proof of strong normalization allowed us to extract a proof, and so eventually even a more direct orthogonality based proof, for call-by-need as a special case. This demonstrates the utility of treating the strategy as a parameter to the reduction theory as in Downen and Ariola (2014): call-by-need, call-by-value, and call-by-name all give rise to their own separate theories, but the parametric approach allows us to abstract out what is the same in all of them.

Our emphasis on directly accounting for many strategies differs somewhat from tradition in the λ-calculus where call-by-name is emphasized above all. As we have seen, there is no single “best” strategy which encompasses all the others. Even in the pure λ-calculus, it's not always enough to think of strategies like call-by-value as just *restrictions* of the seemingly most-general strategy call-by-name, so that β-reduction only fires on values. As the study of call-by-value calculi has brought to light Herbelin and Zimmermann (2007); Sabry and Felleisen (1992), β alone isn't complete for these alternative strategies. As such, proving strong normalization only for β might not be enough; we should instead strive for more complete reduction theories and account for all their necessary rules. Our parametric mixed strategy system allows us to do just that while also allowing us to mix strategies.

It also though raises questions about relating our work back to the λ-calculus. Surely, given how present languages already have features (such as `seq` in Haskell or a `lazy` keyword in some strict languages) for mixing lazy and eager evaluation, it would be desirable to have λ-calculi which support mixing strategies akin to our multi-strategy sequent calculus. However, this is more of a challenge since when we write $v\ v'$ in λ-calculus it is not clear from the syntax what the strategy of the entire resulting expression should be even if we know the strategies for both v and v'. While in the sequent calculus we could annotate the call stack constructor $\cdot$ with a strategy, in λ-calculus it is odd to annotate an application with the strategy of the entire resulting term. Still, such annotations are certainly possible. As such, a multi-strategy λ-calculus, which could be proven strongly normalizing by compilation to our sequent calculus, is left as future work. In particular, while we expect no particular difficulties in handling a fixed set of strategies (including call-by-name, call-by-value, call-by-need, and the non-deterministic strategy) the criteria for stable and focalizing strategies seem specific to a sequent calculus and it would be interesting to see if they could be given a λ-calculus presentation.

As we saw in Chapter V the projection based approach to rewriting with extensionality does not scale beyond lazy functions. An interesting question for future work is to see if there are *other* computational interpretations of extensional axioms which can be used to address the challenge of extensional rewriting for other systems, such as the call-by-value λ-calculus or languages with additional types besides functions. At the very least the systems of Chapter III should be a reminder that finding a rewriting theory corresponding to a specific equational theory can be significantly more involved than simply orienting the equations.

The technical contributions of Chapter's III, IV, and VI are each directly relevant to compiler writers. Together they suggest the possibility of a compiler intermediate language that supports control, multiple strategies, and which has desirable properties including confluence which gives freedom in the order of rewrites applied by the compiler, strong normalization which ensures the absence of spurious infinite loops, and head reduction as a potential code generation scheme consistent with extensionality. It would be particularly interesting to see a compiler based on head reduction instead of the weak head reduction used now.

In terms of the model for strong normalization in Chapter VI, there are many avenues for future work which present themselves. As we proved in Theorem 18, our model already incorporates semantic types which we could interpret as "union" and "intersection" types for all strategies. A now classic result for the call-by-name λ-calculus is that intersection typeability characterizes the set of strongly normalizing terms (Ghilezan (1996)). However, it is also the case that even the basic typing rules for intersection and union types are not strategy agnostic in the presence of effects, with natural seeming rules sound in one strategy but not another (Davies and Pfenning (2000)). Thus, it would be interesting to try to develop general rules for intersection and union types which somehow take strategy into account, and to see if the characterization of normalization could be extended to such a system.

Our normalization proof was presented for a system featuring impredicative polymorphism and multiple kinds (for the multiple strategies). Because of how polymorphism is modeled, there should be no significant challenges in extending it to a system with function kinds, akin to F_ω (Girard (1972)). A classical sequent calculus over such a system could also be thought of as a presentation of classical higher order logic as is implemented in the Isabelle/HOL proof assistant (Nipkow, Wenzel, and Paulson (2002)) as such our proof method can be seen as answering a strategy-refined version of the statement that higher order classical sequent calculus has cut elimination. of classical mathematics, including extensionality and various choice principles, still require computational interpretations which do not follow directly from our framework and would require further work.

Going further, our sequent calculi and strong normalization proof could serve as a stepping stone to building a dependently typed proof assistant in the style of Coq (INRIA (2017)) but extended with classical logic in the universe of propositions. In present Coq, assuming axioms is unsatisfactory as one is left with "programs" which do not compute and so properties such as canonicity or decidable type checking are lost. The computational interpretation of classical logic suggests that this problem may be solvable. However, the solution is still somewhat elusive. In Coq, one can *use* a proof at the level of construction, and, in particular, use a proof of well-foundedness of a relation to justify a recursive procedure. The problem is that it is not immediately apparent how to use a classical realizer witnessing termination of a function to run that function. As such, bringing the full power of classical reasoning to the world of constructive proof assistants remains an open problem.

REFERENCES CITED

Ager, M. S., Biernacki, D., Danvy, O., & Midtgaard, J. (2003). A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th acm sigplan international conference on principles and practice of declaritive programming* (pp. 8–19). New York, NY, USA: ACM. doi: 10.1145/888251.888254

Andreoli, J.-M. (1992). Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, *2*(3), 297-347.

Ariola, Z. M., & Blom, S. (2002). Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, *117*(3), 95 - 168.

Barbanera, F., & Berardi, S. (1994). A symmetric lambda calculus for "classical" program extraction. In *Tacs '94* (pp. 495–515). London, UK, UK: Springer-Verlag.

Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam.

Biernacka, M., & Danvy, O. (2007). A concrete framework for environment machines. *ACM Transactions on Computational Logic*, *9*(1), 1-30. doi: 10.1145/1297658.1297664

Böhm, C. (1968). Alcune proprieta delle forme β-η-normali nel λ-k-calcolo. *Pubblicazioni dell'Istituto per le Applicazioni del Calcolo*, *696*.

Carraro, A., Ehrhard, T., & Salibra, A. (2012). The stack calculus. In *Workshop on logical and semantic frameworks, with applications* (p. 93-108).

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, *33*(2), 346-366. Retrieved from http://www.jstor.org/

`stable/1968337`

Church, A. (1936, April). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, *58*(2), 345–363.

Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, *5*(2), 56-68. Retrieved from `http://www.jstor.org/stable/2266170`

Cousot, P., & Cousot, R. (1979). Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, *81*(1), 43–57.

Curien, P.-L. (1988). *The $\lambda\rho$-calculus: an abstract framework for environment machines.* Ecole Normale Supérieure (Paris). Laboratoire d'Informatique.

Curien, P.-L., & Herbelin, H. (2000). The duality of computation. In *Proceedings of the international conference on functional programming* (pp. 233–243). New York, NY, USA. Retrieved from `http://doi.acm.org/10.1145/351240.351262` doi: 10.1145/351240.351262

Curien, P.-L., & Munch-Maccagnoni, G. (2010). The duality of computation under focus. *Theoretical Computer Science*, 165–181.

Danos, V., Joinet, J.-B., & Schellinx, H. (1997, 09). A new deconstructive logic: Linear logic. *J. Symbolic Logic*, *62*(3), 755–807. Retrieved from `http://projecteuclid.org/euclid.jsl/1183745297`

Danvy, O., & Nielsen, L. R. (2004). *Refocusing in reduction semantics* (Tech. Rep.). BRICS.

David, R., & Py, W. (2001, 03). $\lambda\mu$-calculus and Bohm's theorem. *J. Symbolic Logic*, *66*(1), 407–413.

Davies, R., & Pfenning, F. (2000). Intersection types and computational effects. In *Proceedings of the fifth acm sigplan international conference* *on functional programming* (pp. 198–208). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/351240.351259` doi: 10.1145/351240.351259

de Bruijn, N. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, *75*(5), 381 - 392. doi: 10.1016/1385-7258(72)90034-0

Downen, P., & Ariola, Z. M. (2014). The duality of construction. In *Proceedings of the 23rd european symposium on programming languages and systems - volume 8410.*

Downen, P., Johnson-Freyd, P., & Ariola, Z. M. (2015). Structures for structural recursion. In *Icfp '15* (pp. 127–139). New York, NY, USA: ACM.

Felleisen, M., & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, *103*(2), 235-271.

Fujita, K.-e. (2003). A sound and complete CPS-translation for $\lambda\mu$-calculus. In *Typed lambda calculi and applications* (Vol. 2701, p. 120-134). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/3-540-44904-3_9` doi: 10.1007/3-540-44904-3_9

Gentzen, G. (1935). Untersuchungen über das logische schließen. I. *Mathematische Zeitschrift*, *39*(1), 176-210.

Ghilezan, S. (1996, 01). Strong normalization and typability with intersection types. *Notre Dame J. Formal Logic*, *37*(1), 44–52. Retrieved from `https://doi.org/10.1305/ndjfl/1040067315` doi: 10.1305/ndjfl/1040067315

Girard, J. Y. (1972). *Interprtation fonctionnelle et elimination des coupures de l'arithmtique d'ordre suprieur* (These d'tat). Universit de Paris 7.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, *50*, 1-102.

Girard, J.-Y. (1991). A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, *1*(3), 255-296.

Girard, J.-Y., Taylor, P., & Lafont, Y. (1989a). *Proofs and types.* Cambridge University Press.

Girard, J.-Y., Taylor, P., & Lafont, Y. (1989b). *Proofs and types.* New York, NY, USA: Cambridge University Press.

Gordon, M. (2000). From lcf to hol: A short history. In G. Plotkin, C. Stirling, & M. Tofte (Eds.), *Proof, language, and interaction* (pp. 169–185). Cambridge, MA, USA: MIT Press. Retrieved from `http://dl.acm.org/citation.cfm?id=345868.345890`

Herbelin, H. (2005). C'est maintenant qu'on calcule : Au cœur de la dualité. In *Habilitation à diriger les reserches.*

Herbelin, H., & Ghilezan, S. (2008). An approach to call-by-name delimited continuations. In *Popl* (p. 383-394).

Herbelin, H., & Zimmermann, S. (2007). An operational account of call-by-value minimal and classical λ-calculus in "natural deduction" form. In *In pierre-louis curien, editor, 17 international conference, tlca 07, brasilia, brazil. july 2009, proceedings, volume 5608 of lecture notes in computer science* (pp. 142–156).

Hofmann, M., & Streicher, T. (2002). Completeness of continuation models for $\lambda\mu$-calculus. *Information and Computation*, *179*(2), 332 - 355. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0890540101929475` doi: http://dx.doi.org/10.1006/inco.2001.2947

Howard, W. A. (1980). The formulae-as-types notion of construction. In

J. R. Hindley & J. P. Seldin (Eds.), *To h.b. curry: Essays on combinatory logic, lambda calculus, and formalism.* Academic Press.

Hughes, J. (1989, April). Why functional programming matters. *Comput. J.*, *32*(2), 98–107. doi: 10.1093/comjnl/32.2.98

INRIA. (2017). *The coq proof assistant reference manual.* Retrieved from `https://coq.inria.fr/refman/`

Johnson-Freyd, P., Downen, P., & Ariola, Z. M. (2015). First class call stacks: Exploring head reduction. In O. Danvy & U. de'Liguoro (Eds.), *Proceedings of the workshop on continuations, woc 2016, london, uk, april 12th 2015.* (Vol. 212, pp. 18–35). Retrieved from `https://doi.org/10.4204/EPTCS.212.2` doi: 10.4204/EPTCS.212.2

Johnson-Freyd, P., Downen, P., & Ariola, Z. M. (2017). Call-by-name extensionality and confluence. *Journal of Functional Programming*, *27*, e12. Retrieved from `https://doi.org/10.1017/S095679681700003X` doi: 10.1017/S095679681700003X

Kleene, S. C. (1945, 12). On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, *10*(4), 109–124.

Kleene, S. C., & Rosser, J. B. (1935). The inconsistency of certain formal logics. *Annals of Mathematics*, *36*(3), 630-636. Retrieved from `http://www.jstor.org/stable/1968646`

Klop, J. W., & de Vrijer, R. C. (1989). Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, *80*(2), 97 - 113. Retrieved from `http://www.sciencedirect.com/science/article/pii/089054018990014X` doi: http://dx.doi.org/10.1016/0890-5401(89)90014-X

Klop, J. W., van Oostrom, V., & van Raamsdonk, F. (1993). Combinatory reduction systems: introduction and survey. *THEORETICAL COMPUTER SCIENCE*, *121*, 279–308.

Krivine, J.-L. (2007, September). A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, *20*(3), 199–207. doi: 10.1007/s10990-007-9018-9

Krivine, J.-L. (2009). Realizability in classical logic. In *Interactive models of computation and program behaviour* (Vol. 27, pp. 197–229). Société Mathématique de France.

Lafont, Y., Reus, B., & Streicher, T. (1993). *Continuation semantics or expressing implication by negation* (Tech. Rep.). Univ. München, Inst. für Informatik.

Lengrand, S., & Miquel, A. (2008). Classical Fω, orthogonality and symmetric candidates. *Annals of Pure and Applied Logic*, *153*(1), 3–20.

Levy, P. B. (2001). *Call-by-push-value* (PhD thesis). University of London.

Liskov, B. (1987). Keynote address - data abstraction and hierarchy. In *Oopsla '87*.

Marlow, S. (2002, May). *State monads don't respect the monad laws in haskell.* Haskell mailing list.

Morris, J. H., Jr. (n.d.). Types are not sets. In *Popl '73* (pp. 120–124).

Munch-Maccagnoni, G. (2009). Focalisation and classical realisability. In *Computer science logic* (pp. 409–423).

Munch-Maccagnoni, G. (2013). *Syntax and Models of a non-Associative Composition of Programs and Proofs* (Unpublished doctoral dissertation). Université Paris Diderot.

Munch-Maccagnoni, G., & Scherer, G. (2015). Polarised intermediate representation of lambda calculus with sums. In *Logic in computer science (LICS)* (p. 127-140). doi: 10.1109/LICS.2015.22

Nakazawa, K., & Nagai, T. (2014). Reduction system for extensional lambda-mu calculus. In *Rewriting and typed lambda calculi* (p. 349-363).

Nipkow, T., Wenzel, M., & Paulson, L. C. (2002). *Isabelle/hol: A proof assistant for higher-order logic.* Berlin, Heidelberg: Springer-Verlag.

Parigot, M. (1992). Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the international conference on logic programming and automated reasoning* (p. 190-201). ACM.

Pfenning, F. (2002). Logical frameworks—a brief introduction. In H. Schwichtenberg & R. Steinbrüggen (Eds.), *Proof and system-reliability* (Vol. 62, pp. 137–166). Kluwer Academic Publishers. (Lecture notes from the Marktoberdorf Summer School, July 2001)

Pitts, A. M. (2000, June). Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, *10*(3), 321–359.

Plotkin, G. D. (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, *1*(2), 125 - 159. Retrieved from `http://www.sciencedirect.com/science/article/pii/0304397575900171` doi: http://dx.doi.org/10.1016/0304-3975(75)90017-1

Polonovski, E. (2004). *Explicit substitutions, logic and normalization* (Theses, Université Paris-Diderot - Paris VII). Retrieved from `https://tel.archives-ouvertes.fr/tel-00007962`

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *Proceedings of the acm annual conference - volume 2* (pp. 717–740). New York, NY, USA: ACM. doi: 10.1023/A:1010027404223

Sabry, A., & Felleisen, M. (1992). Reasoning about programs in continuation-passing style. In *Lfp '92* (pp. 288–298). New York, NY, USA: ACM.

Sabry, A., & Felleisen, M. (1993). Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, *6*(3-4), 289-360.

Sabry, A., & Wadler, P. (1997). A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, *19*(6), 916-941.

Saurin, A. (2010, July). Typing streams in the $\Lambda\mu$-calculus. *ACM Transactions on Computational Logic*, *11*(4), 28:1–28:34. Retrieved from `http://doi.acm.org/10.1145/1805950.1805958` doi: 10.1145/1805950.1805958

Sestoft, P. (2002). Demonstrating lambda calculus reduction. In T. A. Mogensen, D. A. Schmidt, & I. H. Sudborough (Eds.), *The essence of computation* (pp. 420–435). New York, NY, USA: Springer-Verlag New York, Inc. doi: 10.1007/3-540-36377-7_20

Tait, W. W. (1967, 8). Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, *32*, 198–212.

Turing, A. M. (1937). Computability and -definability. *The Journal of Symbolic Logic*, *2*(4), 153-163. Retrieved from `http://www.jstor.org/stable/2268280`

Wadler, P. (1989). Theorems for free! In *Fpca '89* (pp. 347–359). New York, NY, USA: ACM.

Wadler, P. (2003). Call-by-value is dual to call-by-name. In *Proceedings of the eighth acm sigplan international conference on functional programming* (pp. 189–201).

Zeilberger, N. (2009). *The logical basis of evaluation order and pattern-matching* (Unpublished doctoral dissertation). Carnegie Mellon University, Pittsburgh, PA, USA.