The Logical Basis of
Evaluation Order and Pattern-Matching

Noam Zeilberger

CMU-CS-09-122
April 17, 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Abstract

An old and celebrated analogy says that writing programs is like proving theorems. This analogy has been productive in both directions, but in particular has demonstrated remarkable utility in driving progress in programming languages, for example leading towards a better understanding of concepts such as abstract data types and polymorphism. One of the best known instances of the analogy actually rises to the level of an isomorphism: between Gentzen's natural deduction and Church's lambda calculus. However, as has been recognized for a while, lambda calculus fails to capture some of the important features of modern programming languages. Notably, it does not have an inherent notion of evaluation order, needed to make sense of programs with side effects. Instead, the historical descendents of lambda calculus (languages like Lisp, ML, Haskell, etc.) impose evaluation order in an ad hoc way.

This thesis aims to give a fresh take on the proofs-as-programs analogy—one which better accounts for features of modern programming languages—by starting from a different logical foundation. Inspired by Andreoli's *focusing proofs* for linear logic, we explain how to axiomatize certain canonical forms of logical reasoning through a notion of *pattern.* Propositions come with an intrinsic *polarity,* based on whether they are defined by patterns of proof, or by patterns of refutation. Applying the analogy, we then obtain a programming language with built-in support for pattern-matching, in which evaluation order is explicitly reflected at the level of types—and hence can be controlled locally, rather than being an ad hoc, global policy decision. As we show, different forms of continuation-passing style (one of the historical tools for analyzing evaluation order) can be described in terms of different *polarizations.* This language provides an elegant, uniform account of both untyped and intrinsically-typed computation (incorporating ideas from infinitary proof theory), and additionally, can be provided an *extrinsic* type system to express and statically enforce more refined properties of programs. We conclude by using this framework to explore the theory of typing and subtyping for intersection and union types in the presence of effects, giving a simplified explanation of some of the unusual artifacts of existing systems.

לאבא ואמא

I would take these six weeks of relative solitude and give this new thing, still in a file called X, a chance to grow. If nothing came of it, I would go back to Fountain City, having wasted only a month and a half. What was a month and a half out of five years?

The new book seemed to want to take place in Pittsburgh, and thus, in my basement room, I returned to the true fountain city, the mysterious source of so many of my ideas. I didn't stop to think about what I was doing, whom it would interest, what my publisher and the critics would think of it, and, sweetest of all, I didn't give a single thought to what I was trying to say. I just wrote.

—Michael Chabon, "Diving into the Wreck", *Maps & Legends*

Contents

Acknowledgments

Foremost I would like to thank Frank Pfenning: for teaching me about logic, for asking me tough questions, and for having enormous patience for my answers. I am also grateful to Peter Lee, for his encouragement and invaluable guidance early in my career at CMU, and for giving me generous freedom to pursue my own (often half-baked) ideas. Discussions with Bob Harper have been a tremendous help and influence—I am thankful to Bob for his passion about programming languages, and for the strength of his convictions. I am also grateful to my external committee member, Paul-André Melliès, for inviting me to Paris, and for trying to convey to me some of his deep insights about polarity and proofs.

Outside of my thesis committee, I would first of all like to thank Dan Licata. I am indebted to him for his impulse to join together and dig deep into the type-theoretic trenches, and many of the ideas in this thesis were tested and refined by our collaboration. I have also learned a lot from fellow scribblers on the whiteboards of Wean (a.k.a. POP students), particularly Kaustuv Chaudhuri, Joshua Dunfield, Deepak Garg, Neel Krishnaswami, William Lovas, Sean McLaughlin, Aleks Nanevski, Jason Reed, Rob Simmons, and Kevin Watkins. And I have benefited greatly from interactions with Jeremy Avigad, Steve Awodey, David Baelde, Andrej Bauer, Lars Birkedal, Peter Hancock, Sebastian Hanowski, Stéphane Lengrand, Paul Levy, Peter Lumsdaine, John Reynolds, Anton Setzer, Chung-chieh Shan, and Matthieu Sozeau, and from some historical sleuthing on the part of Thierry Coquand, Peter Dybjer, and Peter Hancock. In addition to the help of these brilliant humans, I have received incredible insights from software proof assistants (Shalosh B. Ekhad, I hope you're listening), first using Coq (through Sean's influence), then Agda (on Dan's suggestion), and finally (in perhaps surprising chronological order) in Twelf. I am grateful to the developers of all these languages for their selfless efforts in creating these wonderful intellectual playgrounds.

Last but not least, I want to thank my family—for everything—and Pittsburgh, and all its residents and former-residents who have made my time here so rewarding.

Chapter 1

Introduction

A good analogy is like a diagonal frog.

—Kai Krause

This thesis revisits the old analogy between proving and programming: *a proof is like a program, a program is like a proof.* Perhaps the reason why this analogy—known variously as *proofs-as-programs, propositions-as-types,* or the *Curry-Howard correspondence*—has remained fresh after several decades is because it is inspiring in both directions. Computer scientists can tell themselves, "I'm not just wasting my time writing programs all day, I'm proving theorems!" And mathematicians can tell themselves, "I'm not just wasting time proving theorems all day, I'm writing programs!"

But not only in matters of self-esteem, the proofs-as-programs analogy really has demonstrated remarkable utility in driving progress in programming languages. Over the past few decades, many different ideas from logic have permeated into the field, reinterpreted from a computational perspective and collectively organized as *type theory.* In the 1980s, type theory dramatically improved the theoretical understanding of difficult language concepts such as abstract data types and polymorphism, and led directly to the development of groundbreaking new languages such as ML and Haskell. In the other direction, type theory has also been applied back towards the mechanization of mathematics, and the Curry-Howard correspondence forms the basis for successful proof assistants such as Coq and Agda. Not least, the analogy between proving and programming has the social effect of linking two different communities of researchers: although people who write/study proofs and people who write/study programs often have very different motivations, the Curry-Howard correspondence says that in some ways they are doing very similar things.

Yet, how reasonable is the analogy? For certain formal notions of "proof" and "program", it goes beyond the level of an analogy to an *isomorphism.* The best-known example is the isomorphism between Gentzen's natural deduction and Church's simply-typed lambda calculus, formalized by William Howard in 1969. Howard's observation was influential not only in providing support for the analogy, but also in elevating the status of the formalisms themselves—after all, the same mathematical structure was observed arising independently in separate contexts. But then again, did either of these separate formalisms really capture the properties of proofs or programs "as God intended"? Or at least (what may not be exactly the same thing) as they occur in practice?

In fact, it has been recognized for a long time that lambda calculus does not quite match up with even its historical descendents, languages like LISP, ML and Haskell. John Reynolds wrote in 1972,

> Purely applicative languages are often said to be based on a logical system called the lambda calculus, or even to be "syntactically sugared" versions of the lambda calculus. . . However, as we will see, although an unsugared applicative language is syntactically equivalent to the lambda calculus, there is a subtle semantic difference.

This "subtle semantic difference" Reynolds was describing is known as *evaluation order.* If all a program ever does is take an input, compute some mathematical function on it, and return an answer, then the order in which it performs different operations may be important for matters of efficiency, but not for the ultimate result. However, in most real programming languages, that is *not* all that a program can do. Different subroutines may never return an answer, looping endlessly. Or they might find something wrong with the input and raise an exception, or worse, cause a system crash; or they might print to the screen, or prompt the user for input; or read and write locations from memory or disk; or transfer control to another thread. . . . The way in which these possible effects are staged almost always affects the behavior of the program, and the programmer probably had a certain order in mind. That is why most programming languages specify a (mostly) deterministic order of evaluation. But this is something imposed on lambda-calculus in an ad hoc way, and indeed there is no canonical choice. For example, LISP and ML adopt different strategies from Haskell.

The fact that lambda-calculus has this subtle but real and important difference with modern programming languages may seem to place skepticism on the original analogy between programs and proofs. Indeed, side-effects seem to precisely reflect the ways in which real programs are *unlike* proofs. How does a proof raise an exception, or print to the screen?

But evaluation order is not the *only* way in which the lambda-calculus differs from modern functional programming language. Church's original formulation was fundamentally about *functions.* "Everything is a function", as the slogan goes. But while it is possible (as Church showed) to encode any datatype in the pure lambda-calculus, and while these encodings are elegant and insightful—they aren't particularly natural. Hardly anyone would think of using, say, the Church-encoding of binary trees on their first pass at writing a search routine. (Perhaps they will on a second or third pass, if the routine has some subtle control flow they can't capture with a more basic data structure.) One of the simple but enormously practical features of languages like ML and Haskell is the ability to use algebraic data types and define functions on them by *pattern-matching.* This feature often enables programmers to understand the behavior of their subroutines using completely first-order equational reasoning, *without* necessarily having to reason about higher-order functions (and the potential side-effects those functions may incur).

In contrast to evaluation order and side-effects, pattern-matching is something borrowed directly from mathematical practice, where it is used both as a notation for defining functions, and as a notation for writing proofs by case-analysis. And so it reveals a deficiency in the *other* half of the Curry-Howard isomorphism: the fact that Gentzen's natural deduction lacks pattern-matching facilities makes it not so natural. Where an ordinary mathematical proof might just list a set of different cases without extra justification, in natural deduction (and its close relative, sequent calculus) the single case-analysis is converted into a series of steps, breaking down each binary conjunction and disjunction individually, ultimately arriving at the same thing.

It may be unsurprising that performing all of these microscopic steps is not just tedious, but also inefficient if we want to *discover* proofs. The proof search community has therefore devised many different strategies for making larger (but still sound) inferences when looking for proofs, to reduce this inefficiency. A fundamental breakthrough was made by Andreoli [1992], a proof search strategy he called *focusing* for Girard's linear logic [1987]. Andreoli's observation was that in the context of sequent calculus proof search, the linear logic connectives exhibit a natural duality. Depending on which side of the sequent a formula $A \clubsuit B$ is placed (i.e., whether it is assumed to be true or false), the connective $\clubsuit$ will either behave "synchronously" or "asynchronously". Precisely half of the connectives of linear logic—$\otimes$, $\oplus$, 1, 0, and !, which Andreoli called synchronous—are invertible on the left side of the sequent (i.e., they can be decomposed eagerly), and non-invertible on the right. Whereas for the other half—$\parr$, $\&$, $\bot$, $\top$, and ?, which Andreoli called asynchronous—the opposite is true. Andreoli used this observation to build an efficient proof search strategy for linear logic, alternating between dual phases he called *inversion* and *focus*. Girard [1993] followed up on Andreoli's idea, and termed this choice of bias towards the right or to the left the *polarity* of a connective. Girard's insight was that polarity was a very widespread phenomenon, observable not only in linear logic, but also in intuitionistic and constructive versions of classical logic. While the connectives of classical logic appear to have no bias towards proof or refutation, they can be explicitly *polarized*, positively or negatively, which endows them with different constructive content.

The central claim of this thesis is that

focusing proofs provide a logical account of evaluation order and pattern-matching

This claim obviously has at least two parts, one about evaluation order and one about pattern-matching. The first claim can be stated more specifically:

evaluation order is determined by polarity

Because polarity is a property of particular connectives, rather than of an entire logic, this in turn has the implication that *one language can (should) mix different evaluation strategies, by reflecting them at the level of types.* The second claim, more specifically, is that

pattern-matching is justified by polarity

which in turn has an implication that *pattern-matching is not just "syntactic sugar"—it can (should) be dealt with directly in type theory.* Most of these subclaims have already appeared in some form or another prior to this work. But the point is that pattern-matching and evaluation order really are two sides of the same coin. And focusing, I will argue, provides the right concepts for treating evaluation order and pattern-matching in a unified way, and understanding the duality between them.

While focusing was discovered in the early 1990s, many of the ideas behind it are much older. One illustration is the various attempts made in the 1970s—by Dag Prawitz, Michael Dummett, Per Martin-Löf, and others—to provide philosophical interpretations that would in some sense "justify" the logical laws. In particular, Dummett explored the idea that the laws of natural deduction could be justified by alternate "verificationist" or "pragmatist meaning-theories".[1] The idea, essentially, was that either the introduction rules or the elimination rules for a connective could be taken as constituting its *definition*. Taking the first view—the verificationist one—an

[1]In the 1976 William James Lectures, later published as *The Logical Basis of Metaphysics* [Dummett, 1991].

elimination rule can then be justified, by showing that a verification (i.e., introduction) of its premises already implies its conclusion. But taking the pragmatist view, it is the introduction rules which are justified from the elimination rules, by showing that any consequence (i.e., elimination) of the conclusion could have already been derived from the premises.

Dummett's analysis can be displayed graphically as a sort of Aristotelian "square of opposition" of assertions about a proposition A:

$$\begin{tikzcd}
\boxed{\text{direct proof of } A} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \boxed{\text{justified consequence of } A} \arrow[d, leftrightarrow] \arrow[dl, leftrightarrow] \\
\boxed{\text{justified proof of } A} \arrow[r, leftrightarrow] & \boxed{\text{direct consequence of } A}
\end{tikzcd}$$

The verificationist interpretation of A deals in the top half of the square: one explains the meaning of A in terms of its direct proofs, and then one is justified in deriving consequences from A by case-analysis over its direct proofs. The pragmatist interpretation of A deals in the bottom half.

So Dummett's investigation already contained a hint of polarity, and of the duality between focus and inversion. But Dummett insisted on a requirement of *harmony* between the two approaches. What was missing from his account was the possibility of simply accepting *diversity*—that the different meaning-theories actually define *different* connectives. That was the insight of linear logic, which showed we could distinguish a plethora of connectives—two conjunctions ($\otimes$ and $\&$), two disjunctions ($\oplus$ and $\invamp$), etc.—and relate them by modalities. And it accords with our operational intuitions from programming languages, that for example strict products and lazy products really are different things, and we would sometimes like to be able to speak about both within the same language. Our aim is thus to build a *polarized type theory*, where we can say what we mean.

The framework that is probably most closely related to the one developed here is Paul Levy's call-by-push-value [Levy, 2001, 2004]. CBPV also maintains a distinction between two different kinds of types, which Levy calls *value types* and *computation types*, and these correspond, more or less, to positive types and negative types as I use them. However, the language I present here was developed independently of CBPV. The CBPV paradigm arose strictly out of semantic concerns, by trying to unify denotational models of call-by-value and call-by-name, without any dependence on or connection to proof theory. In contrast, the framework developed here arose strictly out of proof-theoretic concerns, by trying to understand the effect of evaluation order on subtyping. The two alternate accounts thus give each other independent confirmation. On the other hand, I also think that the special emphasis on proof theory in this thesis helps us to gain some new ground not already covered by CBPV, particularly with respect to refinement types (e.g., intersection and union types) and subtyping.

The remaining chapters of the thesis are organized as follows:

Chapter 2 (Canonical derivations). We introduce the underlying logical objects which will later be given a computational interpretation. We develop a generic account of proof and refutation for polarized propositions—based on a primitive notion of *pattern*—in the form of an iterated inductive definition. We show that these derivations satisfy suitable principles of identity and composition, and describe how they induce different notions of entailment.

Chapter 3 (Focusing proofs and double-negation translations). We explain in what sense the canonical derivations of Chapter 2 correspond to Andreoli's focusing proofs. We exploit this correspondence to better understand the relationship between classical logic, polarized logic, and minimal logic, showing how to reconstruct different double-negation translations of classical into minimal logic (Glivenko, Gödel-Gentzen, etc.) by factoring them through polarized logic.

Chapter 4 (Proof as programs). We reinterpret the proof objects of Chapter 2 through the lens of Curry-Howard, and show that this gives rise to phenomena familiar from the operational semantics of programming languages, notably pattern-matching and continuation-passing-style. We give an *intrinsic* definition of a programming language—terms are equated with logical derivations—but develop a *type-free* notation for terms, with an equational theory and an environment-based operational semantics. We define observation equivalence, and show that it coincides with syntactic equality in the presence of sufficient effects. We study programming with mixed polarity types, and explain how this gives a rich language for describing mixed evaluation order. Finally, we convert the results about double-negation translations in Chapter 3 to the computational setting, and show how to reconstruct different CPS translations of λ-calculus (call-by-value, call-by-name, etc.) by way of polarization.

Chapter 5 (Concrete notations for abstract derivations). We describe embeddings of the language $\mathcal{L}^+$ defined in Chapter 4 into two existing logical frameworks based on dependent type theory, Agda and Twelf. The two frameworks have different proof-theoretic strength, which helps us to better understand the features of $\mathcal{L}^+$ by compiling them down to lower-level primitives. In particular, the Twelf embedding employs a novel use of defunctionalization to compile pattern-matching.

Chapter 6 (Refinement types and completeness of subtyping). We develop an *extrinsic* view of polarized type theory, allowing more precise properties of terms to be specified through more refined types. One of our motivations is to better understand operationally-sensitive artifacts in historical type systems, giving an explanation for why, e.g., intersection types require a value restriction in ML. We give two interpretations of subtyping—one demanding an explicit witness to the safety of a subtyping relationship, one asking only for the absence of counterexamples—and study the relationship between them. We show that the two forms of subtyping coincide in the presence of sufficient effects.

Chapter 7 (Conclusion). We summarize the contributions of the thesis, and discuss some paths for future work.

Chapter 2

Canonical derivations

Once we have understood how to discover individual patterns which are alive, we may then make a language for ourselves, for any building task we face.

—Christopher Alexander, *The Timeless Way of Building*

In this chapter, I introduce the basic proof objects which will later be given a proofs-as-programs interpretation. I call them *canonical derivations,* since they enumerate canonical forms of proof and refutation. As I will explain in the next chapter, canonical derivations can also be seen as an alternate presentation of Andreoli's focusing proofs, for classical propositional logic. However, I would rather not start out by explaining them that way, because the connection to classical logic in the classical sense is actually only indirect, a kind of double-negation translation—and in fact, canonical derivations have a stronger connection to minimal (and "co-minimal") logic. My primary aim in this chapter is to describe the structure of canonical derivations, and only secondarily to study an entailment relation induced by them (we will see in §2.3.3 that there are actually two distinct but equally legitimate ways of defining entailment).

Our subject is *polarized logic.* That is, logic in which every proposition has a definite polarity, positive or negative. As I alluded to in the Introduction, one way of understanding polarity—in the spirit of the "meaning explanations" of the '70s put forth by Prawitz, Dummett, and Martin-Löf—is that positive propositions are "defined" by their form of introduction, and negative propositions by their form of elimination. In Dummett's sense, positive propositions have a "verificationist meaning-theory", and negative propositions a "pragmatist meaning-theory". The primary contribution of this chapter is a system of logical inference that makes this intuition formal through a notion of *pattern.* A pattern is basically a derivation with holes. The key intuition, which is actually a formal property of polarized logic, is that:

positive connectives are defined by their proof patterns

negative connectives are defined by their refutation patterns

These statements are meant literally. That is, we will define canonical derivations in two stages: first we define individual connectives by describing their proof patterns (for positive connectives) or refutation patterns (for negative connectives), and then we give generic rules for full proof and refutation, described in terms of patterns.

To get this project off the ground, we will have to pay careful attention to the distinction between *propositions* and *judgments* [Martin-Löf, 1996]. To be clear, we are interested in two different forms of judgment about a polarized proposition, on equal footing: both its assertion, A *true*, and its denial, A *false*. For notational concision, we will simply write the former as A and the latter as $\bullet A$, but it is important in mind to keep in mind that these denote judgments, not propositions.

We will treat proof and refutation in a completely symmetric way—or rather, we will treat them asymmetrically in one way for positive polarity, and then treat them asymmetrically in precisely the opposite way for negative polarity. This is a matter of expediency, because it lets us understand the basic idea of polarity without being overburdened by too many distinctions. On the other hand, it is a bit simplistic. General intuitionistic implication, for example, cannot be expressed in this framework—it *is* naturally defined as a negative connective, but in terms of patterns for deriving *arbitrary* consequences, rather than in terms of refutation patterns. However, I think this framework has *enough* asymmetry to be interesting, and that it serves as a good foundation from which to study more sophisticated uses of polarity, by careful generalization and symmetry-breaking.

2.1 A proof-biased logic

To start, in this section I will restrict to only positive connectives, showing how to define patterns, proofs and refutations, and how to establish the identity and composition principles. After this initial setup, identifying a negative fragment—and then generalizing to a unified polarized logic—will be relatively easy. As explained in the Introduction, I will need different symbols to distinguish the polarized connectives—the notation below is mostly borrowed from linear logic.

2.1.1 Refutation frames, proof patterns, connectives

In this section I use the letters A, B, C to range over positive propositions.

Definition 2.1.1 (Frames of refutation holes)**.** *A* **frame** Δ *is a list* $\bullet A_1, \ldots, \bullet A_n$ *of* **refutation holes***. We write* $\cdot$ *for the empty frame, and* Δ_1, Δ_2 *for the concatenation of two frames. We write* $\Delta \in \Delta'$ *for list containment, i.e., the following inductively defined relation:*

$$\frac{}{\Delta \in \Delta} \qquad \frac{\Delta \in \Delta_1}{\Delta \in \Delta_1, \Delta_2} \quad \frac{\Delta \in \Delta_2}{\Delta \in \Delta_1, \Delta_2}$$

Now, we inductively define a judgment $\Delta \Vdash B$ relating frames and positive propositions. Intuitively, $\Delta \Vdash B$ says that it is possible to derive B directly from the premises Δ. Or in other words, a derivation of $\Delta \Vdash B$ gives the *outline* of a proof, leaving holes for refutations. For example, we define conjunction and truth (0-ary conjunction) as follows:

$$\frac{}{\cdot \Vdash 1} \qquad \frac{\Delta_1 \Vdash A \quad \Delta_2 \Vdash B}{\Delta_1, \Delta_2 \Vdash A \otimes B}$$

Intuitively, these definitions express that a proof of truth requires no premises, while a proof of the conjunction $A \otimes B$ requires proofs of A and B, and combines their respective premises. Likewise, we define disjunction and falsehood (0-ary disjunction) with the following rules:

$$\text{(no rule for 0)} \qquad \frac{\Delta \Vdash A}{\Delta \Vdash A \oplus B} \quad \frac{\Delta \Vdash B}{\Delta \Vdash A \oplus B}$$

$$\frac{}{\cdot \Vdash 1} \qquad \frac{\Delta_1 \Vdash A \quad \Delta_2 \Vdash B}{\Delta_1, \Delta_2 \Vdash A \otimes B}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta \Vdash A}{\Delta \Vdash A \oplus B} \quad \frac{\Delta \Vdash B}{\Delta \Vdash A \oplus B}$$

$$\frac{}{\bullet A \Vdash \overset{+}{\neg} A}$$

Figure 2.1: Definition of some positive connectives by proof patterns

Intuitively, these definitions express that there is no proof of falsehood, while a proof of the disjunction $A \oplus B$ requires either a proof of A or a proof of B. Finally, negation is defined as follows:

$$\frac{}{\bullet A \Vdash \overset{+}{\neg} A}$$

This definition obviously doesn't express very much: just that a proof of $\overset{+}{\neg} A$ requires a refutation of A. We use the notation $\overset{+}{\neg}$ to mark this negation as having positive polarity, as opposed to the negative polarity negation $\overset{-}{\neg}$ defined further below. When the polarity is clear from context, however, we sometimes simply write $\neg A$.

Again, I take the above rules as literally a *definition* of the connectives. For quick reference, this definition is displayed in Figure 2.1.

Definition 2.1.2 (Proof patterns). *A derivation of $\Delta \Vdash A$ is called a* **proof pattern**, *or more specifically an A-pattern. We refer to the frame Δ as the frame of the pattern. The set of all A-patterns is called the* **support** *of A.*

Intuitively, the support of A describes all possible ways of proving A. From now on, I won't speak of individual connectives except in examples, instead dealing generically with refutation frames and proof patterns.

Example 2.1.3. Let $C = \neg A \otimes (\neg B_1 \oplus \neg B_2)$. There are exactly two patterns in the support of C:

$$\frac{\bullet A \Vdash \neg A \quad \dfrac{\bullet B_1 \Vdash \neg B_1}{\bullet B_1 \Vdash \neg B_1 \oplus \neg B_2}}{\bullet A, \bullet B_1 \Vdash C} \qquad \frac{\bullet A \Vdash \neg A \quad \dfrac{\bullet B_2 \Vdash \neg B_2}{\bullet B_2 \Vdash \neg B_1 \oplus \neg B_2}}{\bullet A, \bullet B_2 \Vdash C}$$

These two patterns have frames $\bullet A, \bullet B_1$ and $\bullet A, \bullet B_2$, respectively. ■

2.1.2 The definition ordering

The notion of pattern induces a more abstract definition of *subformula* (cf. [Takeuti, 1975]).

Definition 2.1.4 (Definition ordering). *We write $\Delta \prec B$ between a frame and a proposition if there is some proof pattern $\Delta \Vdash B$, and write $A \prec \Delta$ between a proposition and a frame if there is some refutation hole $\bullet A \in \Delta$. The* **definition ordering** *is the transitive closure of $\prec$.*

Definition 2.1.5 (Definition tree, ancestors)**.** *The restriction of $\prec$ below A (written $\prec_A$) is called the* **definition tree** *of A. Conceptually, the tree grows backwards, "towards a simpler time", and we call the (non-root) elements of the tree the* **ancestors** *of A. Similarly, we write $\prec_\Delta$ for the restriction of $\prec$ below Δ, and refer to the elements of the tree as Δ's ancestors.*

We can think of the ancestors of A as its "abstract subformulas". For example, proposition $C = \neg A \otimes (\neg B_1 \oplus \neg B_2)$ has abstract subformulas A, B_1, and B_2. The concrete syntactic subformulas $\neg A$, $\neg B_1 \oplus \neg B_2$, etc., are *not* ancestors of C by this definition.

Proposition 2.1.6. *For any proposition A (or frame Δ) built out of a finite combination of the connectives $1, 0, \otimes, \oplus, \overset{+}{\neg}$, the definition tree $\prec_A$ ($\prec_\Delta$) is well-founded.*

Proof. Every negation in A marks a branch of the definition tree. Indeed, if A is $\overset{+}{\neg}$-free then it has no ancestors (compare this to the usual definition of subformula). □

2.1.3 Proofs and refutations

Suppose we have defined some positive connectives and their proof patterns. Now we can explain how to build actual proofs and refutations. In addition to the judgments A and $\bullet A$, we use the auxiliary judgments Δ and $\#$. Intuitively, Δ asserts the conjunction of all its hypotheses, while $\#$ asserts contradiction. Obviously for contradiction, but also for the other judgments, we are mainly interested in reasoning *relative to a context.*

Definition 2.1.7 (Contexts)**.** *A* **context** *Γ is a list of frames. We define the containment relationship between frames and contexts as follows:*

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

Since a frame is also a list (of refutation hypotheses) we can always flatten a context and view it as a frame—but it will nonetheless be useful to have a conceptual distinction between frames and contexts. We write $\Gamma \vdash J$ to assert a judgment J (any of A, $\bullet A$, Δ, or $\#$) relative to context Γ. We now explain the formal meaning of these judgments with inference rules.

A: *To prove A, we must choose some A-proof pattern, and derive its frame.*

$$\frac{\Delta \Vdash A \quad \Gamma \vdash \Delta}{\Gamma \vdash A}$$

In other words, if a proof pattern describes a proof with holes, to build an actual proof we must fill these holes. Observe that because there can be many possible A-patterns, there can be many possible ways to apply this rule, and proving A requires making a choice.

$\bullet A$: *To refute A, we must examine every possible A-proof pattern, and show how to derive a contradiction from its frame.*

$$\frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A}$$

The arrow in the premise means that to every possible derivation of the left-hand side ($\Delta \Vdash A$), we must give a derivation of the right-hand side ($\Gamma, \Delta \vdash \#$). Note that there is only one possible way to apply this rule, given the definition of A. Since the propositions defined in §2.1.1 all have finite support, we can view this rule as having finitely many premises, one for each A-pattern. In general, though, we would like to give this rule a more open-ended interpretation, and view the premise as literally demanding a *function* from A-patterns to contradictions, rather than a simple list of contradictions.[1] We will say much more about this in Chapters 4 and 5.

Δ: *To assert Δ, we must supply refutations for all its holes.*

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)}$$

These rules tell us explicitly how to unravel a frame, but really the order is arbitrary because frames are associative. Formally, we have the following:

Observation 2.1.8. *Any derivation of $\Gamma \vdash \Delta$ determines a map from refutation holes $\bullet A \in \Delta$ to derivations $\Gamma \vdash \bullet A$, and conversely, given such a map we can build a derivation of $\Gamma \vdash \Delta$. In other words, the two rules above are interchangeable with the following:*

$$\frac{\bullet A \in \Delta \quad \longrightarrow \quad \Gamma \vdash \bullet A}{\Gamma \vdash \Delta}$$

$\#$: *To derive contradiction, we must find a proposition assumed to be false, and prove it.*

$$\frac{\bullet A \in \Gamma \quad \Gamma \vdash A}{\Gamma \vdash \#}$$

Again, there can be many possible ways of applying this rule, for every refutation hole in the context.

Example 2.1.9. Let $C = \neg A \otimes (\neg B_1 \oplus \neg B_2)$ as in Example 2.1.3. By applying the two possible C-patterns and instantiating the general rules, we can build two derived rules for proving C, one for each C-pattern:

$$\frac{\Gamma \vdash \bullet A \quad \Gamma \vdash \bullet B_1}{\Gamma \vdash \neg A \otimes (\neg B_1 \oplus \neg B_2)} \qquad \frac{\Gamma \vdash \bullet A \quad \Gamma \vdash \bullet B_2}{\Gamma \vdash \neg A \otimes (\neg B_1 \oplus \neg B_2)}$$

■

Example 2.1.10. Let C be as above. The derived rule for refuting C has two premises:

$$\frac{\Gamma, \bullet A, \bullet B_1 \vdash \# \quad \Gamma, \bullet A, \bullet B_2 \vdash \#}{\Gamma \vdash \bullet \neg A \otimes (\neg B_1 \oplus \neg B_2)}$$

■

We take the above rules to be *canonical*, in the sense that they enumerate canonical forms of proof, refutation, etc., guided completely by the definition ordering. For this reason, we explicitly omit rules such as:

[1] Our notation is borrowed from Martin-Löf's for the theory of iterated inductive definitions. We should note that this inductive definition really is iterated in an essential way: the reason it makes sense to quantify over proof patterns is because they have already been given an inductive definition, prior to proofs. The reader familiar with the work of Buchholz et al. [1981] might notice that the refutation rule bears a very close formal resemblance to the so-called Ω-rule.

Frames	Δ	::=	$\bullet A \mid \cdot \mid (\Delta_1, \Delta_2)$
Contexts	Γ	::=	$\cdot \mid \Gamma, \Delta$

$$\frac{\Delta \Vdash A \quad \Gamma \vdash \Delta}{\Gamma \vdash A} \qquad \frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A}$$

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)} \qquad \frac{\bullet A \in \Gamma \quad \Gamma \vdash A}{\Gamma \vdash \#}$$

Figure 2.2: Canonical derivations for positive propositions

$$\frac{\Gamma \vdash \bullet A \quad \Gamma \vdash A}{\Gamma \vdash \#}$$

which would allow deriving contradiction by picking some arbitrary proposition, and showing that it has both a proof and a refutation. Instead, we will show below that this and similar rules are *admissible:* if $\Gamma \vdash \bullet A$ and $\Gamma \vdash A$ then there is a canonical derivation of $\Gamma \vdash \#$ (i.e., one that begins by finding some $\bullet B \in \Gamma$ and showing $\Gamma \vdash B$). We summarize the definition of canonical derivations in Figure 2.2.

2.1.4 Identity and composition

How do we know that canonical derivations are a reasonable notion? One sanity check is that they satisfy identity and composition principles. In general, an identity principle for a hypothetical judgment allows us to derive a corresponding conclusion for any assumption. Since a context Γ can be seen both as a collection of refutation holes $\bullet A$ or of arbitrary frames Δ, we conceptually distinguish two identity principles:

Identity (refutation)**.** *If* $\bullet A \in \Gamma$ *then* $\Gamma \vdash \bullet A$

Identity (frame)**.** *If* $\Delta \in \Gamma$ *then* $\Gamma \vdash \Delta$

We say that these are the identity principles respectively *on* A and *on* Δ. To demonstrate their validity, we give a pair of mutually recursive derivations. From $\bullet A \in \Gamma$, we derive $\Gamma \vdash \bullet A$ as follows:

$$\frac{\Delta \Vdash A \quad \longrightarrow \quad \dfrac{\bullet A \in \Gamma, \Delta \qquad \dfrac{\Delta \Vdash A \quad \overset{\vdots}{\Gamma, \Delta \vdash \Delta}}{\Gamma, \Delta \vdash A}}{\Gamma, \Delta \vdash \#}}{\Gamma \vdash \bullet A}$$

where in the last step (from the bottom) we are appealing to frame identity. Frame identity is trivial, since it just expands into a list of refutation identities:

$$\overset{\vdots}{\Gamma \vdash \bullet A} \qquad \frac{}{\Gamma \vdash \cdot} \qquad \frac{\overset{\vdots}{\Gamma \vdash \Delta_1} \quad \overset{\vdots}{\Gamma \vdash \Delta_2}}{\Gamma \vdash \Delta_1, \Delta_2}$$

These derivations use a few trivial properties of the containment relation, namely that $\bullet A \in \Gamma$ implies $\bullet A \in \Gamma, \Delta$, and that $\Delta_1, \Delta_2 \in \Gamma$ implies $\Delta_1 \in \Gamma$. Otherwise, though, coming up with them was a mechanical exercise—these are really the only possible generic derivations, given the rules of Figure 2.2. But do they make sense as derivations? Are they well-founded? The answer depends on the definition ordering.

Theorem 2.1.11. *The identities on A and Δ are well-founded just in case $\prec_A$ and $\prec_\Delta$ are well-founded.*

Proof. The recursive calls between the two derivations precisely mirror the definition ordering. □

As we saw in Proposition 2.1.6, the definition ordering is always well-founded if we consider only the boolean logical connectives. Later, when we study arbitrary recursive types, this will no longer be the case, and in order to make sense of the identity principles we will have to move to a *coinductive* interpretation of derivations.

In general, a composition principle is a way of combining two related inferences into a single inference. We have two ways of composing canonical derivations (recall that J ranges over judgments A, $\bullet A$, Δ', or $\#$):

Composition (reduction). *If* $\Gamma \vdash A$ *and* $\Gamma \vdash \bullet A$ *then* $\Gamma \vdash \#$

Composition (substitution). *If* $\Gamma \vdash \Delta$ *and* $\Gamma(\Delta) \vdash J$ *then* $\Gamma \vdash J$

In the substitution principle, the standard notation $\Gamma(\Delta)$ indicates a context with Δ plugged somewhere into Γ. To be more explicit, it says that Γ can be rewritten as the concatenation of two contexts $\Gamma = \Gamma_1, \Gamma_2$, and that $\Gamma(\Delta) = \Gamma_1, \Delta, \Gamma_2$.

We say that the these are the composition principles *on* A and *on* Δ, respectively. Again, we illustrate the validity of the composition principles with a pair of mutually recursive definitions. Note that we require an additional principle, that it is always possible to weaken a canonical derivation with an additional frame.

Weakening. *If* $\Gamma \vdash J$ *then* $\Gamma(\Delta) \vdash J$

Proof. Immediate by induction on the derivation of $\Gamma \vdash J$, using the properties of the containment relation. □

Now, suppose we have canonical derivations of $\Gamma \vdash A$ and $\Gamma \vdash \bullet A$. These *must* have the following form:

$$\frac{\Delta \Vdash A \quad \Gamma \vdash \Delta}{\Gamma \vdash A} \qquad \frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A}$$

By plugging in the A-pattern from the proof of A into the premise of the refutation of A, we get a derivation of $\Gamma, \Delta \vdash \#$. Then we derive $\Gamma \vdash \#$ by composing with the derivation of $\Gamma \vdash \Delta$.

Suppose we have derivations of $\Gamma \vdash \Delta$ and $\Gamma(\Delta) \vdash J$. We consider the possible shapes of the latter:

$$\frac{\Delta' \Vdash A \quad \Gamma(\Delta) \vdash \Delta'}{\Gamma(\Delta) \vdash A} \qquad \frac{\Delta' \Vdash A \quad \longrightarrow \quad \Gamma(\Delta), \Delta' \vdash \#}{\Gamma(\Delta) \vdash \bullet A}$$

$$\frac{}{\Gamma(\Delta) \vdash \cdot} \qquad \frac{\Gamma(\Delta) \vdash \Delta_1 \quad \Gamma(\Delta) \vdash \Delta_2}{\Gamma(\Delta) \vdash (\Delta_1, \Delta_2)} \qquad \frac{\bullet A \in \Gamma(\Delta) \quad \Gamma(\Delta) \vdash A}{\Gamma(\Delta) \vdash \#}$$

Most of these cases are trivial: we just apply the composition principle recursively to the sub-derivations, and re-apply the rule (in the case of the refutation rule, we must also invoke weakening on the first derivation). The only interesting case is the contradiction rule, when we have $\bullet A \in \Gamma(\Delta)$ by virtue of $\bullet A \in \Delta$. Since $\Gamma \vdash \Delta$ and $\bullet A \in \Delta$ implies $\Gamma \vdash \bullet A$ (Observation 2.1.8), after performing substitution on $\Gamma(\Delta) \vdash A$ to obtain $\Gamma \vdash A$, we can apply reduction to obtain $\Gamma \vdash \#$.

Again, we must verify that these definitions are well-founded.

Theorem 2.1.12. *The compositions on A and Δ are well-founded just in case $\prec_A$ and $\prec_\Delta$ are well-founded.*

Proof. As with the identity principle, the recursive calls between the two forms of composition precisely mirror the definition ordering. □

2.1.5 Complex frames

We have seen that the identity and composition principles can be justified on a generic basis, by taking advantage of the uniform definition of connectives by patterns. This is interesting from a philosophical perspective, in the sense that it gives a "justification" of the logical laws in the style of Prawitz/Dummett/Martin-Löf, exploiting a general inversion principle for positive propositions. It is also interesting from a purely proof-theoretic perspective, because as we will see in the next chapter, the composition of two canonical derivations corresponds exactly to the *cut* of two sequent calculus proofs, so our generic composition theorem is also a generic cut-elimination theorem. Although attempts at giving generic criteria for cut-elimination have been made before,[2] it is still a common belief that proving cut-elimination theorems requires a tedious case analysis on all possible matchings of the rules for the different connectives—which we entirely avoided. And as we will see in Chapter 4, this pattern-based justification of identity and cut also has a deep computational significance, giving us, for example, a generic proof of type safety for a programming language.

So it is nice that we have available this sort of generic justification. On the other hand, when presenting a particular derivation, there are times when we *don't care* about the justification. Suppose, e.g., that we want to derive $\bullet A \oplus \overset{+}{\neg} A \vdash \#$. A canonical derivation proceeds like so:

$$\dfrac{\dfrac{\Delta \Vdash A \quad \longrightarrow \quad \dfrac{\dfrac{\Delta \Vdash A \qquad \overset{\textit{Id}}{\bullet A \oplus \neg A, \Delta \vdash \Delta}}{\bullet A \oplus \neg A, \Delta \vdash A \oplus \neg A}\;*_1}{\bullet A \oplus \neg A, \Delta \vdash \#}}{\bullet A \oplus \neg A \vdash A \oplus \neg A}\;*_2}{\bullet A \oplus \neg A \vdash \#}$$

where *Id* marks the (frame) identity principle, and $*_1$ and $*_2$ indicate the two derived rules for proving $A \oplus \neg A$:

$$\dfrac{\Delta \Vdash A \quad \Gamma \vdash \Delta}{\Gamma \vdash A \oplus \neg A}\;*_1 \qquad\qquad \dfrac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A \oplus \neg A}\;*_2$$

[2]See in particular the recent work of Ciabattoni and Terui [2006], which gives a survey of prior work.

The $*_2$ rule quantifies over all patterns $\Delta \Vdash A$, yet, in the derivation of $\bullet A \oplus \neg A \vdash \#$, we don't actually perform any analysis of these patterns, simply reusing them to build a proof of A. It might be said, then, that including the patterns explicitly in the derivation is unnecessary notational overhead.

Suppose that we could instead simply place a hypothesis A in the context to stand abstractly for this quantification over patterns. Then we might give a notationally friendlier presentation of the same canonical derivation as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{\overset{Id}{\bullet A \oplus \neg A, A \vdash A}}{\bullet A \oplus \neg A, A \vdash A \oplus \neg A}\,*_1'}{\bullet A \oplus \neg A, A \vdash \#}}{\bullet A \oplus \neg A \vdash A \oplus \neg A}\,*_2'}{\bullet A \oplus \neg A \vdash \#}$$

Having this shorthand notation will be very convenient as we work with canonical derivations, so let us introduce it formally.

Definition 2.1.13 (Complex frames)**.** *A* **complex frame** *can contain* **complex proof hypotheses** *A in addition to refutation holes $\bullet A$. We distinguish frames that don't contain any complex hypotheses as* **simple***.*

We call these frames complex because they can always be decomposed into simple frames. The only way to use a complex hypothesis inside a frame is to perform a case distinction on patterns:

$$\frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(A) \vdash J}$$

For example, we can apply this rule to derive the identity principle for complex proof variables:

$$\cfrac{\Delta \Vdash A \quad \longrightarrow \quad \cfrac{\Delta \Vdash A \quad \overset{Id}{\Gamma(\Delta) \vdash \Delta}}{\Gamma(\Delta) \vdash A}}{\Gamma(A) \vdash A}$$

This might seem backwards—isn't the point of of complex hypotheses that we *don't* have to analyze them? Formally, what we have to observe is that the above rule is *invertible,* i.e., its conclusion implies its premise.

Proposition 2.1.14 (Pattern substitution)**.** *If $\Gamma(A) \vdash J$ and $\Delta \Vdash A$ then $\Gamma(\Delta) \vdash J$.*

Proof. Trivial, by walking through the canonical derivation to find the place, if any, where the complex hypothesis is used, and substituting the pattern into the premise. ☐

So, we *are* justified in reading the above rule as bidirectional,

$$\frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\overline{\Gamma(A) \vdash J}}$$

Frames $\quad \Delta \;::=\; \cdots \mid A$

$$\frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(A) \vdash J}$$

Figure 2.3: Complex proof hypotheses

but it is important to understand that we are *not* actually adding a new rule going in the upside-down direction. When presenting a canonical derivation, we are permitted to introduce new complex variables precisely because they can be analyzed away—just as we are justified in using the identity and composition principles because they can be eliminated. Note that we do have to verify that the composition principles still hold in the presence of complex hypotheses, but this is trivial, because we can always apply pattern substitution to bring the case analysis to the front and compose simple subderivations.

Again, let's look at the second, prettier derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{Id}{\bullet A \oplus \neg A, A \vdash A}}{\bullet A \oplus \neg A, A \vdash A \oplus \neg A}\,*_1'}{\bullet A \oplus \neg A, A \vdash \#}}{\bullet A \oplus \neg A \vdash A \oplus \neg A}\,*_2'}{\bullet A \oplus \neg A \vdash \#}$$

Step $*_2'$ in the derivation is an *admissible* step, rather than a canonical rule. Step $*_1'$ is likewise only an admissible step (because the derived rule $*_1$ actually requires us to go all the way down to a pattern for A).

Finally, let us mention one other way of understanding complex hypotheses. When describing the canonical rule of refutation in §2.1.3, we deliberately left open-ended the ways in which functions from proof patterns to contradictions could be constructed. By introducing complex hypotheses, we are making explicit a particular form of construction, whereby functions are simply defined by substitution, rather than case analysis. Because this is such a ubiquitous form of definition, it seems worthwhile to treat it explicitly.

2.2 A refutation-biased logic

We repeat the development of §2.1, but with everything reversed.

2.2.1 Proof frames, refutation patterns, definition ordering

In this section I use the letters A, B, C to range over *negative* propositions.

Definition 2.2.1 (Frames of proof holes)**.** *Frames are now taken to be lists of hypotheses* $A_1, \ldots, A_n$*. The* $A_i \in \Delta$ *are called* **proof holes***.*

Negative connectives are defined by the judgment $\Delta \Vdash \bullet A$, which asserts that we can build a refutation of A leaving holes for premises Δ. For example, we give a negative definition of conjunction as follows:

$$\text{(no rule for } \top\text{)} \qquad \frac{\Delta \Vdash \bullet A}{\Delta \Vdash \bullet A \& B} \quad \frac{\Delta \Vdash \bullet B}{\Delta \Vdash \bullet A \& B}$$

$$\frac{}{\cdot \Vdash \bullet \bot} \qquad \frac{\Delta_1 \Vdash \bullet A \quad \Delta_2 \Vdash \bullet B}{\Delta_1, \Delta_2 \Vdash \bullet A \parr B}$$

$$\frac{}{A \Vdash \bullet \neg A}$$

Figure 2.4: Definition of some negative connectives by refutation patterns

$$\text{(no rule for } \top\text{)} \qquad \frac{\Delta \Vdash \bullet A}{\Delta \Vdash \bullet A \& B} \quad \frac{\Delta \Vdash \bullet B}{\Delta \Vdash \bullet A \& B}$$

Intuitively this says that there is no refutation of truth, and to refute $A \& B$ we can refute either A or B. Negative disjunction is defined as follows:

$$\frac{}{\cdot \Vdash \bullet \bot} \qquad \frac{\Delta_1 \Vdash \bullet A \quad \Delta_2 \Vdash \bullet B}{\Delta_1, \Delta_2 \Vdash \bullet A \parr B}$$

This says that falsehood is directly refutable, while a refutation of the disjunction $A \parr B$ requires refutations of both A and B, combining their respective premises. And finally, negative polarity negation is defined by the axiom $A \Vdash \bullet \neg A$

Definition 2.2.2 (Refutation patterns)**.** *A derivation of $\Delta \Vdash \bullet A$ is called a* **refutation pattern***, or more specifically an A-pattern. We use the letter d to range over refutation patterns. As we did with proof patterns, we refer to the frame Δ as the frame of d, and to the set of all A-refutation patterns as the support of A.*

Example 2.2.3. Let $C' = \neg A \& (\neg B_1 \parr \neg B_2)$. There are exactly two C'-patterns:

$$\frac{\overline{A \Vdash \bullet \neg A}}{A \Vdash \bullet C'} \qquad \frac{\dfrac{B_1 \Vdash \bullet \neg B_1 \quad B_2 \Vdash \bullet \neg B_2}{B_1, B_2 \Vdash \bullet \neg B_2 \parr \neg B_2}}{B_1, B_2 \Vdash \bullet C'}$$

■

The definition ordering for negative propositions is defined completely analogously to the positive case: $\Delta \prec B$ if there is some refutation pattern $\Delta \Vdash \bullet B$, and $A \prec \Delta$ it there is some proof hole $A \in \Delta$.

2.2.2 Proofs and refutations, identity and composition, complex hypotheses

We follow the template of §2.1.3, but reverse the roles of proof and refutation. Explicitly, we define the four judgments with the following canonical rules (summarized in Figure 2.5):

$\bullet A$: *To refute A, we must choose some A-refutation pattern, and derive its frame.*

$$\frac{\Delta \Vdash \bullet A \quad \Gamma \vdash \Delta}{\Gamma \vdash \bullet A}$$

Frames	Δ	$::=$	$A \mid \cdot \mid (\Delta_1, \Delta_2)$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \Delta$

$$\frac{\Delta \Vdash \bullet A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A} \qquad \frac{\Delta \Vdash \bullet A \quad \Gamma \vdash \Delta}{\Gamma \vdash \bullet A}$$

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)} \qquad \frac{A \in \Gamma \quad \Gamma \vdash \bullet A}{\Gamma \vdash \#}$$

Figure 2.5: Canonical derivations for negative propositions

A: *To prove A, we must examine every possible A-refutation pattern, and show how to derive a contradiction from its frame.*

$$\frac{\Delta \Vdash \bullet A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A}$$

Δ: *To assert Δ, we must provide evidence for all of its proof holes.*

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)}$$

$\#$: *To derive contradiction, we must find some hypothesis assumed to be true, and refute it.*

$$\frac{A \in \Gamma \quad \Gamma \vdash \bullet A}{\Gamma \vdash \#}$$

It is important to understand that for negative propositions, proof means proof-by-contradiction, whereas refutation must be direct. This is dual to the situation for positive propositions, where proof must be direct, while refutation is by contradiction.

Example 2.2.4. Let C' be as in Example 2.2.3. The derived rules for proving and refuting C' are:

$$\frac{\Gamma, A \vdash \# \quad \Gamma, B_1, B_2 \vdash \#}{\Gamma \vdash C'} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \bullet C'} \quad \frac{\Gamma \vdash B_1 \quad \Gamma \vdash B_2}{\Gamma \vdash \bullet C'}$$

Let us compare these rules with those for the positive proposition $C = \neg A \otimes (\neg B_1 \oplus \neg B_2)$:

$$\frac{\Gamma \vdash \bullet A \quad \Gamma \vdash \bullet B_1}{\Gamma \vdash C} \quad \frac{\Gamma \vdash \bullet A \quad \Gamma \vdash \bullet B_2}{\Gamma \vdash C} \qquad \frac{\Gamma, \bullet A, \bullet B_1 \vdash \# \quad \Gamma, \bullet A, \bullet B_2 \vdash \#}{\Gamma \vdash \bullet C}$$

The positive C and negative C' are both legitimate interpretations of the unpolarized proposition $\neg A \wedge (\neg B_1 \vee \neg B_2)$, but as we see they result in different rules of proof and refutation.

■

For this new notion of canonical derivations, we can state identity and composition principles analogous to those of §2.1.4:

Identity (proof). *If $A \in \Gamma$ then $\Gamma \vdash A$*

Identity (frame). *If $\Delta \in \Gamma$ then $\Gamma \vdash \Delta$*

Composition (reduction). *If* $\Gamma \vdash A$ *and* $\Gamma \vdash \bullet A$ *then* $\Gamma \vdash \#$

Composition (substitution). *If* $\Gamma \vdash \Delta$ *and* $\Gamma(\Delta) \vdash J$ *then* $\Gamma \vdash J$

The proof of these principles is likewise completely analogous, conditioned on well-foundedness of the definition ordering.

Indeed, there is an obvious bijection between the two forms of canonical derivations. Define the dual A° of a formula A as follows:

$$\begin{aligned} 1^\circ &= \bot & \bot^\circ &= 1 \\ 0^\circ &= \top & \top^\circ &= 0 \\ (A \oplus B)^\circ &= A^\circ \& B^\circ & (A \& B)^\circ &= A^\circ \oplus B^\circ \\ (A \otimes B)^\circ &= A^\circ \invamp B^\circ & (A \invamp B)^\circ &= A^\circ \otimes B^\circ \\ (\overset{+}{\neg} A)^\circ &= \overset{-}{\neg} A^\circ & (\overset{-}{\neg} A)^\circ &= \overset{+}{\neg} A^\circ \end{aligned}$$

the dual of an assertion/denial as follows:

$$(A)^\circ = \bullet A^\circ \quad (\bullet A)^\circ = A^\circ$$

and set $\#^\circ = \#$, and extend $(-)^\circ$ to frames pointwise. Note that $(-)^\circ$ is an involution on both propositions and judgments.

Proposition 2.2.5 (Duality)**.**

1. $\Delta \Vdash A$ *iff* $\Delta^\circ \Vdash \bullet A^\circ$
2. $\Gamma \vdash J$ *iff* $\Gamma \vdash J^\circ$
3. $\Gamma(\Delta) \vdash J$ *iff* $\Gamma(\Delta^\circ) \vdash J$

Proof. (1) is immediate from the definition of patterns. From (1), we derive (2) and (3) by mutual induction on the derivations. For example, suppose we have a refutation of a positive proposition:

$$\frac{\Delta \Vdash A \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A}$$

Given a refutation pattern $\Delta \Vdash \bullet A^\circ$, by (1) and the fact that $A^{\circ\circ} = A$ we obtain a proof pattern $\Delta^\circ \Vdash A$, then a derivation of $\Gamma, \Delta^\circ \vdash \#$ by the premise, and hence $\Gamma, \Delta \vdash \#$ by (3) and the fact that $\Delta^{\circ\circ} = \Delta$. Therefore $\Gamma \vdash A^\circ$. □

Finally, just as we introduced complex proof hypotheses to simplify the presentation of derivations in proof-biased logic, here we can introduce complex refutation hypotheses $\bullet A$. Complex refutation variables are used with the following rule, which is invertible:

$$\frac{\Delta \Vdash \bullet A \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(\bullet A) \vdash J}$$

$$\begin{array}{lll} \text{Frames} & \Delta & ::= \bullet A \mid A \mid \cdot \mid (\Delta_1, \Delta_2) \\ \text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, \Delta \end{array}$$

$$\frac{\Delta \Vdash A^+ \quad \Gamma \vdash \Delta}{\Gamma \vdash A^+} \qquad \frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A^+}$$

$$\frac{\Delta \Vdash \bullet A^- \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A^-} \qquad \frac{\Delta \Vdash \bullet A^- \quad \Gamma \vdash \Delta}{\Gamma \vdash \bullet A^-}$$

$$\frac{\bullet A^+ \in \Gamma \quad \Gamma \vdash A^+}{\Gamma \vdash \#} \qquad \frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)} \qquad \frac{A^- \in \Gamma \quad \Gamma \vdash \bullet A^-}{\Gamma \vdash \#}$$

$$\frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(A^+) \vdash J} \qquad \frac{\Delta \Vdash \bullet A^- \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(\bullet A^-) \vdash J}$$

Figure 2.6: Canonical derivations in polarized logic, with complex hypotheses

2.3 Propositional polarized logic

2.3.1 A unified view

We have shown two different ways of defining logical connectives: either positively by their proof patterns, or negatively by their refutation patterns. Now we explain how these alternatives are not incompatible, in the sense that they define different fragments of a single, polarized logic. From now on I use the letters A, B, C to range over *polarized propositions*, which have a definite, positive or negative polarity. Positive polarity is marked explicitly by writing A^+, negative polarity by writing A^-, but this annotation can also be left out when the polarity is clear from context.

Frames can now contain holes for both proofs and refutations, and connectives are defined either by their proof patterns or by their refutation patterns. Canonical derivations in polarized logic are formed by combining the inference rules for positive and negative logic, as summarized in Figure 2.6. We include in the figure both rules for using complex hypotheses.

We likewise combine the identity and composition principles.

Identity (proof)**.** *If* $A \in \Gamma$ *then* $\Gamma \vdash A$

Identity (refutation)**.** *If* $\bullet A \in \Gamma$ *then* $\Gamma \vdash \bullet A$

Identity (frame)**.** *If* $\Delta \in \Gamma$ *then* $\Gamma \vdash \Delta$

Composition (reduction)**.** *If* $\Gamma \vdash \bullet A$ *and* $\Gamma \vdash A$ *then* $\Gamma \vdash \#$

Composition (substitution)**.** *If* $\Gamma(\Delta) \vdash J$ *and* $\Gamma \vdash \Delta$ *then* $\Gamma \vdash J$

Again, these are verified by an argument completely analogous to that of §2.1.4, conditioned on well-foundedness of the definition ordering.

If you have been paying close attention to these definitions, however, you will notice that this is still only a trivial combination of two logics. All of the connectives defined in §2.1.1 and

§2.2.1 preserve polarity, which means that a positive proposition has only positive ancestors, and a negative proposition only negative ancestors. Since the structure of canonical derivations mirrors the definition ordering, there is no real interaction between the two fragments. But what makes polarized logic non-trivial is that we *can* define additional connectives that change polarity. In particular, we will find the following pair of connectives most interesting:

$$\frac{}{A^- \Vdash {\downarrow}A} \qquad \frac{}{\bullet A^+ \Vdash \bullet{\uparrow}A}$$

$\downarrow$ coerces a negative proposition into a positive one, and $\uparrow$ coerces a positive proposition into a negative one. Note that the polarity annotations on the left-hand sides of the pattern axioms are not connectives, they are only to emphasize the polarity flip. When we don't care about the polarity of A and of the resulting shift, we simply write $\updownarrow A$. These connectives—pronounced "down shift" and "up shift", or just "shift"—may at first seem logically vacuous, particularly to a classical logician. But note that A is always an ancestor of $\updownarrow A$, and for this reason the shifts actually have an interesting effect on canonical derivations and on the identity and composition principles. In a sense we can view the shifts as modalities, mediating between the constructively weaker (more permissive) notion of negative proof (by-contradiction) and the stronger notion of positive (direct) proof, and between the weaker notion of positive refutation (by-contradiction) and the stronger notion of negative (direct) refutation.

Proposition 2.3.1. *The shift connectives have the following derived rules of proof and refutation:*

$$\frac{\Gamma \vdash A^-}{\Gamma \vdash {\downarrow}A} \quad \frac{\Gamma, A^- \vdash \#}{\Gamma \vdash \bullet{\downarrow}A} \qquad \frac{\Gamma, \bullet A^+ \vdash \#}{\Gamma \vdash {\uparrow}A} \quad \frac{\Gamma \vdash \bullet A^+}{\Gamma \vdash \bullet{\uparrow}A}$$

Because these are the *only* canonical rules of proof/refutation for shifted proposition, the following instances of composition (reduction)...

1. If $\Gamma \vdash {\downarrow}A$ and $\Gamma \vdash \bullet{\downarrow}A$ then $\Gamma \vdash \#$.
2. If $\Gamma \vdash {\uparrow}A$ and $\Gamma \vdash \bullet{\uparrow}A$ then $\Gamma \vdash \#$.

...reduce immediately to the following instances of composition (substitution) on singleton frames:

1. If $\Gamma \vdash A^-$ and $\Gamma, A^- \vdash \#$ then $\Gamma \vdash \#$.
2. If $\Gamma, \bullet A^+ \vdash \#$ and $\Gamma \vdash \bullet A^+$ then $\Gamma \vdash \#$.

In Chapter 4, we will revisit these principles from a computational perspective.

Besides the shift connectives, we can define two additional connectives that mix polarity:

$$\frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash \bullet A^+ \to B^-} \qquad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash A^+ - B^-}$$

Implication $A \to B$ is defined as a *negative* connective: to refute an implication $A \to B$, we give a proof of A and a refutation of B. The connective $A - B$ is the dual of implication familiar from *subtractive logic* [Crolard, 2001]: its proof conditions are exactly the same as the refutation conditions for $A \to B$. Note that the polarities of the subcomponents A^+ and B^- ensure that we can continue to decompose their proof/refutation patterns, and also that $A \to B$ and $A - B$ have the same ancestors, exactly the union of the ancestors of A and B.

$$\frac{}{A^- \Vdash {\downarrow}A} \qquad \frac{}{\bullet A^+ \Vdash \bullet{\uparrow}A}$$

$$\frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash A^+ - B^-} \qquad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash \bullet A^+ \to B^-}$$

Figure 2.7: The polarity mediating connectives

Proposition 2.3.2. *The following rules for $A \to B$ and $A - B$ are admissible (double-lines indicating bidirectionality):*

$$\dfrac{\dfrac{\Gamma, \bullet B^- \vdash \bullet A^+}{\Gamma, A^+ \vdash B^-}}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \bullet B}{\Gamma \vdash \bullet A \to B}$$

$$\frac{\Gamma \vdash A \to B}{\Gamma \vdash \bullet A - B} \qquad \frac{\Gamma \vdash \bullet A \to B}{\Gamma \vdash A - B}$$

Note that the usual *involutive negation* of linear logic can be defined in terms of implication and subtraction: the dual of a positive proposition A^+ is defined as $A^\perp = A \to \perp$, while the dual of a negative proposition B^- is defined as $B^\perp = 1 - B$.

Proposition 2.3.3 (Involution)**.**

1. $\Delta \Vdash 1 - (A^+ \to \perp)$ *iff* $\Delta \Vdash A^+$
2. $\Delta \Vdash \bullet(1 - A^-) \to \perp$ *iff* $\Delta \Vdash \bullet A^-$

2.3.2 Atomic propositions

We have assumed so far that all propositions are constructed out of the polarized connectives, but we should also consider indecomposable, *atomic* propositions. We use the letters X, Y, Z to stand for atomic propositions, and keeping the assumption that all propositions have a definite polarity, write X^+, Y^-, etc., to indicate the polarity of an atom.

To include atoms in canonical derivations, we add a pair of pattern rules:

$$\frac{}{X^+ \Vdash X^+} \qquad \frac{}{\bullet X^- \Vdash \bullet X^-}$$

and a pair of rules for satisfying the atomic hypotheses in a frame:

$$\frac{X^+ \in \Gamma}{\Gamma \vdash X^+} \qquad \frac{\bullet X^- \in \Gamma}{\Gamma \vdash \bullet X^-}$$

In other words, the only way to reason about atoms is axiomatically.

The identity principle (if $X^+ \in \Gamma$ then $\Gamma \vdash X^+$, and if $\bullet X^- \in \Gamma$ then $\Gamma \vdash \bullet X^-$) is therefore trivial for atomic hypotheses, as is composition: since the only way a derivation of $\Gamma(\Delta) \vdash J$ can use an atomic hypothesis, e.g., $X^+ \in \Delta$ is to show $\Gamma(\Delta) \vdash X^+$, but $\Gamma \vdash \Delta$ already implies $\Gamma \vdash X^+$.

Frames $\hbar \;::=\; \cdots \mid X^+ \mid \bullet X^-$

$$\frac{}{X^+ \Vdash X^+} \qquad \frac{}{\bullet X^- \Vdash \bullet X^-}$$

$$\frac{X^+ \in \Gamma}{\Gamma \vdash X^+} \qquad \frac{\bullet X^+ \in \Gamma}{\Gamma \vdash \bullet X^-}$$

Figure 2.8: Atomic propositions

Definition 2.3.4. *The propositions of* **propositional polarized logic** *(PPL) are built out of polarized atoms using any finite combination of the aforementioned positive connectives* $(1, 0, \otimes, \oplus, \overset{+}{\neg})$*, negative connectives* $(\top, \bot, \&, \invamp, \overset{-}{\neg})$*, and mixed polarity connectives* $(\updownarrow, \to, -)$*.*

Because of the triviality of the identity and composition principles, we do not include atoms in the definition ordering, i.e., the definition trees of X^+ and X^- are empty.

Proposition 2.3.5. *For any proposition A in PPL, and any frame Δ built out of PPL propositions, the definition trees $\prec_A$ and $\prec_\Delta$ are well-founded.*

Theorem 2.3.6. *The identity and composition principles are admissible on all propositions of PPL.*

Proof. A corollary of Proposition 2.3.5 and the generalization of Theorems 2.1.11 and 2.1.12 to derivations of polarized logic. □

2.3.3 The entailment relation(s)

One of the traditional views of logic is as a partial order on propositions, i.e., an entailment relation. We have given an explanation of inference rules and canonical derivations in polarized logic, but have not really discussed entailment for polarized propositions. An excuse for this omission is that there are actually two different canonical ways of defining entailment. Essentially, we can either define entailment "positively" by $A \vdash B$, or "negatively" by $\bullet B \vdash \bullet A$. That is, we can ask whether a proof of A implies a proof of B, or whether a refutation of B implies a refutation of A. In general, these two notions of entailment do not coincide: when the antecedent is positive, positive entailment is a stronger requirement than negative entailment, whereas when the consequent is negative the opposite is the case.

Definition 2.3.7. *We say that A* **positively entails** *B ($A \leq^+ B$) if $A \vdash B$.*

Definition 2.3.8. *We say that A* **negatively entails** *B ($A \leq^- B$) if $\bullet B \vdash \bullet A$.*

Proposition 2.3.9. *$A^+ \leq^+ B$ implies $A^+ \leq^- B$ (for arbitrary polarity B), and $A \leq^- B^-$ implies $A \leq^+ B^-$ (for arbitrary polarity A).*

Proof. We derive $A^+ \leq^- B$ from $A^+ \leq^+ B$ as follows:

$$\cfrac{\cfrac{\overset{Id}{\bullet B, A^+ \vdash \bullet B} \qquad \overset{A^+ \leq^+ B}{\bullet B, A^+ \vdash B}}{\bullet B, A^+ \vdash \#}\dagger}{\bullet B \vdash \bullet A^+}$$

where at step (†) we are applying composition (reduction). To derive $A \leq^+ B^-$ from $A \leq^- B^-$ we reason dually. ☐

Corollary 2.3.10. *$A^+ \leq^+ B^-$ iff $A^+ \leq^- B^-$.*

Because positive entailment is always the stronger relationship between two positive propositions, and negative entailment always the stronger relationship between negative propositions, we say that an entailment $A \leq B$ between like-polarity propositions holds **strongly** if A and B both have polarity p and $A \leq^p B$, or **weakly** if $A \leq^{-p} B$. Because the two forms of entailment coincide when the antecedent is positive and the consequent negative, we simply write $A^+ \leq B^-$ without further specification. Note that when the antecedent is negative and the consequent positive, in general neither form of entailment implies the other. For example, we have $\top \leq^+ 1$ (because 1 has a trivial proof) and $\top \not\leq^- 1$ (because $\top$ is irrefutable), but also $\bot \leq^- 0$ (because $\bot$ has a trivial refutation) and $\bot \not\leq^+ 0$ (because 0 is unprovable). For arbitrary polarities and forms of entailment, we write $A \equiv B$ if both $A \leq B$ and $B \leq A$.

Proposition 2.3.11. *$\leq^+$ and $\leq^-$ are reflexive and transitive.*

Proof. Immediate by the identity and composition principles. ☐

Proposition 2.3.12. *The following rules of entailment between like-polarity propositions are valid strongly (and hence also weakly):*

$$\frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 \otimes A_2 \leq B_1 \otimes B_2} \qquad A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C \qquad A \otimes B \equiv B \otimes A \qquad A \equiv A \otimes 1$$

$$A \otimes B \leq A \qquad A \otimes B \leq B \qquad A \leq A \otimes A$$

$$0 \leq A \qquad A \leq A \oplus B \qquad B \leq A \oplus B \qquad \frac{A \leq C \quad B \leq C}{A \oplus B \leq C}$$

$$A \otimes (B \oplus C) \leq (A \otimes B) \oplus (A \otimes C) \qquad A \otimes 0 \leq B$$

$$\frac{A \leq B \quad A \leq C}{A \leq B \& C} \qquad A \& B \leq A \qquad A \& B \leq B \qquad A \leq \top$$

$$A ⅋ A \leq A \qquad A \leq A ⅋ B \qquad B \leq A ⅋ B$$

$$A ⅋ \bot \equiv A \qquad A ⅋ B \equiv B ⅋ A \qquad (A ⅋ B) ⅋ C \equiv A ⅋ (B ⅋ C) \qquad \frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 ⅋ A_2 \leq B_1 ⅋ B_2}$$

$$A \leq B ⅋ \top \qquad (A ⅋ B) \& (A ⅋ C) \leq A ⅋ (B \& C)$$

$$A \to B \equiv A^\perp ⅋ B \qquad A - B \equiv A \otimes B^\perp \qquad \frac{A \leq B}{\updownarrow A \leq \updownarrow B} \qquad \frac{B \leq A}{\neg A \leq \neg B}$$

Proof. Routine calculation from the definition of the connectives (Figures 2.1, 2.4, 2.7). Note that transitivity implies that the following rules for $\otimes, 1, ⅋, \bot$ are also valid:

$$\frac{A \leq B \quad A \leq C}{A \leq B \otimes C} \qquad A \leq 1 \qquad \bot \leq A \qquad \frac{A \leq C \quad B \leq C}{A ⅋ B \leq C}$$

□

Definition 2.3.13 (Galois connection, cf. [Gierz et al., 2003]). *Let $\mathcal{X}$ and $\mathcal{Y}$ be two partially ordered sets. A (monotone)* **Galois connection** *$f \dashv g$ is a pair of monotone functions $f : \mathcal{X} \to \mathcal{Y}$ and $g : \mathcal{Y} \to \mathcal{X}$ such that $x \le g(y)$ iff $f(x) \le y$. An antitone Galois connection between $\mathcal{X}$ and $\mathcal{Y}$ is a monotone Galois connection between $\mathcal{X}$ and $\mathcal{Y}^{\mathrm{op}}$.*

Proposition 2.3.14. *The following (monotone/antitone) Galois connections are valid (indicating the polarities of A, B, and C as needed for clarity):*

$$\frac{B^+ \le^+ \overset{+}{\neg} A}{A^+ \le^+ \overset{+}{\neg} B} \qquad \frac{\overset{-}{\neg} B \le^- A^-}{\overset{-}{\neg} A \le^- B^-} \qquad \cfrac{\cfrac{\uparrow A \le^- B^-}{A^+ \le B^-}}{A^+ \le^+ \downarrow B} \qquad \frac{A \otimes B \le C^-}{A^+ \le B \to C} \qquad \frac{A^+ \le B^- \invamp C^-}{A^+ - B^- \le C^-}$$

$$\frac{B \le A^\perp}{A \le B^\perp} \qquad \frac{B^\perp \le A}{A^\perp \le B}$$

Proof. Routine calculations. We illustrate the Galois connection $\uparrow \dashv \downarrow$ as an example:

$$\begin{aligned} \uparrow A \le^- B^- \quad &\text{iff} \quad \bullet B^- \vdash \bullet \uparrow A \\ &\text{iff} \quad \bullet B^- \vdash \bullet A^+ \\ &\text{iff} \quad \bullet B^-, A^+ \vdash \# \\ &\text{iff} \quad A^+ \vdash B^- \\ &\text{iff} \quad A^+ \vdash \downarrow B \\ &\text{iff} \quad A^+ \le^+ \downarrow B \end{aligned}$$

□

Corollary 2.3.15. $A^{\perp\perp} \equiv A$.

Proposition 2.3.16. *The following entailments are only weakly valid:*

$$1 \le A \oplus \overset{+}{\neg} A \qquad \overset{+}{\neg}\overset{+}{\neg} A \le A \qquad \downarrow\uparrow A \le A \qquad A \le \uparrow\downarrow A \qquad A \le \overset{-}{\neg}\overset{-}{\neg} A \qquad A \& \overset{-}{\neg} A \le \bot$$

Proof. First, we show the three entailments on the left are weakly (i.e., negatively) valid (the argument for the three on the right is dual).

1. $1 \le^- A \oplus \overset{+}{\neg} A$: as in §2.1.5.

2. The following derivation shows $\overset{+}{\neg}\overset{+}{\neg} A \le^- A$:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\bullet A, \bullet\neg A \vdash \bullet A}\,Id}{\bullet A, \bullet\neg A \vdash \neg A}\,*}{\bullet A, \bullet\neg A \vdash \#}}{\bullet A \vdash \bullet\neg\neg A}$$

applying the derived rule for proving $\overset{+}{\neg} A$:

$$\frac{\Gamma \vdash \bullet A}{\Gamma \vdash \overset{+}{\neg} A} \; *$$

3. The following derivation shows $\downarrow\uparrow A \leq^- A$:

$$\dfrac{\dfrac{\dfrac{\overset{Id}{\bullet A, \uparrow A \vdash \bullet A}}{\bullet A, \uparrow A \vdash \bullet \uparrow A}}{\bullet A, \uparrow A \vdash \#}}{\bullet A \vdash \bullet \downarrow\uparrow A}$$

   applying the derived rules for refuting $\downarrow$ and $\uparrow$.

Next, we observe that these entailments are not strongly (i.e., positively) valid.

1. $1 \leq^+ X \oplus \overset{+}{\neg} X$ iff either $\cdot \vdash X$ or $\cdot \vdash \bullet X$, but both fail.
2. $\overset{+}{\neg}\overset{+}{\neg} X \leq^+ X$ iff $\bullet \overset{+}{\neg} X \vdash X$ iff $X \in (\bullet \overset{+}{\neg} X)$, which is false.
3. $\downarrow\uparrow X \leq^+ X$ iff $\uparrow X \vdash X$ iff $X \in (\uparrow X)$, which is false.

□

These results about $\leq^+$ and $\leq^-$ may be suggestive. In terms of strong entailment, the positive fragment encodes propositional logic with *minimal* negation [Johansson, 1937], which is like intuitionistic negation except that $\overset{+}{\neg} 1 \leq^+ 0$ is not valid. Negative strong entailment encodes the dual logic, where for example double-negation elimination is valid but double-negation introduction is not.[3] And in terms of weak entailment, both positive and negative entailment collapse to classical logic. We will prove all of these facts rigorously in Chapter 3.

2.4 Linear and affine canonical derivations

Although we have used linear logic notation for the polarized connectives, they are non-linear in the sense that they satisfy some non-linear entailments. In particular, the following entailments from Proposition 2.3.12 may seem incongruous with the notation:

$$A \otimes B \leq A \quad A \otimes B \leq B \quad A \leq A \otimes A$$
$$A \parr A \leq A \quad A \leq A \parr B \quad B \leq A \parr B$$

These entailments witness the structural properties of weakening ("hypotheses can be thrown away") and contraction ("hypotheses can be reused"). As it turns out, if we look back at the definition of canonical derivations, we see that these structural properties are fairly isolated. Hypothesis reuse can only occur in the rule for asserting a concatenation of frames, and in the rules of contradiction:

$$\frac{\bullet A^+ \in \Gamma \quad \Gamma \vdash A^+}{\Gamma \vdash \#} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)} \qquad \frac{A^- \in \Gamma \quad \Gamma \vdash \bullet A^-}{\Gamma \vdash \#}$$

[3] This has been called *co-minimal logic* by Vakarelov [2005].

And hypotheses may only be thrown away in the rules for satisfying atomic propositions or the empty frame:

$$\frac{\bullet X^- \in \Gamma}{\Gamma \vdash \bullet X^-} \qquad \frac{}{\Gamma \vdash \cdot} \qquad \frac{X^+ \in \Gamma}{\Gamma \vdash X^+}$$

To obtain *linear* canonical derivations, we simply replace the above rules with the following:

$$\frac{\Gamma \vdash A^+}{\Gamma(\bullet A^+) \vdash \#} \qquad \frac{\Gamma_1 \vdash \Delta_2 \quad \Gamma_1 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash (\Delta_1, \Delta_2)} \qquad \frac{\Gamma \vdash \bullet A^-}{\Gamma(A^-) \vdash \#}$$

$$\frac{}{\bullet X^- \vdash \bullet X^-} \qquad \frac{}{\cdot \vdash \cdot} \qquad \frac{}{X^+ \vdash X^+}$$

The remaining rules of proof and refutation remain unchanged. We can similarly obtain *affine* canonical derivations by only replacing the first three rules. When we want to distinguish ordinary canonical derivations from linear/affine canonical derivations, we call the former *unrestricted canonical derivations.* For reference, we include the complete definition of all three kinds of canonical derivations in Figure 2.9. The modified notions of identity and composition for linear canonical derivations are:

Identity (proof). $A \vdash A$

Identity (refutation). $\bullet A \vdash \bullet A$

Identity (frame). $\Delta \vdash \Delta$

Composition (reduction). *If* $\Gamma_1 \vdash \bullet A$ *and* $\Gamma_2 \vdash A$ *then* $\Gamma_1, \Gamma_2 \vdash \#$

Composition (substitution). *If* $\Gamma_1(\Delta) \vdash J$ *and* $\Gamma_2 \vdash \Delta$ *then* $\Gamma_1(\Gamma_2) \vdash J$

For affine canonical derivations, only the notions of composition are modified. Again, the derivations witnessing these principles are completely analogous to those in §2.1.4, and are well-founded just when the definition ordering is well-founded. Note that the notion of pattern remains unchanged from before, and so the definition ordering remains unchanged as well.

We can define strong and weak entailment using linear/affine canonical derivations just as we did with unrestricted canonical derivations in §2.3.3. The properties of these entailment relations are only slightly different:

Observation 2.4.1. *All the rules of strong entailment from Proposition 2.3.12 hold for linear canonical derivations,* **except for** *the following:*

$$A \parr A \leq A \qquad A \leq A \parr B \qquad B \leq A \parr B$$
$$A \otimes B \leq A \qquad A \otimes B \leq B \qquad A \leq A \otimes A$$

Observation 2.4.2. *All the rules of strong entailment from Proposition 2.3.12 hold for affine canonical derivations,* **except for** *the following:*

$$A \parr A \leq A \qquad A \leq A \otimes A$$

Observation 2.4.3. *All of the Galois connections of Proposition 2.3.14 hold for both linear and affine canonical derivations.*

Frames	Δ	$::=$	$\bullet A \mid A \mid X^+ \mid \bullet X^- \mid \cdot \mid (\Delta_1, \Delta_2)$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \Delta$

$\boxed{\text{Proof and refutation}}$

$$\frac{\Delta \Vdash A^+ \quad \Gamma \vdash \Delta}{\Gamma \vdash A^+} \qquad \frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A^+}$$

$$\frac{\Delta \Vdash \bullet A^- \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A^-} \qquad \frac{\Delta \Vdash \bullet A^- \quad \Gamma \vdash \Delta}{\Gamma \vdash \bullet A^-}$$

$\boxed{\text{Complex hypotheses}}$

$$\frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(A^+) \vdash J} \qquad \frac{\Delta \Vdash \bullet A^- \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(\bullet A^-) \vdash J}$$

$\boxed{\text{With weakening}}$ $\boxed{\text{Weakening-free}}$

$$\frac{\bullet X^- \in \Gamma}{\Gamma \vdash \bullet X^-} \qquad \frac{}{\Gamma \vdash \cdot} \qquad \frac{X^+ \in \Gamma}{\Gamma \vdash X^+} \qquad\qquad \frac{}{\bullet X^- \vdash \bullet X^-} \qquad \frac{}{\cdot \vdash \cdot} \qquad \frac{}{X^+ \vdash X^+}$$

$\boxed{\text{With contraction}}$

$$\frac{\bullet A^+ \in \Gamma \quad \Gamma \vdash A^+}{\Gamma \vdash \#} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)} \qquad \frac{A^- \in \Gamma \quad \Gamma \vdash \bullet A^-}{\Gamma \vdash \#}$$

$\boxed{\text{Contraction-free}}$

$$\frac{\Gamma \vdash A^+}{\Gamma(\bullet A^+) \vdash \#} \qquad \frac{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash (\Delta_1, \Delta_2)} \qquad \frac{\Gamma \vdash \bullet A^-}{\Gamma(A^-) \vdash \#}$$

Figure 2.9: Definition of linear, affine, and unrestricted canonical derivations

Observation 2.4.4. *The weak entailments $1 \leq A \oplus \overset{+}{\neg} A$ and $A \& \overset{-}{\neg} A \leq \bot$ fail for both linear and affine canonical derivations, while the rest of the weak entailments in Proposition 2.3.16 hold.*

Proof. By inspection of the unrestricted derivations in §2.3.3, observing that none make essential use of weakening, and only these two make essential use of contraction. For example, in the unrestricted derivation of $1 \leq^{-} A \oplus \overset{+}{\neg} A$, which reduces to showing $\bullet A \oplus \overset{+}{\neg} A \vdash \#$, we must use the hypothesis $\bullet A \oplus \overset{+}{\neg} A$ twice. $\square$

2.5 Related Work

The literature on polarity, proof theory, and constructive accounts of classical logic is huge. I touch only upon some of the more closely related work.

Polarity in classical linear logic. There is a very old, related notion of polarity in logic and proof theory, in the sense of positive and negative *occurrences* of formulas (see, e.g., Herbrand [1930], Kleene [1967], Schütte [1977]). Polarity as a property of connectives—the notion we use here—was introduced by Girard [1991a, 1993] as a way of recovering constructive content from classical logic, and as an attempt at understanding some general properties of logic—classical, intuitionistic, linear, etc.—in a unified way. The formal treatment of polarity given here (two syntactically segregated classes of connectives, with "shift operators" acting as intermediaries) is a simplification of the original approach, first described in a note by Girard [1991b] and taken up in his more recent work on "ludics" [2001]. This approach has also been given extensive treatment in Olivier Laurent's dissertation [2002].

Polarity outside of classical linear logic. Similar formal devices have appeared elsewhere. For example, as mentioned in the Introduction, Levy's call-by-push-value language maintains a syntactic separation between *value types* and *computation types*, with coercions between them. Likewise, the Concurrent Logical Framework [Watkins et al., 2002] maintains a separation between *synchronous types* and *asynchronous types*. Unlike our presentation and those of Girard and Laurent, in both these settings there is an asymmetry between the two polarities, which makes them somewhat more subtle. They are nonetheless *polarities*, in the sense that they describe the bias of individual connectives towards introduction or elimination—the difference is just that the overall framework of CBPV/CLF has an asymmetry between introduction and elimination. (We will discuss the connection with CBPV in more detail in Chapter 4.)

Proof-theoretic semantics. At the start of the chapter and in the Introduction, I gave a paraphrase of Michael Dummett's attempt at finding a "justification" of the logical laws, through an explanation of the meaning of the logical connectives. The idea that the introduction rules for a connective somehow determine its meaning goes back to an offhand remark by Gentzen, who wrote that "an introduction rule gives, so to say, a definition of the constant in question" [1935, p. 80]. Without a direct connection to structural proof theory, this idea was already explored by various people in the 1930s, particularly Wittgenstein ("It is what is regarded as the justification of an assertion that constitutes the sense of the assertion" [Wittgenstein, 1974, I,§40]), and Brouwer-Heyting-Kolmogorov in their interpretations of intuitionistic logic [Heyting, 1974, Kolmogorov, 1932]. Gentzen's remark, however, was first explored rigorously by Prawitz [1974], by treating the meaning of a proposition as given by its *canonical proofs,* or verifications. For Prawitz, canonical proof meant proof ending in an introduction rule, and the justification of an elimination rule consisted of a local reduction step. For example, from the introduction rule for conjunction,

$$\frac{A \quad B}{A \wedge B}$$

one can justify the two elimination rules,

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

by the following reductions:

$$\dfrac{\dfrac{\begin{matrix}\vdots & \vdots\\ A & B\end{matrix}}{A \wedge B}}{A} \leadsto \begin{matrix}\vdots\\ A\end{matrix} \qquad \dfrac{\dfrac{\begin{matrix}\vdots & \vdots\\ A & B\end{matrix}}{A \wedge B}}{B} \leadsto \begin{matrix}\vdots\\ B\end{matrix}$$

These reductions show how to derive a proof of the conclusion of each of the elimination rules, given a *canonical* proof of their premise.

Dummett [1991] built upon Prawitz's intuition in several ways. First, he realized that by defining a more restrictive notion of canonical proof—ending in a *series* of introduction rules, rather than just a single one—he could then have a more expansive justification procedure—applicable to arbitrary inferences, rather than only to the standard elimination rules. Second, he made the leap of considering that the connectives could alternatively be defined by their elimination rules, which would then justify their introduction rules.[4] For these combined reasons, Dummett's analysis seems to me to have had great foresight in prefiguring the concept of polarity. In some ways it is actually more general than the analysis given here, since Dummett based the pragmatist meaning-theory on a notion of canonical *consequence*, rather than canonical refutation. As I explained at the start of the chapter, I have chosen to present here a symmetric view of proof patterns and refutation patterns as a matter of expediency, since it simplifies the framework while still conveying the basic insight of polarity.

Dummett realized that certain connectives could be given certain interpretations only with difficulty. For example, to give implication and universal quantification a verificationist interpretation, he had to significantly weaken the notion of canonical proof (pp. 272–277). However, he did not take the step of suggesting that the two interpretations could define *different* connectives. He still required "harmony between the two aspects of linguistic practice" (p. 287), rather than "diversity".

[4]Dummett in fact attributes this idea to Martin-Löf, who he says "constructed an entire meaning-theory for the language of mathematics on the basis of the assumption that it is the elimination rules that determine meaning." This is likely a reference to Martin-Löf's work with Peter Hancock [Hancock and Martin-Löf, 1975], about which Martin-Löf wrote to Dummett shortly before the William James Lectures [Martin-Löf, 1976]. For example, Martin-Löf wrote:

> To explain the meaning of an implication $A \supset B$, we must explain what is the purpose (function, role) of a canonical proof of $A \supset B$. And, specializing the explanation given above [for the dependent function space], this purpose is to be applied to a canonical proof of the proposition denoted by A, thereby yielding a canonical proof of the proposition denoted by B. In no way is it correct to say that the meaning of $A \supset B$ is determined by the introduction rule.

On the other hand, Hancock/Martin-Löf never discuss the role of canonical consequence in *justifying* the introduction rules. In the 1983 Siena Lectures, Martin-Löf seems to explicitly adopt a verificationist stance—"The meaning of a proposition is determined by... what counts as a verification of it" (Lecture 3)—and he explains the meaning of implication in terms of its introduction rule. However, there is no real contradiction between the two positions, because the explanation of implication given in the Siena lectures reduces its meaning to that of the *hypothetical judgment*, which is explained in terms of elimination, i.e., in terms of substitution for hypotheses.

Schroeder-Heister has used the phrase *proof-theoretic semantics* for these attempts at finding meaning for the logical connectives entirely within the logical rules [Kahle and Schroeder-Heister, 2006]. Girard's recent work [1998, 2001, 2006] can also be seen in these terms, as trying to give an "internal" semantics of proofs. His attempt, however, draws on many additional concepts from games semantics.

Game-theoretic semantics. Lorenzen [1960, 1961] and Henkin [1961] first explored the idea of treating the truth or falsehood of a proposition as the result of a game between Proponent and Opponent. Henkin's work was later built upon by Hintikka [1973], while Lorenzen's was revisited in the light of linear logic by Blass [1992]. Games semantics for linear logic has been a very active topic of research since Blass's original paper. Although there has always been some tension between the desires of games semantics and the demands of linear logic (see the paper by Melliès and Tabareau [2007, 2008] for a discussion), the basic framework is very compelling. Polarity has a simple interpretation: if a proposition describes a game between Player and Opponent, then polarity says who gets to make the first move [Laurent, 2004b]. Likewise, negation has a very elegant definition: it simply swaps the roles of Player and Opponent. But this means that negation is always an involution, which is at odds with the goal of using games semantics to model mainstream functional programming languages, and negations as continuations. Melliès and Tabareau therefore propose a new direction for games semantics, with non-involutive negation playing the central role.

With canonical derivations, we have seen that there is space for many different kinds of negations. At the most basic level, there is negation at the level of judgments, $\bullet A$. The question of whether it is involutive does not really make sense, because it cannot be iterated. However, as we saw in terms of the differences between strong and weak entailment, for propositions of a given polarity, there is a fundamental asymmetry between assertion A and denial $\bullet A$. But then there is also negation at the level of logical connectives, and we found that there are many: non-involutive negations $\neg A$ of both polarities, and the polarity-reversing negations $A^\perp = A \to \perp$ and $B^\perp = 1 - B$, which together form an involution.

Assertion and denial. We gave refutation a first-class status, distinct from the proof of a negation. Such analyses have been used before in trying to understand the proof theory of classical logic, as well as to make sense of different paraconsistent logics. Smiley's article [1996] represents one such analysis, as does Stewart's analysis of classical natural deduction [1999], and Restall's of multiple conclusion sequent calculus [2005]. (Our notation A and $\bullet A$ for assertion and denial is borrowed from Stewart.) Bellin and Biasi [2004] also draw a similar analogy to the one we made, connecting the duality between assertion and denial (or "conjecture", as they put it) and the duality between positive and negative polarity.

Display Logic. In addition to assertion and denial, our description of canonical derivations relied crucially on the notion of *frame.* In a sense, frames can be seen as turning the comma "," into a connective on judgments, appearing both to the left and to the right of the turnstile. (Note this is different from the comma in Gentzen's multiple conclusion sequent calculus, which means different things on the left and on the right.) We made it *almost* a first-class connective by allowing complex frames, though not completely first-class because we still forbade the contradiction judgment $\#$ in frames. It seems there is an analogy to be drawn with Belnap's *display logic* [Belnap, 1982], which is also formulated in terms of first-class "structural connectives". As Restall [1995, 1998] has observed, these structural connectives can be assigned polarities. Galois connections of the sort we described in §2.3.3 also play an important role, through the link to Dunn's "gaggle theory" [Dunn, 1991]. And there is a generic proof of cut admissibility for display logic, given properties of the structural rules [Belnap, 1982, Dawson and Goré, 2002]. These similarities may hint at a deeper connection.

Infinitary proof theory and iterated inductive definitions. As mentioned in Footnote 1, our rule of refutation for positive propositions, as well as our rule of proof for negative propositions, bears a striking formal resemblance to the Buchholz Ω-rule for iterated inductive definitions [Buchholz et al., 1981]. Buchholz's rule was a powerful generalization of the so-called ω-rule for first-order arithmetic ("derive $\forall n.A(n)$ given proofs of $A(0), A(1), A(2), \ldots$") suggested by Hilbert [1931] and studied by Novikov [1943], Schütte [1950], and Lorenzen [1951]. We have stayed clear of infinity in this chapter, but will embrace it wholeheartedly in Chapter 4, with recursive types that have both infinitely many patterns, and infinitely descending definition trees. The elegance of the approach based on infinitary proof theory is that the *procedure* for composition/cut-elimination is unaffected by whether or not types are infinite, only the argument about whether or not it terminates.

Chapter 3

Focusing proofs and double-negation translations

During a lecture the Oxford linguistic philosopher J. L. Austin made the claim that although a double negative in English implies a positive meaning, there is no language in which a double positive implies a negative. To which Morgenbesser responded in a dismissive tone, "Yeah, yeah."

—Wikipedia entry for Sidney Morgenbesser

As already acknowledged, the preceding chapter was a bit of revisionist history. I described how a certain notion of proof and refutation arises naturally by considering connectives to be defined by *patterns:* either proof patterns (positive connectives), or refutation patterns (negative connectives). I called these canonical derivations, and gave a presentation roughly in the style of Martin-Löf—in the sense of distinguishing multiple forms of judgments—and fundamentally an *iterated* inductive definition, including rules structurally very similar to Buchholz's Ω-rule.

In this chapter, I will explain how canonical derivations can also be seen as an alternative presentation of Jean-Marc Andreoli's *focusing strategies* for sequent calculus. This, of course, is not simply a remarkable coincidence, because the system I presented was derived backwards, with polarity and focusing already in mind. My reason for delaying the discussion of focusing is in part technical: the sequent calculus notation makes a number of unnecessary distinctions, which leads to more cumbersome inference rules, identity and composition principles. But there is also a conceptual reason. At least at first glance, focusing seems to be an extra layer of complexity grafted onto the sequent calculus to obtain a better proof search procedure. Sequent calculus comes first, and focusing is secondary. But this conceptual order is backwards, I would argue: focusing proofs are simpler and more basic than ordinary sequent calculus proofs, precisely because they have a natural interpretation as canonical forms of proof and refutation. At the end of the chapter, we will see how this correspondence gives us a way of understanding focusing as a sort of double-negation interpretation, and indeed a recipe for reconstructing many different double-negation translations of classical logic into minimal logic.

$$\frac{}{A \vdash A}\ \mathit{init} \qquad \frac{\mathfrak{L}_1 \vdash \mathfrak{R}_1, A \quad A, \mathfrak{L}_2 \vdash \mathfrak{R}_2}{\mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2}\ \mathit{cut}$$

$$\frac{\mathfrak{L} \vdash \mathfrak{R}}{1, \mathfrak{L} \vdash \mathfrak{R}}\ 1L \qquad \frac{A, B, \mathfrak{L} \vdash \mathfrak{R}}{A \otimes B, \mathfrak{L} \vdash \mathfrak{R}}\ \otimes L \qquad \frac{\mathfrak{L}_1 \vdash \mathfrak{R}_1, A \quad \mathfrak{L}_2 \vdash \mathfrak{R}_2, B}{\mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2, A \otimes B}\ \otimes R \qquad \frac{}{\cdot \vdash 1}\ 1R$$

$$\frac{A, \mathfrak{L} \vdash \mathfrak{R}}{A \& B, \mathfrak{L} \vdash \mathfrak{R}}\ \& L_1 \qquad \frac{B, \mathfrak{L} \vdash \mathfrak{R}}{A \& B, \mathfrak{L} \vdash \mathfrak{R}}\ \& L_2 \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A \quad \mathfrak{L} \vdash \mathfrak{R}, B}{\mathfrak{L} \vdash \mathfrak{R}, A \& B}\ \& R \qquad \frac{}{\mathfrak{L} \vdash \mathfrak{R}, \top}\ \top R$$

$$\frac{}{0, \mathfrak{L} \vdash \mathfrak{R}}\ 0L \qquad \frac{A, \mathfrak{L} \vdash \mathfrak{R} \quad B, \mathfrak{L} \vdash \mathfrak{R}}{A \oplus B, \mathfrak{L} \vdash \mathfrak{R}}\ \oplus L \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A}{\mathfrak{L} \vdash \mathfrak{R}, A \oplus B}\ \oplus R_1 \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, B}{\mathfrak{L} \vdash \mathfrak{R}, A \oplus B}\ \oplus R_2$$

$$\frac{}{\bot \vdash \cdot}\ \bot L \qquad \frac{A, \mathfrak{L}_1 \vdash \mathfrak{R}_1 \quad B, \mathfrak{L}_2 \vdash \mathfrak{R}_2}{A \parr B, \mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2}\ \parr L \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A, B}{\mathfrak{L} \vdash \mathfrak{R}, A \parr B}\ \parr R \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}}{\mathfrak{L} \vdash \mathfrak{R}, \bot}\ \bot R$$

Figure 3.1: Sequent calculus for multiplicative-additive linear logic

3.1 Focusing proof search for linear logic

Focusing was originally discovered in the context of linear logic [Girard, 1987, Andreoli, 1992], and is most vividly illustrated there. A standard, two-sided presentation of multiplicative-additive linear logic (MALL) is given in Figure 3.1. Sequents are treated modulo reordering of formulas, so that the structural property of *exchange* is implicit. The structural properties of *weakening* and *contraction* are explicitly omitted. We write $\mathfrak{L} \vdash_\ell \mathfrak{R}$ to indicate that the sequent $\mathfrak{L} \vdash \mathfrak{R}$ is derivable from these inference rules. To begin we will give a simple-minded proof search algorithm as a decision procedure for $\vdash_\ell$, and then see how to refine this procedure through focusing.

3.1.1 Naive proof search for linear logic

The naive procedure relies only on a few facts about MALL, which we state without proof.

Definition 3.1.1. *We say that B is an* **immediate syntactic subformula** *of A (written $B \lessdot A$) if $A = \odot(B_1, \ldots, B_n)$ for some n-ary connective $\odot$, and $B = B_i$ for some i.*

We extend the syntactic subformula ordering to a multiset ordering on sequents: we say that the sequent $\mathfrak{L}' \vdash \mathfrak{R}'$ is strictly smaller than $\mathfrak{L} \vdash \mathfrak{R}$ (written $\mathfrak{L}' \vdash \mathfrak{R}' \lessdot \mathfrak{L} \vdash \mathfrak{R}$), if the formulas of $\mathfrak{L}' \uplus \mathfrak{R}'$ are obtained by removing some formula $A \in \mathfrak{L} \uplus \mathfrak{R}$ and replacing it by a list $A_1, \ldots, A_n$ of immediate syntactic subformulas.

Proposition 3.1.2. *The ordering $\lessdot$ on sequents is well-founded.*

Definition 3.1.3. *The L-rules and R-rules are called* **logical rules.** *The unique formula introduced on the left or right in the conclusion of a logical rule is called its* **principal formula***. The syntactic subformulas of the principal formula appearing in the premises of a logical rule are called* **active formulas.** *The remaining formulas carried through in $\mathfrak{L}$ and $\mathfrak{R}$ are called* **the context***. We sometimes write the context as a sequent, i.e., the context of the formula A in the sequent $\mathfrak{L} \vdash \mathfrak{R}, A$ (or $A, \mathfrak{L} \vdash \mathfrak{R}$) is written as $\mathfrak{L} \vdash \mathfrak{R}$.*

Observation 3.1.4. *In every logical rule, the premises are strictly smaller than the conclusion.*

Theorem 3.1.5 (Init-elimination)**.** *Any derivation that uses* (*init*) *can be converted to one where* (*init*) *is restricted to atomic formulas.*

Theorem 3.1.6 (Cut-elimination)**.** *Any derivation that uses* (*cut*) *can be converted to one that doesn't.*

Corollary 3.1.7. *If $\mathfrak{L} \vdash_\ell \mathfrak{R}$, then there is a derivation of $\mathfrak{L} \vdash \mathfrak{R}$ using only logical rules and the* (*init*) *rule restricted to atomic formulas.*

Now, consider the following simple decision procedure for MALL sequents (and as a special case, MALL formulas), which attempts to build a proof "backwards", i.e., starting from the goal $\mathfrak{L} \vdash \mathfrak{R}$ as the root, and trying to build up a proof tree:

1. Find a logical rule whose conclusion matches the goal sequent $\mathfrak{L} \vdash \mathfrak{R}$, and recursively try to prove each premise as a goal. If the rule has no premises ($1R$, $\top R$, $0L$, $\bot L$) then the sequent is provable and we are done. Note that some rules ($\otimes R$, $⅋ L$) can be applied in multiple ways, by choosing different splittings of the context.

2. Suppose the sequent does not fit the conclusion of any logical rule: if it is an atomic initial sequent $X \vdash X$ then we can apply (*init*) to complete the proof; otherwise, the sequent is unprovable, and we must backtrack to one of our earlier goals in (1), and try to prove it by different means (i.e., with a different rule, or a different splitting of the context).

Corollary 3.1.7 implies that if $\mathfrak{L} \vdash_\ell \mathfrak{R}$, then this procedure will always find a proof given an oracle for step (1). The combination of Observation 3.1.4 and Proposition 3.1.2, together with the fact that there are only finitely many logical rules and finitely many splittings of a context, implies that the oracle is superfluous, and the procedure will always terminate with either a derivation of $\mathfrak{L} \vdash \mathfrak{R}$ or the knowledge that it is unprovable.

That said, the procedure is wildly inefficient. The source of this inefficiency is the large number of potential rules/context-splittings that must be tried in step (1). The context-splitting problem, which we call *multiplicative nondeterminism*, can be mitigated by a range of techniques (falling under the corporate-sounding title "resource management"), and is in any case peculiar to the rules ($\otimes R$) and ($⅋ L$)—let us put it aside, and concentrate on the problem of picking a rule. We can potentially choose *any* non-atomic formula in the goal sequent as the principal formula of a logical rule, and then possibly choose between multiple rules for that formula (if the formula is $A \oplus B$ on the right of the sequent, or $A \& B$ on the left). Again, let us put aside the latter source of nondeterminism—we call it *additive nondeterminism*[1]—and concentrate on the act of choosing some formula in the sequent to be the principal formula. Suppose we make the wrong choice and the new goals are unprovable, a priori that does not tell us that the original goal is unprovable, and we must go back and try a different formula. As a simple example, suppose we are trying to prove $X \oplus Y \vdash X \oplus Y$, and begin by choosing the formula on the right. Whether we apply ($\oplus R_1$) or ($\oplus R_2$), the new goal will be unprovable—and yet the original sequent *is* provable, if we begin by choosing the formula on the left and applying ($\oplus L$). In general, then, it seems we must backtrack and try every possible non-atomic formula in a sequent as the principal formula, if we want to be guaranteed of either finding a proof or establishing that the sequent

[1]Girard calls $\otimes$ and $⅋$ multiplicative, $\&$ and $\oplus$ additive, hence this terminology—although additive nondeterminism is "multiplicative" in the sense that the search space multiplies as we work up the proof tree.

is unprovable. . . but this impression turns out to be mostly mistaken! Andreoli's focusing proof search consists of two observations that whittle down much of the nondeterminism in picking a principal formula.

3.1.2 Observation #1: Invertibility and the Inversion Phase

The first observation is simple: many of the rules are invertible (recall, a rule is invertible if its conclusion implies its premises). In particular, all of the following left rules are invertible:

$$\frac{}{0, \mathfrak{L} \vdash \mathfrak{R}}\ 0L \qquad \frac{A, \mathfrak{L} \vdash \mathfrak{R} \quad B, \mathfrak{L} \vdash \mathfrak{R}}{A \oplus B, \mathfrak{L} \vdash \mathfrak{R}}\ \oplus L \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}}{1, \mathfrak{L} \vdash \mathfrak{R}}\ 1L \qquad \frac{A, B, \mathfrak{L} \vdash \mathfrak{R}}{A \otimes B, \mathfrak{L} \vdash \mathfrak{R}}\ \otimes L$$

as are their dual right rules:

$$\frac{}{\mathfrak{L} \vdash \mathfrak{R}, \top}\ \top R \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A \quad \mathfrak{L} \vdash \mathfrak{R}, B}{\mathfrak{L} \vdash \mathfrak{R}, A \& B}\ \& R \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}}{\mathfrak{L} \vdash \mathfrak{R}, \bot}\ \bot R \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A, B}{\mathfrak{L} \vdash \mathfrak{R}, A \parr B}\ \parr R$$

We say that the connectives $\otimes, \oplus, 1, 0$ are *left-invertible,* while $\&, \parr, \top, \bot$ are *right-invertible.* When an invertible rule is applied during step (1) of proof search, there is no need for backtracking if the new goals fail: that the premises are unprovable *is* sufficient evidence that the conclusion is unprovable. In other words, if rules are read bottom-up as *goal transformers* (taking a goal to a set of new goals), an invertible rule is a "safe" transformation in the sense that it preserves provability. We can take this a bit further by building in an *inversion phase.*

Definition 3.1.8. *A formula inside a sequent (or to be more precise, an* occurrence *of a formula) is* **invertible** *if it matches the conclusion of an invertible logical rule, and* **stable** *otherwise. A sequent/context is invertible if it contains at least one invertible formula, and stable if it contains only stable formulas.*

During the inversion phase of proof search, we greedily apply invertible rules as goal transformers, until we are left with a set of stable sequents as goals. Note that the order in which we pick different invertible formulas in the sequent to invert is irrelevant, not only with respect to provability but also with respect to the ultimate set of stable sequents. For example, when inverting $(A_1 \& A_2) \otimes X \vdash Y \& (B_1 \oplus B_2)$, whether we first decompose the $\otimes$ on the left, or the $\&$ on the right, we are left with the same two stable sequents:

$$\frac{\dfrac{A_1 \& A_2, X \vdash Y \quad A_1 \& A_2, X \vdash B_1 \oplus B_2}{A_1 \& A_2, X \vdash Y \& (B_1 \oplus B_2)}\ \& R}{(A_1 \& A_2) \otimes X \vdash Y \& (B_1 \oplus B_2)}\ \otimes L \qquad \frac{\dfrac{A_1 \& A_2, X \vdash Y}{(A_1 \& A_2) \otimes X \vdash Y}\ \otimes L \quad \dfrac{A_1 \& A_2, X \vdash B_1 \oplus B_2}{(A_1 \& A_2) \otimes X \vdash B_1 \oplus B_2}\ \otimes L}{(A_1 \& A_2) \otimes X \vdash Y \& (B_1 \oplus B_2)}\ \& R$$

More generally, we can view inversion as replacing a single formula A in context $\mathfrak{L} \vdash \mathfrak{R}$ with a unique set of stable contexts $\mathfrak{L}_i \vdash \mathfrak{R}_i$, $i = 1..n$, deriving the new set of goals $\mathfrak{L}_i, \mathfrak{L} \vdash \mathfrak{R}, \mathfrak{R}_i$. Since the original context is carried through unchanged, it doesn't matter in which order we examine the different invertible formulas in a sequent. And since the result of inversion is unique, we can view the inversion phase as operating in one big deterministic step.[2]

[2]The reader may have noticed we left out $(1R)$ and $(\bot L)$ from the list of invertible logical rules, although they are trivially invertible since they have no premises. This omission can be understood if we read the rules as imposing a side condition on the context:

$$\frac{\mathfrak{L} = \cdot \quad \mathfrak{R} = \cdot}{\mathfrak{L} \vdash \mathfrak{R}, 1} \qquad \frac{\mathfrak{L} = \cdot \quad \mathfrak{R} = \cdot}{\bot, \mathfrak{L} \vdash \mathfrak{R}}$$

In terms of proof search, the presence of 1 on the right or $\bot$ on the left cannot be applied greedily, because they force the rest of the context to be empty.

3.1.3 Observation #2: Focalization and the Focus Phase

The second observation is much less obvious, although it turns out to be the dual of the first. Consider the remaining right rules:

$$\frac{\mathfrak{L} \vdash \mathfrak{R}, A}{\mathfrak{L} \vdash \mathfrak{R}, A \oplus B} \oplus R_1 \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, B}{\mathfrak{L} \vdash \mathfrak{R}, A \oplus B} \oplus R_2 \qquad \frac{\mathfrak{L}_1 \vdash \mathfrak{R}_1, A \quad \mathfrak{L}_2 \vdash \mathfrak{R}_2, B}{\mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2, A \otimes B} \otimes R \qquad \frac{}{\cdot \vdash 1} 1R$$

and dual left rules:

$$\frac{A, \mathfrak{L} \vdash \mathfrak{R}}{A \& B, \mathfrak{L} \vdash \mathfrak{R}} \& L_1 \qquad \frac{B, \mathfrak{L} \vdash \mathfrak{R}}{A \& B, \mathfrak{L} \vdash \mathfrak{R}} \& L_2 \qquad \frac{A, \mathfrak{L}_1 \vdash \mathfrak{R}_1 \quad B, \mathfrak{L}_2 \vdash \mathfrak{R}_2}{A \parr B, \mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2} \parr L \qquad \frac{}{\bot \vdash \cdot} \bot L$$

Suppose we have a stable sequent with non-atomic formulas. To build a potential proof of this sequent, we must begin (backwards) by applying one of these rules. But where do we go after that? First, pay attention to the fact that in all of these rules, each premise has exactly one active formula (cf. Definition 3.1.3). Andreoli's observation was the following: after we (nondeterministically) choose some formula to be principal, and then apply some logical rule on that formula (absorbing any multiplicative/additive nondeterminism), it is sufficient to take the active formula in each premise of the rule as the principal formula of the *next* logical rule—and pick the active formula in each of *its* premises as the next principal formula, etc. Andreoli calls this part of proof search the *focusing phase,* since we are always focused on a particular formula in a sequent. The focusing phase ends once we reach either an invertible premise, or an atom. Let us make this slightly more precise:

Definition 3.1.9. *We say that proof search has entered the* **focus** *phase/is* **focused on** *a particular formula inside a stable sequent, if we have committed to using that formula as the principal formula of the first logical rule (i.e., at the root of the proof), and to maintaining focus on the unique active formula in each premise of that rule, unless that formula is invertible or atomic. Prior to such a commitment, we say that proof search is in the* **neutral** *phase.*

We can make a few observations about this definition:

- The focus formula can be either on the left or right of the sequent: left if its outermost connective is among $\&, \parr, \top, \bot$, right if its outermost connective is among $\otimes, \oplus, 1, 0$.
- The definition does not completely specify the treatment of atoms, which is flexible: we can end the focus phase either by completing the proof with an atomic initial sequent, or by going back to the neutral phase. However, for any particular atom, we must be consistent about this choice, either always using an initial sequent when the atom is in right-focus, or always when it is in left-focus.
- If the focus phase ends by going back to an inversion phase, there is exactly one formula to invert.

Focusing proof search consists of the entire cycle, starting from an inversion phase, moving to the neutral and then the focus phase, and then either completing the proof with an initial sequent or going back to an inversion phase. What is remarkable is that this search strategy is complete, i.e., it will always find a proof if one exists. Since the inversion phase obviously preserves provability, what remains to show completeness is the *focalization lemma.*

Lemma 3.1.10 (Focalization [Andreoli, 1992]). *Any provable stable sequent has a proof that begins by focusing on some formula.*

Corollary 3.1.11. *Focusing proof search is complete.*

Focalization is not, prima facie, an obvious property. It implies that once we have chosen the principal formula of the first logical rule, we do not need to make any more choices about principal formulas until we get back to (and complete) an inversion phase. What seemed like a hopeless amount of nondeterminism in the naive proof search algorithm of §3.1.1 can actually be whittled down to the following in focusing proof search:

- a nondeterministic transition from the neutral phase to the focus phase, picking a focus formula
- any multiplicative/additive nondeterminism incurred during the focus phase

This is really a difference of orders of magnitude, and so focusing is very important for efficient backwards proof search in linear logic—but that is selling it short. For one, focusing is also essential to practical "forward" search procedures for linear logic (where we try to work down from axioms to the goal).[3] The wider significance of focusing proof search, though, is the effect it has on *proofs*. We will explain this contention by describing the close correspondence between MALL focusing proofs and linear canonical derivations.

3.2 Relating focusing proofs to canonical derivations

3.2.1 Polarity, invertibility, and focalization

We can see that the MALL connectives divide very neatly according to their focusing strategy: $\otimes, \oplus, 1, 0$ are inverted on the left and focused on the right, while $\&, \mathbin{\rotatebox[origin=c]{180}{\&}}, \top, \bot$ are inverted on the right and focused on the left. This tentatively suggests the following relationship between connectives' focusing behavior and the notion of polarity used in Chapter 2:

positive $\sim$ *invert-left/focus-right* *negative* $\sim$ *invert-right/focus-left*

This correspondence might seem a bit strange conceptually, though. In Chapter 2, we explained polarity as a way of *defining* the propositional connectives, either in terms of proof or in terms of refutation. Here we have worked backwards, starting with the sequent calculus for linear logic—which already distinguishes between different forms of conjunction and disjunction—and showing how to derive Andreoli's focusing algorithm by reasoning about the invertibility and focalizability of rules. Whereas before we saw positive and negative polarity as two legitimate alternatives, here it appears there is no room for choice about the focusing strategy.

But that is not entirely correct. For one, we already saw some flexibility in the treatment of atoms—and even for logical connectives, focusing behavior cannot always be completely determined by sequent calculus rules. Consider negation, and the (seemingly trivial) shift:

$$\frac{\mathfrak{L} \vdash \mathfrak{R}, A}{\neg A, \mathfrak{L} \vdash \mathfrak{R}} \qquad \frac{A, \mathfrak{L} \vdash \mathfrak{R}}{\mathfrak{L} \vdash \mathfrak{R}, \neg A}$$

$$\frac{A, \mathfrak{L} \vdash R}{\updownarrow A, \mathfrak{L} \vdash \mathfrak{R}} \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A}{\mathfrak{L} \vdash \mathfrak{R}, \updownarrow A}$$

[3]See [Chaudhuri and Pfenning, 2005].

In typical presentations of linear logic, negation is defined as a syntactic operation $(-)^\perp$ on formulas, rather than through left and right rules—and we can see one reason why: its introduction rules do not fit the pattern above, since *both* are invertible. Likewise, both left and right rules for $\updownarrow$ are invertible. The fact that every rule is invertible means we are not forced into adopting a particular focusing strategy for $\neg A$ and $\updownarrow A$, and either of the following strategies for navigating between focus and inversion phases are sensible:

1. Always remain in the same phase (i.e., keep A in focus if the conclusion is in focus, invert A if the conclusion is being inverted), or

2. Always switch phases (i.e., invert A if the conclusion is in focus, stop inverting A if the conclusion is being inverted)

We will adopt strategy (2) for both. This corresponds to declaring that $\neg$ *preserves* polarity, while $\updownarrow$ *reverses* polarity: the rules for $\neg A$ end the focus/inversion phase because A has the same polarity but is moved to the opposite side of the sequent, while the rules for $\updownarrow A$ end focus/inversion because they retain A on the same side of the sequent although it has opposite polarity. On the other hand, there is nothing about the sequent calculus rules that forces us into this policy. For example, we could adopt strategy (1) for negation—but then we should call it by a different name.

$$\frac{\mathfrak{L} \vdash \mathfrak{R}, A}{A^\perp, \mathfrak{L} \vdash \mathfrak{R}} \qquad \frac{A, \mathfrak{L} \vdash \mathfrak{R}}{\mathfrak{L} \vdash \mathfrak{R}, A^\perp}$$

The negation $A^\perp$ is defined by the same introduction rules as $\neg A$, but declaring that $A^\perp$ has the opposite polarity of A corresponds to adopting focusing behavior (1) rather than (2).

These examples show that focusing behavior cannot always be inferred by looking at the sequent calculus rules. But we *can* make it explicit, just as we made polarity explicit in Chapter 2. We will again adopt our conventions:

1. Every formula (including atoms) has definite positive or negative polarity, indicated A^+ or A^-

2. Every connective combines formulas of a specific polarity to produce a formula of specific polarity.

In particular, the connectives $\otimes, \oplus, 1, 0$ combine positive formulas into a positive formula, while $\&, \parr, \top, \perp$ combine negative formulas into a negative formula. As in §2.3.1, we distinguish different versions of $\updownarrow A$ and $\neg A$, based on the polarity of A: we write $\uparrow A$ and $\overset{+}{\neg} A$ when A is positive (the results are respectively negative or positive), and $\downarrow A$ and $\overset{-}{\neg} A$ when A is negative (the results are respectively positive or negative). Finally, we add implication $A \to B$ (usually written $A \multimap B$ in MALL) and subtraction $A - B$ with their standard rules:

$$\frac{A, \mathfrak{L} \vdash \mathfrak{R}}{\mathfrak{L} \vdash \mathfrak{R}, A \to B} \qquad \frac{\mathfrak{L}_1 \vdash A, \mathfrak{R}_1 \quad B, \mathfrak{L}_2 \vdash \mathfrak{R}_2}{A \to B, \mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2} \qquad \frac{A, \mathfrak{L} \vdash \mathfrak{R}}{A - B, \mathfrak{L} \vdash \mathfrak{R}} \qquad \frac{\mathfrak{L}_1 \vdash A, \mathfrak{R}_1 \quad B, \mathfrak{L}_2 \vdash \mathfrak{R}_2}{\mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2, A - B}$$

And we adopt the convention that $A \to B$ (resp. $A - B$) is negative (resp. positive) if A is positive and B negative. Note that certain formulas of MALL are not well-polarized according to these conventions—for example $1\&1$, because $\&$ only applies to negative formulas. However, to produce a logically equivalent well-polarized formula we can simply insert shifts, e.g., $\uparrow 1 \& \uparrow 1$.

Under these conventions, technically we are working with what Olivier Laurent calls MALLP, or polarized MALL. However, since the difference between MALL and MALLP is very minor, we will keep referring to it as MALL, implicitly assuming the polarity conventions.

3.2.2 Focusing proofs, through a microscope

In §3.2, we explained focusing informally as a search procedure over a subset of all MALL sequent calculus proofs, which are called the *focusing proofs*. An alternative, more structural way of viewing focusing is that it is itself defined by a sequent calculus that maintains "punctuation" to distinguish between the inversion, neutral, and focus phases. Andreoli defined such a sequent calculus (Σ_3) in his original paper, and in this section we will consider a very similar presentation. We begin by applying the polarity discipline to give a more precise characterization of contexts.

Recall (Definition 3.1.8) that a stable context $\mathfrak{L} \vdash \mathfrak{R}$ cannot contain any invertible formulas. Thus the left half can only contain negative formulas or positive atoms (non-atomic positive formulas are left-invertible), and the right half only positive formulas or negative atoms (non-atomic negative formulas are right-invertible). We will use bold-face letters $\mathbf{L}$ and $\mathbf{R}$ to range over these stable halves:

$$\begin{array}{rcl} \mathbf{L} & ::= & \cdot \mid \mathbf{L}, A^- \mid \mathbf{L}, X^+ \\ \mathbf{R} & ::= & \cdot \mid \mathbf{R}, A^+ \mid \mathbf{R}, X^- \end{array}$$

An invertible context, on the other hand, can contain positive formulas on the left or negative formulas on the right. We use italics letters L and R to range over these invertible halves:

$$\begin{array}{rcl} L & ::= & \cdot \mid L, A^+ \\ R & ::= & \cdot \mid R, A^- \end{array}$$

Note we don't force A to be non-atomic, but as part of the inversion phase we will transfer any atoms X^+ from L to $\mathbf{L}$, and any X^- from R to $\mathbf{R}$. Now, let us reformulate the focusing algorithm by giving inference rules for deriving

$$\begin{array}{rll} & \textit{invertible sequents} & L; \mathbf{L} \vdash \mathbf{R}; R \\ & \textit{neutral sequents} & \mathbf{L} \vdash \mathbf{R} \\ \text{and} & \textit{focused sequents} & \mathbf{L} \vdash \mathbf{R}; [A^+] \text{ or } [A^-]; \mathbf{L} \vdash \mathbf{R}. \end{array}$$

To prove an invertible sequent we apply a series of invertible logical rules, for example:

$$\frac{A, B, L; \mathbf{L} \vdash \mathbf{R}; R}{A \otimes B, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}; R, A \quad L; \mathbf{L} \vdash \mathbf{R}; R, B}{L; \mathbf{L} \vdash \mathbf{R}; R, A \& B}$$

$$\frac{A, L; \mathbf{L} \vdash \mathbf{R}; R \quad B, L; \mathbf{L} \vdash \mathbf{R}; R}{A \oplus B, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}; R, A, B}{L; \mathbf{L} \vdash \mathbf{R}; R, A \parr B}$$

$$\frac{A, L; \mathbf{L} \vdash \mathbf{R}; R, B}{A - B, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{A, L; \mathbf{L} \vdash \mathbf{R}; R, B}{L; \mathbf{L} \vdash \mathbf{R}; R, A \to B}$$

The inversion phase draws to a close as we slowly bring formulas from L and R into the stable context, eventually reaching a neutral sequent:

$$\frac{L; X^+, \mathbf{L} \vdash \mathbf{R}; R}{X^+, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}, X^-; R}{L; \mathbf{L} \vdash \mathbf{R}; R, X^-}$$

$$\frac{L; A^-, \mathbf{L} \vdash \mathbf{R}; R}{\downarrow A, L; \mathbf{L} \vdash \mathbf{R}; R} \quad \frac{L; \mathbf{L} \vdash \mathbf{R}, A^+; R}{L; \mathbf{L} \vdash \mathbf{R}; R, \uparrow A} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}, A^+; R}{\overset{+}{\neg} A, L; \mathbf{L} \vdash \mathbf{R}; R} \quad \frac{L; \mathbf{L}, A^- \vdash \mathbf{R}; R}{L; \mathbf{L} \vdash \mathbf{R}; R, \overset{-}{\neg} A}$$

$$\frac{\mathbf{L} \vdash \mathbf{R}}{\cdot; \mathbf{L} \vdash \mathbf{R}; \cdot}$$

To prove a neutral sequent $\mathbf{L} \vdash \mathbf{R}$, we must pick some formula in $\mathbf{L}$ or $\mathbf{R}$ to focus on:

$$\frac{[A^-]; \mathbf{L} \vdash \mathbf{R}}{A^-, \mathbf{L} \vdash \mathbf{R}} \quad \frac{\mathbf{L} \vdash \mathbf{R}; [A^+]}{\mathbf{L} \vdash \mathbf{R}, A^+}$$

Note that we treat the stable context as unordered, so the focus formula can come from anywhere in the context (not necessarily from the perimeter). During the focus phase, we decompose the formula by applying logical rules, for example:

$$\frac{\mathbf{L}_1 \vdash \mathbf{R}_1; [A] \quad \mathbf{L}_2 \vdash \mathbf{R}_2; [B]}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [A \otimes B]} \qquad \frac{[A]; \mathbf{L} \vdash \mathbf{R}}{[A \& B]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{[B]; \mathbf{L} \vdash \mathbf{R}}{[A \& B]; \mathbf{L} \vdash \mathbf{R}}$$

$$\frac{\mathbf{L} \vdash \mathbf{R}; [A]}{\mathbf{L} \vdash \mathbf{R}; [A \oplus B]} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; [B]}{\mathbf{L} \vdash \mathbf{R}; [A \oplus B]} \qquad \frac{[B]; \mathbf{L}_1 \vdash \mathbf{R}_1 \quad [A]; \mathbf{L}_2 \vdash \mathbf{R}_2}{[A \parr B]; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2}$$

$$\frac{\mathbf{L}_1 \vdash \mathbf{R}_1; [A] \quad [B]; \mathbf{L}_2 \vdash \mathbf{R}_2}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [A - B]} \qquad \frac{\mathbf{L}_1 \vdash \mathbf{R}_1; [A] \quad [B]; \mathbf{L}_2 \vdash \mathbf{R}_2}{[A \multimap B]; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2}$$

Observe that in these rules, the premises are still focused sequents. By our polarity conventions, the only way to end the focus phase (other than by reaching a logical rule with no premises, such as for the constants 1 and $\bot$) is by reaching either an atom, a shift $\updownarrow$, or negation $\neg$:

$$\frac{}{X^+ \vdash \cdot; [X^+]} \quad \frac{}{[X^-]; \cdot \vdash X^-} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{\mathbf{L} \vdash \mathbf{R}; [\downarrow A]} \quad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{[\uparrow A]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{\mathbf{L} \vdash \mathbf{R}; [\overset{+}{\neg} A]} \quad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{[\overset{-}{\neg} A]; \mathbf{L} \vdash \mathbf{R}}$$

This completes our definition of focusing proofs (summarized in Figure 3.2 on a representative fragment of logical rules). Again, the relevant fact for proof search is that the focusing sequent calculus is complete for MALL provability, starting from an inversion phase. Writing $\vdash_{[\ell]}$ for provability in the focusing sequent calculus, we have the following (note we have to be careful about separating invertible formulas from the stable context):

Theorem 3.2.1 (Completeness of Σ_3-style focusing)**.** *If* $L, \mathbf{L} \vdash_\ell \mathbf{R}, R$ *then* $L; \mathbf{L} \vdash_{[\ell]} \mathbf{R}; R$

This is a more structural way of restating Corollary 3.1.7. The focusing rules are also obviously sound for MALL provability, because if we ignore structural punctuation, each is either an instance of an ordinary sequent calculus rule, or else trivial (i.e., the conclusion and the premise are the same).

Theorem 3.2.2 (Soundness of Σ_3-style focusing for MALL)**.**

- *If* $\mathbf{L} \vdash_{[\ell]} \mathbf{R}$ *then* $\mathbf{L} \vdash_\ell \mathbf{R}$
- *If* $\mathbf{L} \vdash_{[\ell]} \mathbf{R}; [A]$ *then* $\mathbf{L} \vdash_\ell \mathbf{R}, A$
- *If* $[A]; \mathbf{L} \vdash_{[\ell]} \mathbf{R}$ *then* $A, \mathbf{L} \vdash_\ell \mathbf{R}$
- *If* $L; \mathbf{L} \vdash_{[\ell]} \mathbf{R}; R$ *then* $L, \mathbf{L} \vdash_\ell \mathbf{R}, R$

$\boxed{\text{Contexts}}$

Left-stable	$\mathbf{L}$	$::=$	$\cdot \mid \mathbf{L}, A^- \mid \mathbf{L}, X^+$
Right-stable	$\mathbf{R}$	$::=$	$\cdot \mid \mathbf{R}, A^+ \mid \mathbf{R}, X^-$
Left-invertible	L	$::=$	$\cdot \mid L, A^+$
Right-invertible	R	$::=$	$\cdot \mid R, A^-$

$\boxed{\text{Sequents}}$

Neutral	$\mathbf{L} \vdash \mathbf{R}$
Right-focus	$\mathbf{L} \vdash \mathbf{R}; [A^+]$
Left-focus	$[A^-]; \mathbf{L} \vdash \mathbf{R}$
Inversion	$L; \mathbf{L} \vdash \mathbf{R}; R$

. . . .

Inversion phase

$$\frac{L; \mathbf{L} \vdash \mathbf{R}; R}{1, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{A, B, L; \mathbf{L} \vdash \mathbf{R}; R}{A \otimes B, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}; R, A \quad L; \mathbf{L} \vdash \mathbf{R}; R, B}{L; \mathbf{L} \vdash \mathbf{R}; R, A \& B} \qquad \frac{}{L; \mathbf{L} \vdash \mathbf{R}; R, \top}$$

$$\frac{}{0, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{A, L; \mathbf{L} \vdash \mathbf{R}; R \quad B, L; \mathbf{L} \vdash \mathbf{R}; R}{A \oplus B, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}; R, A, B}{L; \mathbf{L} \vdash \mathbf{R}; R, A \parr B} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}; R}{L; \mathbf{L} \vdash \mathbf{R}; R, \bot}$$

Inversion → Neutral

$$\frac{L; X^+, \mathbf{L} \vdash \mathbf{R}; R}{X^+, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; A^-, \mathbf{L} \vdash \mathbf{R}; R}{{\downarrow}A, L; \mathbf{L} \vdash \mathbf{R}; R} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}, A^+; R}{L; \mathbf{L} \vdash \mathbf{R}; R, {\uparrow}A} \qquad \frac{L; \mathbf{L} \vdash \mathbf{R}, X^-; R}{L; \mathbf{L} \vdash \mathbf{R}; R, X^-} \qquad \frac{\mathbf{L} \vdash \mathbf{R}}{\cdot; \mathbf{L} \vdash \mathbf{R}; \cdot}$$

Neutral → Focus

$$\frac{\mathbf{L} \vdash \mathbf{R}; [A^+]}{\mathbf{L} \vdash \mathbf{R}, A^+} \qquad \frac{[A^-]; \mathbf{L} \vdash \mathbf{R}}{A^-, \mathbf{L} \vdash \mathbf{R}}$$

Focus phase

$$\frac{}{\cdot \vdash \cdot; [1]} \qquad \frac{\mathbf{L}_1 \vdash \mathbf{R}_1; [A] \quad \mathbf{L}_2 \vdash \mathbf{R}_2; [B]}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [A \otimes B]} \qquad \frac{[A]; \mathbf{L} \vdash \mathbf{R}}{[A \& B]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{[B]; \mathbf{L} \vdash \mathbf{R}}{[A \& B]; \mathbf{L} \vdash \mathbf{R}} \qquad \text{(no rule for } \top\text{)}$$

$$\text{(no rule for } 0\text{)} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; [A]}{\mathbf{L} \vdash \mathbf{R}; [A \oplus B]} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; [B]}{\mathbf{L} \vdash \mathbf{R}; [A \oplus B]} \qquad \frac{[B]; \mathbf{L}_1 \vdash \mathbf{R}_1 \quad [A]; \mathbf{L}_2 \vdash \mathbf{R}_2}{[A \parr B]; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2} \qquad \frac{}{[\bot]; \cdot \vdash \cdot}$$

Focus → Inversion/Initial

$$\frac{}{X^+ \vdash \cdot; [X^+]} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{\mathbf{L} \vdash \mathbf{R}; [{\downarrow}A]} \qquad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{[{\uparrow}A]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{}{[X^-]; \cdot \vdash X^-}$$

Figure 3.2: A "Σ_3-style" focusing sequent calculus for polarized MALL (fragment)

3.2.3 Focusing proofs, standing back and squinting

We gave some background and intuition in §3.1, but the focusing sequent calculus defined in §3.2.2 may nonetheless appear daunting. Not only have we yet to formally demonstrate the completeness theorem (merely quoting the result of Andreoli [1992]), but we haven't even shown how to derive seemingly trivial theorems, such as the initial sequents

$$A^+; \cdot \vdash A^+ \qquad \text{and} \qquad A^- \vdash \cdot; A^-$$

In the ordinary MALL sequent calculus, it is easy to build a derivation of $A \vdash A$ bottom-up by induction on the structure of A, starting with the left (right) rule if A is positive (negative), and then applying the right (left) rule. For example when $A = B \otimes C$:

$$\dfrac{\dfrac{B \vdash B \quad C \vdash C}{B, C \vdash B \otimes C}}{B \otimes C \vdash B \otimes C}$$

With focusing proofs this approach does not work, because the derivation must finish inverting B and C before applying any right rules. Essentially, the problem is that the syntactic structure of A is too fine a level of granularity for reasoning about focusing proofs.

So let us stand back and try to get a larger view of focusing, taking as an example the composite formula $C = {\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)$. To show C in right-focus, we must begin with one of the following derivations:

$$\dfrac{\dfrac{\mathbf{L}_1 \vdash \mathbf{R}_1; A^-}{\mathbf{L}_1 \vdash \mathbf{R}_1; [{\downarrow}A]} \quad \dfrac{\dfrac{\mathbf{L}_2 \vdash \mathbf{R}_2; B_1^-}{\mathbf{L}_2 \vdash \mathbf{R}_2; [{\downarrow}B_1]}}{\mathbf{L}_2 \vdash \mathbf{R}_2; [{\downarrow}B_1 \oplus {\downarrow}B_2]}}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]} \qquad \dfrac{\dfrac{\mathbf{L}_1 \vdash \mathbf{R}_1; A^-}{\mathbf{L}_1 \vdash \mathbf{R}_1; [{\downarrow}A]} \quad \dfrac{\dfrac{\mathbf{L}_2 \vdash \mathbf{R}_2; B_2^-}{\mathbf{L}_2 \vdash \mathbf{R}_2; [{\downarrow}B_2]}}{\mathbf{L}_2 \vdash \mathbf{R}_2; [{\downarrow}B_1 \oplus {\downarrow}B_2]}}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]}$$

Or in other words, collapsing the intermediate steps, with one of the following derived rules.

$$\dfrac{\mathbf{L}_1 \vdash \mathbf{R}_1; A^- \quad \mathbf{L}_2 \vdash \mathbf{R}_2; B_1^-}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]} \qquad \dfrac{\mathbf{L}_1 \vdash \mathbf{R}_1; A^- \quad \mathbf{L}_2 \vdash \mathbf{R}_2; B_2^-}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]}$$

Note each rule has a pair of invertible sequents as premises, each with a single invertible formula. To show C in left-inversion, we must begin with the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{A^-, B_1^-, \mathbf{L} \vdash \mathbf{R}}{{\downarrow}B_1; A^-, \mathbf{L} \vdash \mathbf{R}} \quad \dfrac{A^-, B_2^-, \mathbf{L} \vdash \mathbf{R}}{{\downarrow}B_2; A^-, \mathbf{L} \vdash \mathbf{R}}}{{\downarrow}B_1 \oplus {\downarrow}B_2; A^-, \mathbf{L} \vdash \mathbf{R}}}{{\downarrow}A, {\downarrow}B_1 \oplus {\downarrow}B_2; \mathbf{L} \vdash \mathbf{R}}}{{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2); \mathbf{L} \vdash \mathbf{R}}$$

Or collapsing the intermediate steps, with the following derived rule:

$$\dfrac{A^-, B_1^-, \mathbf{L} \vdash \mathbf{R} \quad A^-, B_2^-, \mathbf{L} \vdash \mathbf{R}}{{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2); \mathbf{L} \vdash \mathbf{R}}$$

Note the rule has a pair of neutral sequents as premises, each with two formulas in addition to the stable context $\mathbf{L} \vdash \mathbf{R}$. Let us place the derived focus and inversion rules side-by-side:

$$\dfrac{\mathbf{L}_1 \vdash \mathbf{R}_1; A^- \quad \mathbf{L}_2 \vdash \mathbf{R}_2; B_1^-}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]} \qquad \dfrac{A^-, B_1^-, \mathbf{L} \vdash \mathbf{R} \quad A^-, B_2^-, \mathbf{L} \vdash \mathbf{R}}{{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2); \mathbf{L} \vdash \mathbf{R}}$$

$$\dfrac{\mathbf{L}_1 \vdash \mathbf{R}_1; A^- \quad \mathbf{L}_2 \vdash \mathbf{R}_2; B_2^-}{\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]}$$

The symmetry is now obvious, and we can use these rules to easily derive an initial sequent for C, assuming we have initial sequents for its negative subformulas A, B_1, and B_2:

$$\dfrac{\dfrac{\dfrac{A^- \vdash \cdot; A^- \quad B_1^- \vdash \cdot; B_1^-}{A^-, B_1^- \vdash \cdot; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]}}{A^-, B_1^- \vdash {\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)} \quad \dfrac{\dfrac{A^- \vdash \cdot; A^- \quad B_2^- \vdash \cdot; B_2^-}{A^-, B_2^- \vdash \cdot; [{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)]}}{A^-, B_2^- \vdash {\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)}}{{\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2); \cdot \vdash {\downarrow}A \otimes ({\downarrow}B_1 \oplus {\downarrow}B_2)}$$

How do we generalize from this example? The similarity between the derived rules above and Examples 2.1.9 and 2.1.10 from the previous chapter should tip us off: we can reformulate the focus and inversion phases in terms of *patterns*. Where we previously defined proof patterns and refutation patterns, we now define *right patterns* and *left patterns*.

Definition 3.2.3 (Left/right patterns)**.** *A derivation of* $\mathbf{L} \Vdash \mathbf{R}; [A^+]$ *is called a* **right pattern**, *while a derivation of* $[A^-]; \mathbf{L} \Vdash \mathbf{R}$ *is called a* **left pattern**. *Alternatively, these can be called* A-**patterns**. *The contexts* $\mathbf{L}$ *and* $\mathbf{R}$ *in the conclusion of an* A*-pattern are called its* **frame**, *and more specifically* $\mathbf{L}$ *is its* **left-frame**, $\mathbf{R}$ *its* **right-frame**. *The set of all* A*-patterns is called the* **support** *of* A.

As in Chapter 2, we will take a formula to be literally defined by its support. For example, we define positive conjunction and disjunction as follows:

$$\dfrac{}{\cdot \Vdash \cdot; [1]} \qquad \dfrac{\mathbf{L}_1 \Vdash \mathbf{R}_1; [A] \quad \mathbf{L}_2 \Vdash \mathbf{R}_2; [B]}{\mathbf{L}_1, \mathbf{L}_2 \Vdash \mathbf{R}_1, \mathbf{R}_2; [A \otimes B]}$$

$$\text{(no rule for } 0\text{)} \qquad \dfrac{\mathbf{L} \Vdash \mathbf{R}; [A]}{\mathbf{L} \Vdash \mathbf{R}; [A \oplus B]} \qquad \dfrac{\mathbf{L} \Vdash \mathbf{R}; [B]}{\mathbf{L} \Vdash \mathbf{R}; [A \oplus B]}$$

So far these look just like the right-focusing rules in Figure 3.2, only replacing $\vdash$ with $\Vdash$. The difference is that before where we had rules for ending the focus phase and transitioning to inversion, here we instead give axioms for building right patterns:

$$\dfrac{}{X^+ \Vdash \cdot; [X^+]} \qquad \dfrac{}{A^- \Vdash \cdot; [{\downarrow}A]} \qquad \dfrac{}{\cdot \Vdash A^+; [\overset{+}{\neg} A]}$$

In particular, note that where the focusing rules for ${\downarrow}A$ and $\overset{+}{\neg} A$ each had a single premise with the formula A in right- or left-inversion, respectively, the pattern rules for these connectives place the formula A in the left-frame and right-frame, respectively.

Similarly, we define negative conjunction and disjunction with left pattern rules that look just like their left-focusing rules:

$$\text{(no rule for } \top\text{)} \qquad \dfrac{[A]; \mathbf{L} \Vdash \mathbf{R}}{[A \& B]; \mathbf{L} \Vdash \mathbf{R}} \qquad \dfrac{[B]; \mathbf{L} \Vdash \mathbf{R}}{[A \& B]; \mathbf{L} \Vdash \mathbf{R}}$$

$$\dfrac{}{[\bot]; \cdot \Vdash \cdot} \qquad \dfrac{[A]; \mathbf{L}_1 \Vdash \mathbf{R}_1 \quad [B]; \mathbf{L}_1 \Vdash \mathbf{R}_2}{[A \invamp B]; \mathbf{L}_1, \mathbf{L}_2 \Vdash \mathbf{R}_1, \mathbf{R}_2}$$

But we define atomic propositions, $\uparrow$ and $\urcorner$ by axioms:

$$\frac{}{[X^-]; \cdot \Vdash X^-} \qquad \frac{}{[\uparrow A]; \cdot \Vdash A^+} \quad \frac{}{[\urcorner A]; A^- \Vdash \cdot}$$

How do we derive the focus and inversion rules from these patterns?

It is helpful to first define some more "punctuation". We introduce a new sequent form $\bigotimes\!\!\!\!\raisebox{0pt}{}\ \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$, which is expanded by multiplicative rules:

$$\frac{⅋\, \mathbf{R}_1'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}_1' \quad ⅋\, \mathbf{R}_2'; \mathbf{L}_2 \vdash \mathbf{R}_2; \bigotimes \mathbf{L}_2'}{⅋\, \mathbf{R}_1', \mathbf{R}_2'; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; \bigotimes \mathbf{L}_1', \mathbf{L}_2'} \qquad \frac{}{⅋\, \cdot; \cdot \vdash \cdot; \bigotimes \cdot}$$

$$\frac{}{⅋\, \cdot; X^+ \vdash \cdot; \bigotimes X^+} \qquad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{⅋\, A^+; \mathbf{L} \vdash \mathbf{R}; \bigotimes \cdot} \qquad \frac{}{⅋\, X^-; \cdot \vdash X^-; \bigotimes \cdot} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{⅋\, \cdot; \mathbf{L} \vdash \mathbf{R}; \bigotimes A^-}$$

Now, the rules for deriving left- and right-focused sequents can be expressed concisely:

$$\frac{\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \quad ⅋\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'}{\mathbf{L} \vdash \mathbf{R}; [A^+]} \qquad \frac{[A^-]; \mathbf{L}' \Vdash \mathbf{R}' \quad ⅋\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'}{[A^-]; \mathbf{L} \vdash \mathbf{R}}$$

As can the rules for left- and right-inversion:

$$\frac{[A^-]; \mathbf{L}' \Vdash \mathbf{R}' \quad \longrightarrow \quad \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'}{\mathbf{L} \vdash \mathbf{R}; A^-} \quad \frac{\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \quad \longrightarrow \quad \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'}{A^+; \mathbf{L} \vdash \mathbf{R}}$$

Again, the notation $- \longrightarrow -$ expresses that for every derivation of the judgment on the left, the judgment on the right is derivable. So for example, the right-inversion rule says that to derive $\mathbf{L} \vdash \mathbf{R}; A^-$, we must derive a set of neutral sequents $\mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'$, for every $\mathbf{L}'$ and $\mathbf{R}'$ forming the left- and right-frame of an A-pattern.

Example 3.2.4. Let $C = \downarrow A \otimes (\downarrow B_1 \oplus \downarrow B_2)$. By instantiating these rules with the two patterns for C (derivations of $A^-, B_1^- \Vdash \cdot; [C]$ and $A^-, B_2^- \Vdash \cdot; [C]$), we obtain exactly the focus and inversion rules derived above. ■

This completes the pattern-based reformulation of focusing proofs, summarized in Figures 3.3 and 3.4. We call this a "reformulation" because it does not essentially change the structure of proofs, except by collapsing multiple steps in the focus and inversion phases.

Notation. *We associate a derivation $\mathcal{D}$ with a judgment $\mathcal{J}$ either by writing the the derivation over the judgment $\overset{\mathcal{D}}{\mathcal{J}}$, or by separating them with a double-colon $\mathcal{D} :: \mathcal{J}$.*

Lemma 3.2.5 (Focus phase collapse). *We can interpret sequents $⅋\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ in the Σ_3-style system as above, by expanding it out with multiplicative rules. Then there is a Σ_3-style proof $\mathcal{D} :: (\mathbf{L} \vdash \mathbf{R}; [A^+])$ (respectively $[A^-]; \mathbf{L} \vdash \mathbf{R}$) iff there exists $\mathbf{L}' \Vdash \mathbf{R}'; [A^+]$ (resp. $[A^-]; \mathbf{L}' \Vdash \mathbf{R}'$) and a Σ_3-style proof of $⅋\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ built out of subderivations of $\mathcal{D}$.*

Proof. By induction on A. □

Lemma 3.2.6 (Inversion phase collapse). *Let $L = A_1^+, \ldots, A_m^+$, $R = A_{m+1}^-, \ldots, A_{m+n}^-$. There is a Σ_3-style proof $\mathcal{D} :: (L; \mathbf{L} \vdash \mathbf{R}; R)$ iff for any $\mathbf{L}_1 \Vdash \mathbf{R}_1; [A_1^+], \ldots, \mathbf{L}_m \Vdash \mathbf{R}_m; [A_m^+]$, and $[A_{m+1}^-]; \mathbf{L}_{m+1} \Vdash \mathbf{R}_{m+1}$, ..., $[A_{m+n}^-]; \mathbf{L}_{m+n} \Vdash \mathbf{R}_{m+n}$, there is a proof $\mathcal{E} :: (\mathbf{L}_{m+n}, \ldots, \mathbf{L}_1, \mathbf{L} \vdash \mathbf{R}, \mathbf{R}_1, \ldots, \mathbf{R}_{m+n})$, where $\mathcal{E}$ is a subderivation of $\mathcal{D}$.*

$$\frac{}{\cdot \Vdash \cdot; [1]} \qquad \frac{\mathbf{L}_1 \Vdash \mathbf{R}_1; [A] \quad \mathbf{L}_2 \Vdash \mathbf{R}_2; [B]}{\mathbf{L}_1, \mathbf{L}_2 \Vdash \mathbf{R}_1, \mathbf{R}_2; [A \otimes B]}$$

$$\text{(no rule for 0)} \qquad \frac{\mathbf{L} \Vdash \mathbf{R}; [A]}{\mathbf{L} \Vdash \mathbf{R}; [A \oplus B]} \qquad \frac{\mathbf{L} \Vdash \mathbf{R}; [B]}{\mathbf{L} \Vdash \mathbf{R}; [A \oplus B]}$$

$$X^+ \Vdash \cdot; [X^+] \qquad A^- \Vdash \cdot; [{\downarrow}A] \qquad \cdot \Vdash A^+; [\overset{+}{\neg} A]$$

. . . .

$$\text{(no rule for } \top\text{)} \qquad \frac{[A]; \mathbf{L} \Vdash \mathbf{R}}{[A \& B]; \mathbf{L} \Vdash \mathbf{R}} \qquad \frac{[B]; \mathbf{L} \Vdash \mathbf{R}}{[A \& B]; \mathbf{L} \Vdash \mathbf{R}}$$

$$\frac{}{[\bot]; \cdot \Vdash \cdot} \qquad \frac{[A]; \mathbf{L}_1 \Vdash \mathbf{R}_1 \quad [B]; \mathbf{L}_1 \Vdash \mathbf{R}_2}{[A \parr B]; \mathbf{L}_1, \mathbf{L}_2 \Vdash \mathbf{R}_1, \mathbf{R}_2}$$

$$[X^-]; \cdot \Vdash X^- \qquad [{\uparrow}A]; \cdot \Vdash A^+ \qquad [\overset{-}{\neg} A]; A^- \Vdash \cdot$$

Figure 3.3: Definition of some polarized MALL connectives by patterns

$\boxed{\text{Sequents}}$

Neutral	$\mathbf{L} \vdash \mathbf{R}$
Right-focus	$\mathbf{L} \vdash \mathbf{R}; [A^+]$
Left-inversion	$A^+; \mathbf{L} \vdash \mathbf{R}$
Right-inversion	$\mathbf{L} \vdash \mathbf{R}; A^-$
Left-focus	$[A^+]; \mathbf{L} \vdash \mathbf{R}$
Multiplicative	$\parr\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$

. . . .

$$\frac{\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \quad \parr\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'}{\mathbf{L} \vdash \mathbf{R}; [A^+]} \qquad \frac{\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \quad \longrightarrow \quad \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'}{A^+; \mathbf{L} \vdash \mathbf{R}}$$

$$\frac{[A^-]; \mathbf{L}' \Vdash \mathbf{R}' \quad \longrightarrow \quad \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'}{\mathbf{L} \vdash \mathbf{R}; A^-} \qquad \frac{[A^-]; \mathbf{L}' \Vdash \mathbf{R}' \quad \parr\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'}{[A^-]; \mathbf{L} \vdash \mathbf{R}}$$

$$\frac{\parr\, \mathbf{R}_1'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}_1' \quad \parr\, \mathbf{R}_2'; \mathbf{L}_2 \vdash \mathbf{R}_2; \bigotimes \mathbf{L}_2'}{\parr\, \mathbf{R}_1', \mathbf{R}_2'; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; \bigotimes \mathbf{L}_1', \mathbf{L}_2'} \qquad \frac{}{\parr\, \cdot; \cdot \vdash \cdot; \bigotimes \cdot}$$

$$\frac{}{\parr\, \cdot; X^+ \vdash \cdot; \bigotimes X^+} \qquad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{\parr\, A^+; \mathbf{L} \vdash \mathbf{R}; \bigotimes \cdot} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{\parr\, \cdot; \mathbf{L} \vdash \mathbf{R}; \bigotimes A^-} \qquad \frac{}{\parr\, X^-; \cdot \vdash X^-; \bigotimes \cdot}$$

$$\frac{\mathbf{L} \vdash \mathbf{R}; [A^+]}{\mathbf{L} \vdash \mathbf{R}, A^+} \qquad \frac{[A^-]; \mathbf{L} \vdash \mathbf{R}}{A^-, \mathbf{L} \vdash \mathbf{R}}$$

Figure 3.4: A pattern-based formulation of MALL focusing proofs

Proof. By induction on L and R. $\square$

Corollary 3.2.7 (Equivalence of focusing calculi)**.** *Writing* $\vdash_{[\ell *]}$ *for provability in the pattern-based formulation, we have:*

1. $\mathbf{L} \vdash_{[\ell *]} \mathbf{R}; [A^+]$ *iff* $\mathbf{L} \vdash_{[\ell]} \mathbf{R}; [A^+]$
2. $A^+; \mathbf{L} \vdash_{[\ell *]} \mathbf{R}$ *iff* $A^+; \mathbf{L} \vdash_{[\ell]} \mathbf{R}$
3. $\mathbf{L} \vdash_{[\ell *]} \mathbf{R}; A^-$ *iff* $\mathbf{L} \vdash_{[\ell]} \mathbf{R}; A^-$
4. $[A^-]; \mathbf{L} \vdash_{[\ell *]} \mathbf{R}$ *iff* $[A^-]; \mathbf{L} \vdash_{[\ell]} \mathbf{R}$
5. $\mathbf{L} \vdash_{[\ell *]} \mathbf{R}$ *iff* $\mathbf{L} \vdash_{[\ell]} \mathbf{R}$

Proof. By induction on derivations, applying Lemmas 3.2.5 and 3.2.6. $\square$

Corollary 3.2.8 (Soundness of pattern-based formulation)**.**

1. *If* $\mathbf{L} \vdash_{[\ell *]} \mathbf{R}; [A^+]$ *then* $\mathbf{L} \vdash_{\ell} \mathbf{R}, A^+$
2. *If* $[A^-]; \mathbf{L} \vdash_{[\ell *]} \mathbf{R}$ *then* $A^-, \mathbf{L} \vdash_{\ell} \mathbf{R}$
3. *If* $\mathbf{L} \vdash_{[\ell *]} \mathbf{R}; A^-$ *then* $\mathbf{L} \vdash_{\ell} \mathbf{R}, A^-$
4. *If* $A^+; \mathbf{L} \vdash_{[\ell *]} \mathbf{R}$ *then* $A^+, \mathbf{L} \vdash_{\ell} \mathbf{R}$
5. *If* $\mathbf{L} \vdash_{[\ell *]} \mathbf{R}$ *then* $\mathbf{L} \vdash_{\ell} \mathbf{R}$
6. *If* $\bigparr \mathbf{R}'; \mathbf{L} \vdash_{[\ell *]} \mathbf{R}; \bigotimes \mathbf{L}'$ *then* $(\bigparr \mathbf{R}'), \mathbf{L} \vdash_{\ell} \mathbf{R}, (\bigotimes \mathbf{L}')$

Proof. (1)-(5) follow by composing Corollary 3.2.7 with Theorem 3.2.2. (6) reduces to (3) and (4) by expanding $\bigparr \mathbf{R}'; \mathbf{L} \vdash_{[\ell *]} \mathbf{R}; \bigotimes \mathbf{L}'$ into a list of sequents, applying soundness, and then recombining the MALL proofs with a series of $\parr L$ and $\otimes R$ rules to obtain $(\bigparr \mathbf{R}'), \mathbf{L} \vdash_{\ell} \mathbf{R}, (\bigotimes \mathbf{L}')$. $\square$

3.2.4 Focusing proofs are canonical derivations

While the correspondence between Figures 3.2 and 3.4 is fairly direct, what should be even more striking is the correspondence between Figure 3.4 and the linear rules of Figure 2.9. Indeed, there is a trivial syntactic isomorphism between them. For any stable context $\mathbf{L} \vdash \mathbf{R}$, we can build a corresponding simple frame $\Delta_{\mathbf{L}\vdash\mathbf{R}}$:

$$\Delta_{\mathbf{L}\vdash\mathbf{R}} = \{A^- \mid A^- \in \mathbf{L}\} \cup \{X^+ \mid X^+ \in \mathbf{L}\} \cup \{\bullet A^+ \mid A^+ \in \mathbf{R}\} \cup \{\bullet X^- \mid X^- \in \mathbf{R}\}$$

And conversely, given any simple frame Δ, we can build the stable context $\mathbf{L}_\Delta \vdash \mathbf{R}_\Delta$:

$$\begin{aligned} \mathbf{L}_\Delta &= \{A^- \mid A^- \in \Delta\} \cup \{X^+ \mid X^+ \in \Delta\} \\ \mathbf{R}_\Delta &= \{A^+ \mid \bullet A^+ \in \Delta\} \cup \{X^- \mid \bullet X^- \in \Delta\} \end{aligned}$$

We say that a stable context $\mathbf{L} \vdash \mathbf{R}$ and a frame of simple hypotheses Δ are **interchangeable** (written $(\mathbf{L} \vdash \mathbf{R}) \longleftrightarrow \Delta$) when $\Delta = \Delta_{\mathbf{L}\vdash\mathbf{R}}$, $\mathbf{L} = \mathbf{L}_\Delta$, $\mathbf{R} = \mathbf{R}_\Delta$. We extend this convention to contexts of simple hypotheses, writing $(\mathbf{L} \vdash \mathbf{R}) \longleftrightarrow \Gamma$, by viewing Γ as a frame.

Now, let $(\mathbf{L} \vdash \mathbf{R}) \longleftrightarrow \Gamma$ and $(\mathbf{L}' \vdash \mathbf{R}') \longleftrightarrow \Delta$ be interchangeable. We say that judgments are interchangeable as follows:

$$\begin{array}{cc}
\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \longleftrightarrow \Delta \Vdash A^+ & [A^-]; \mathbf{L}' \Vdash \mathbf{R}' \longleftrightarrow \Delta \Vdash \bullet A^- \\
\mathbf{L} \vdash \mathbf{R}; [A^+] \longleftrightarrow \Gamma \vdash A^+ & A^+; \mathbf{L} \vdash \mathbf{R} \longleftrightarrow \Gamma \vdash \bullet A^+ \\
\mathbf{L} \vdash \mathbf{R}; A^- \longleftrightarrow \Gamma \vdash A^- & [A^-]; \mathbf{L} \vdash \mathbf{R} \longleftrightarrow \Gamma \vdash \bullet A^- \\
\bigparr \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}' \longleftrightarrow \Gamma \vdash \Delta & \mathbf{L} \vdash \mathbf{R} \longleftrightarrow \Gamma \vdash \#
\end{array}$$

Under this translation guide, the rules for building MALL focusing proofs are identical to the rules for building linear canonical derivations. As a corollary, the identity and composition principles for linear canonical derivations can be translated into initial sequents and cuts on focusing proofs.

Theorem 3.2.9 (Initial sequents). *The following initial sequents are derivable in* $\vdash_{[\ell*]}$*:*

1. $A^+; \cdot \vdash A^+$
2. $A^- \vdash \cdot; A^-$
3. $X^+ \vdash \cdot; [X^+]$
4. $[X^-]; \cdot \vdash X^-$

Theorem 3.2.10 (Cut-admissibility). *The following cuts are admissible in* $\vdash_{[\ell*]}$*:*

1. *If* $\mathbf{L}_1 \vdash \mathbf{R}_1; [A^+]$ *and* $A^+; \mathbf{L}_2 \vdash \mathbf{R}_2$ *then* $\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2$
2. *If* $\mathbf{L}_1 \vdash \mathbf{R}_1; A^-$ *and* $[A^-]; \mathbf{L}_2 \vdash \mathbf{R}_2$ *then* $\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2$
3. *If* $\bigparr \mathbf{R}'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}'$ *and* $\mathbf{L}', \mathbf{L}_2 \vdash \mathbf{R}_2, \mathbf{R}'; [A^+]$ *then* $\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; [A^+]$
4. *If* $\bigparr \mathbf{R}'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}'$ *and* $[A^-]; \mathbf{L}', \mathbf{L}_2 \vdash \mathbf{R}_2, \mathbf{R}'$ *then* $[A^-]; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2$
5. *If* $\bigparr \mathbf{R}'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}'$ *and* $\mathbf{L}', \mathbf{L}_2 \vdash \mathbf{R}_2, \mathbf{R}'; A^-$ *then* $\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; A^-$
6. *If* $\bigparr \mathbf{R}'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}'$ *and* $A^+; \mathbf{L}', \mathbf{L}_2 \vdash \mathbf{R}_2, \mathbf{R}'$ *then* $A^+; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2$
7. *If* $\bigparr \mathbf{R}'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}'$ *and* $\bigparr \mathbf{R}''; \mathbf{L}', \mathbf{L}_2 \vdash \mathbf{R}_2, \mathbf{R}'; \bigotimes \mathbf{L}''$ *then* $\bigparr \mathbf{R}''; \mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2; \bigotimes \mathbf{L}''$
8. *If* $\bigparr \mathbf{R}'; \mathbf{L}_1 \vdash \mathbf{R}_1; \bigotimes \mathbf{L}'$ *and* $\mathbf{L}', \mathbf{L}_2 \vdash \mathbf{R}_2, \mathbf{R}'$ *then* $\mathbf{L}_1, \mathbf{L}_2 \vdash \mathbf{R}_1, \mathbf{R}_2$

Proof. These are all instances of the identity and composition principles for linear canonical derivations, under the translation guide. □

The translation guide, in effect, says that focusing proofs and canonical derivations are the same thing. On the other hand, the long list of cut principles in Theorem 3.4.8 gives a suggestion as to why we introduced the notion of canonical derivations in the first place, beyond any philosophical motivation: it is simply a better notation. Cuts (3–8) are all instances of a single composition principle:

Composition (substitution). *If* $\Gamma_1(\Delta) \vdash J$ *and* $\Gamma_2 \vdash \Delta$ *then* $\Gamma_1(\Gamma_2) \vdash J$

3.2.5 Complex hypotheses and weak focalization

If canonical derivations in a context of simple hypotheses correspond to focusing proofs, what happens when we add complex hypotheses? Following the convention we established above, complex proof hypotheses $A^+ \in \Gamma$ should become positive formulas on the left side of a sequent, while complex refutation hypotheses $\bullet A^- \in \Gamma$ should become negative formulas on the right side. But such sequents are not stable! In other words, contexts of arbitrary hypotheses correspond to arbitrary sequents. If we don't care about separating simple from complex hypotheses, we can state the criteria for interchangeability $(\mathfrak{L} \vdash \mathfrak{R}) \longleftrightarrow \Gamma$ more concisely:

$$\Gamma_{\mathfrak{L} \vdash \mathfrak{R}} = \mathfrak{L}, \bullet\mathfrak{R}$$

$$\mathfrak{L}_\Gamma = \{A \mid A \in \Gamma\} \qquad \mathfrak{R}_\Gamma = \{A \mid \bullet A \in \Gamma\}$$

Now, as we discussed, the first step in a bottom-up search for a focusing proof of $L; \mathbf{L} \vdash \mathbf{R}; R$ is to eagerly invert the formulas in L and R, creating a set of stable sequents as goals. But the rules for using complex hypotheses do not *force* such a discipline. Recall the rules:

$$\frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(A^+) \vdash J} \qquad \frac{\Delta \Vdash \bullet A^- \quad \longrightarrow \quad \Gamma(\Delta) \vdash J}{\Gamma(\bullet A^-) \vdash J}$$

We are allowed to retain complex hypotheses as long we wish and invert them at any stage—even while another formula is in focus. Therefore, in the terminology of Laurent [2004a], canonical derivations with complex hypotheses correspond to proofs that are *weakly focalized*, as opposed to fully focalized. On the other hand, Laurent's observation was that a weakly focalized proof can be easily converted into a fully focalized one, precisely because invertible rules are invertible. In other words, a canonical derivation with complex hypotheses can be trivially converted into one with only simple hypotheses, by eagerly applying pattern substitution (Prop. 2.1.14).

3.3 Completeness of focusing proofs

In this section we prove the completeness of focusing, using the interpretation as canonical derivations. Our proof is similar in structure to that of Laurent [2004a], who simplified Andreoli's original proof [1992]. We prove a weak focalization lemma, and then show this implies full focusing.

Lemma 3.3.1 (Weak focalization). *Let $(\mathfrak{L} \vdash \mathfrak{R}) \longleftrightarrow \Gamma$. Then $\mathfrak{L} \vdash_\ell \mathfrak{R}$ implies there is a linear canonical derivation of $\Gamma \vdash \#$.*

Proof. By induction on the MALL sequent calculus proof. Without loss of generality, we can assume the MALL derivation does not have any uses of *cut*, and that *init* is restricted to atomic formulas. For atomic initial sequents, we directly apply the identity principle for linear canonical derivations. Otherwise, the derivation ends in a logical rule.

There are essentially two kinds of cases: the rule introduces a formula that is either in the inverting context ($A^+ \in \mathfrak{L}$ or A^- in $\mathfrak{R}$) or in the stable context ($A^+ \in \mathfrak{R}$ or $A^- \in \mathfrak{L}$). The negative polarity cases are dual to positive polarity, so we show here only some illustrative examples where a positive formula is introduced on the left (invertible) or right (stable).

- **Case** $\otimes R$: The proof ends in $\dfrac{\mathfrak{L}_1 \vdash \mathfrak{R}_1, A \quad \mathfrak{L}_2 \vdash \mathfrak{R}_2, B}{\mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{L}_2, A \otimes B}$, where $(\mathfrak{L}_1 \vdash \mathfrak{R}_1) \longleftrightarrow \Gamma_1$ and $(\mathfrak{L}_2 \vdash \mathfrak{R}_2) \longleftrightarrow \Gamma_2$. By the induction hypothesis, there exist linear canonical derivations (lcds) $C_1 :: (\Gamma_1, \bullet A \vdash \#)$ and $C_2 :: (\Gamma_2, \bullet B \vdash \#)$. Now, note that for any patterns $p_1 :: (\Delta_1 \Vdash A)$ and $p_2 :: (\Delta_2 \Vdash B)$, we can build a derivation $C_{(p_1,p_2)} :: (\bullet A \otimes B, \Delta_1, \Delta_2 \vdash \#)$ as follows:

$$C_{(p_1,p_2)} = \dfrac{\dfrac{\dfrac{\overset{p_1}{\Delta_1 \Vdash A} \quad \overset{p_2}{\Delta_2 \Vdash B}}{\Delta_1, \Delta_2 \Vdash A \otimes B} \quad \overset{Id}{\Delta_1, \Delta_2 \vdash \Delta_1, \Delta_2}}{\Delta_1, \Delta_2 \vdash A \otimes B}}{\bullet A \otimes B, \Delta_1, \Delta_2 \vdash \#}$$

  Then we derive $\Gamma_1, \Gamma_2, \bullet A \otimes B \vdash \#$ with

$$\dfrac{\overset{C_1}{\Gamma_1, \bullet A \vdash \#} \quad \dfrac{\overset{p_1}{\Delta_1 \Vdash A} \quad \longrightarrow \quad \dfrac{\overset{C_2}{\Gamma_2, \bullet B \vdash \#} \quad \dfrac{\overset{p_2}{\Delta_2 \Vdash B} \quad \longrightarrow \quad \overset{C_{(p_1,p_2)}}{\bullet A \otimes B, \Delta_1, \Delta_2 \vdash \#}}{\bullet A \otimes B, \Delta_1 \vdash \bullet B}}{\Gamma_2, \bullet A \otimes B, \Delta_1 \vdash \#}\,\dagger}{\Gamma_2, \bullet A \otimes B \vdash \bullet A}}{\Gamma_1, \Gamma_2, \bullet A \otimes B \vdash \#}\,\dagger$$

  where ($\dagger$) indicates uses of the composition principle (the right derivation being substituted into the left). Observe that the derivation here is a bit arbitrary: we could just as well use C_1 and C_2 in the opposite order.

- **Case** $\oplus R$: The proof ends in either $\dfrac{\mathfrak{L} \vdash \mathfrak{R}, A}{\mathfrak{L} \vdash \mathfrak{R}, A \oplus B}$ or $\dfrac{\mathfrak{L} \vdash \mathfrak{R}, B}{\mathfrak{L} \vdash \mathfrak{R}, A \oplus B}$, where $(\mathfrak{L} \vdash \mathfrak{R}) \longleftrightarrow \Gamma$. The two cases are symmetric, so assume the former, and by the i.h. there exists an lcd $C :: (\Gamma, \bullet A \vdash \#)$. Analogously to (case $\otimes$), we first note that for any $p :: (\Delta \Vdash A)$, there exists a derivation $C_{\mathsf{inl}\,p} :: (\bullet A \oplus B, \Delta \vdash \#)$:

$$C_{\mathsf{inl}\,p} = \dfrac{\dfrac{\dfrac{\overset{p}{\Delta \Vdash A}}{\Delta \Vdash A \oplus B} \quad \overset{Id}{\Delta \vdash \Delta}}{\Delta \vdash A \oplus B}}{\bullet A \oplus B, \Delta \vdash \#}$$

  Then we construct

$$\dfrac{\overset{C}{\Gamma, \bullet A \vdash \#} \quad \dfrac{\overset{p}{\Delta \Vdash A} \quad \longrightarrow \quad \overset{C_{\mathsf{inl}\,p}}{\bullet A \oplus B, \Delta \vdash \#}}{\bullet A \oplus B \vdash \bullet A}}{\Gamma, \bullet A \oplus B \vdash \#}\,\dagger$$

- **Case** $\downarrow R$: The proof ends in $\dfrac{\mathfrak{L} \vdash \mathfrak{R}, A}{\mathfrak{L} \vdash \mathfrak{R}, \downarrow A}$, where $(\mathfrak{L} \vdash \mathfrak{R}) \longleftrightarrow \Gamma$. By the i.h., there exists an lcd $C :: (\Gamma, \bullet A^-)$. Then we construct:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}C\\ \Gamma, \bullet A^- \vdash \#\end{array}}{\Gamma \vdash A^-}\dagger_2}{\Gamma \vdash {\downarrow} A}\dagger_1}{\Gamma, \bullet {\downarrow} A \vdash \#}$$

using the derived rule $\dagger_1$ for proving ${\downarrow}A$ (Prop. 2.3.1), and the admissible step $\dagger_2$ introducing a complex hypothesis.

- **Case** $\otimes L$: By the i.h., there is an lcd $C :: (\Gamma, A^+, B^+ \vdash \#)$. We immediately derive $\Gamma, A \otimes B \vdash \#$ using the analysis rule, by noting that every $A \otimes B$-pattern decomposes as a pair of an A-pattern and a B-pattern, and applying pattern substitution for each hypothesis.
- **Case** ${\downarrow}L$: By the i.h., there is an lcd $C :: (\Gamma, A^- \vdash \#)$. We immediately derive $\Gamma, {\downarrow}A \vdash \#$ by applying the analysis rule, noting that there is only a single ${\downarrow}A$-pattern, with frame A^-.

□

Corollary 3.3.2 (Focusing completeness)**.** *If $\mathfrak{L} \vdash \mathfrak{R}$ has a MALL proof, then it has a fully focusing proof.*

Proof. Although we introduced complex hypotheses in the proof of weak focalization (e.g., in cases $({\downarrow}R)$ and $(\otimes L)$), these were admissible steps, rather than canonical rules. The canonical derivation that results from weak focalization never introduces complex hypotheses—it only decomposes them—and so by pattern substitution, we can perform this decomposition as the first (bottom-up) step of a fully focusing proof. □

Corollary 3.3.3. *Weak entailment for linear canonical derivations coincides with MALL entailment, i.e., $A^+ \leq^- B^+$ (or $A^- \leq^+ B^-$) iff $A \vdash_\ell B$.*

Proof. In the positive case, by definition, we have $A^+ \leq^- B^+$ iff there is a lcd of $\bullet B^+ \vdash \bullet A^+$, which is equivalent to $\bullet B^+, A^+ \vdash \#$. This is true if (Lemma 3.3.1) and only if (Corollary 3.2.8) $A^+ \vdash_\ell B^+$. Likewise, in the negative case, by definition $A^- \leq^+ B^-$ iff there is a lcd of $A^- \vdash B^-$, which is equivalent to $\bullet B^-, A^+ \vdash \#$, and holds if and only if $A^- \vdash_\ell B^-$. □

3.4 Unrestricted derivations and classical sequent calculus

We have seen the close correspondence between linear canonical derivations and Andreoli's focusing proofs for MALL, and examined the relationship with MALL itself via soundness and completeness theorems. Now, we will show how unrestricted canonical derivations are in the same sort of relationship with classical logic.

3.4.1 Polarizations of classical logic

To get to this result as quickly as possible, we will take as our axiomatization of classical logic the polarized MALL sequent calculus, together with explicit structural rules of weakening and contraction:

$$\frac{\mathfrak{L} \vdash \mathfrak{R}}{A, \mathfrak{L} \vdash \mathfrak{R}}\,WL \qquad \frac{A, A, \mathfrak{L} \vdash \mathfrak{R}}{A, \mathfrak{L} \vdash \mathfrak{R}}\,CL \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}, A, A}{\mathfrak{L} \vdash \mathfrak{R}, A}\,CR \qquad \frac{\mathfrak{L} \vdash \mathfrak{R}}{\mathfrak{L} \vdash \mathfrak{R}, A}\,WR$$

We write $\vdash_c$ for provability in this sequent calculus. As is well-known, this axiomatization contains a lot of redundancy. For example, the two forms of conjunction $\otimes$ and $\&$ are logically equivalent, as are the two forms of disjunction.[4] The different negations ($\overset{+}{\neg}$, $\overset{-}{\neg}$) are also logically equivalent, although they already were in MALL. Formally, we are working with formulas in the syntax of PPL (cf. Definition 2.3.4), which correspond to different *polarizations* of classical propositions. To make this precise, let us use letters a, b, c to range over formulas of propositional logic, built out of atoms and the connectives $\mathrm{T}, \mathrm{F}, \wedge, \vee, \sim$.

Definition 3.4.1 (Polarization)**.** *Let $|-|$ be the map*

$$|1| = |\top| = \mathrm{T} \qquad |0| = |\bot| = \mathrm{F} \qquad |X| = X$$
$$|A \otimes B| = |A \& B| = |A| \wedge |B| \qquad |A \oplus B| = |A ⅋ B| = |A| \vee |B|$$
$$|\neg A| = \sim |A| \qquad |\updownarrow A| = |A|$$
$$|A \to B| = \sim |A| \vee |B| \qquad |A - B| = |A| \wedge \sim |B|$$

from PPL formulas to formulas of propositional logic. A **polarization** *of b is a PPL formula A such that b and $|A|$ are classically equivalent. We extend the terminology pointwise to sequents of formulas.*

Example 3.4.2. Among the (infinitely many) polarizations of $X \wedge Y$ are:

$$X^+ \otimes Y^+ \quad \downarrow X^- \otimes Y^+ \quad X^- \& \uparrow Y^+ \quad \downarrow\uparrow X^+ \otimes Y^+ \quad \Uparrow(X^+ \otimes Y^+) \quad \ldots$$

■

Proposition 3.4.3. *Let $a_1, \ldots, a_m \vdash b_1, \ldots, b_n$ be a classical sequent, and $\mathfrak{L} \vdash \mathfrak{R}$ be a polarization. Then $a_1, \ldots, a_m \vdash b_1, \ldots, b_n$ is classically true iff $\mathfrak{L} \vdash_c \mathfrak{R}$.*

Proof. Standard, reading the rules under the map $|-|$. □

Proposition 3.4.3 says that we can treat a classical formula and its polarization into PPL as interchangeable in terms of provability. In terms of proof search, however, we can view polarization as committing to a particular *focusing strategy* for proving the classical formula. The soundness and completeness theorems will tell us that all strategies are acceptable.

3.4.2 Focusing proofs for classical sequent calculus

As in §3.2.2 and §3.2.3, we give two essentially equivalent formulations of focusing proofs: one in the style of Andreoli's Σ_3 (Figure 3.5) and the other in terms of patterns (Figure 3.6). The Σ_3-style calculus for PPL is almost identical to the one for MALL (recall Figure 3.2), except in the following ways:

1. In the rules for focusing on a particular formula (group "**Neutral** $\to$ **Focus**"), the focus formula is retained inside the stable context.

2. In the focusing rules for $\otimes$ and ⅋, the entire context is passed to both premises, rather than a nondeterministic splitting.

[4]Since formulas must be well-polarized, we write these equivalences as a pair of equivalences, e.g., $\downarrow A \otimes \downarrow B \equiv \Uparrow(A \& B)$ and $\Downarrow(A \otimes B) \equiv \uparrow A \& \uparrow B$.

Inversion phase

(as in Figure 3.2)

Inversion → Neutral

(as in Figure 3.2)

Neutral → Focus

$$\frac{A^+ \in \mathbf{R} \quad \mathbf{L} \vdash \mathbf{R}; [A^+]}{\mathbf{L} \vdash \mathbf{R}} \qquad \frac{A^- \in \mathbf{L} \quad [A^-]; \mathbf{L} \vdash \mathbf{R}}{\mathbf{L} \vdash \mathbf{R}}$$

Focus phase

$$\frac{}{\mathbf{L} \vdash \mathbf{R}; [1]} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; [A] \quad \mathbf{L} \vdash \mathbf{R}; [B]}{\mathbf{L} \vdash \mathbf{R}; [A \otimes B]} \qquad \frac{[A]; \mathbf{L} \vdash \mathbf{R}}{[A \& B]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{[B]; \mathbf{L} \vdash \mathbf{R}}{[A \& B]; \mathbf{L} \vdash \mathbf{R}} \quad \text{(no rule for } \top\text{)}$$

$$\text{(no rule for } 0\text{)} \quad \frac{\mathbf{L} \vdash \mathbf{R}; [A]}{\mathbf{L} \vdash \mathbf{R}; [A \oplus B]} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; [B]}{\mathbf{L} \vdash \mathbf{R}; [A \oplus B]} \qquad \frac{[B]; \mathbf{L} \vdash \mathbf{R} \quad [A]; \mathbf{L} \vdash \mathbf{R}}{[A \parr B]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{}{[\bot]; \mathbf{L} \vdash \mathbf{R}}$$

Focus → Inversion/Initial

$$\frac{X^+ \in \mathbf{L}}{\mathbf{L} \vdash \mathbf{R}; [X^+]} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{\mathbf{L} \vdash \mathbf{R}; [{\downarrow}A]} \qquad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{[{\uparrow}A]; \mathbf{L} \vdash \mathbf{R}} \qquad \frac{X^- \in \mathbf{R}}{[X^-]; \mathbf{L} \vdash \mathbf{R}}$$

Figure 3.5: A "Σ_3-style" focusing sequent calculus for PPL

$$\frac{\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \quad \parr\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'}{\mathbf{L} \vdash \mathbf{R}; [A^+]} \qquad \frac{\mathbf{L}' \Vdash \mathbf{R}'; [A^+] \quad \longrightarrow \quad \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'}{A^+; \mathbf{L} \vdash \mathbf{R}}$$

$$\frac{[A^-]; \mathbf{L}' \Vdash \mathbf{R}' \quad \longrightarrow \quad \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'}{\mathbf{L} \vdash \mathbf{R}; A^-} \qquad \frac{[A^-]; \mathbf{L}' \Vdash \mathbf{R}' \quad \parr\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'}{[A^-]; \mathbf{L} \vdash \mathbf{R}}$$

$$\frac{\parr\, \mathbf{R}'_1; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'_1 \quad \parr\, \mathbf{R}'_2; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'_2}{\parr\, \mathbf{R}'_1, \mathbf{R}'_2; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'_1, \mathbf{L}'_2} \qquad \frac{}{\parr\, \cdot; \mathbf{L} \vdash \mathbf{R}; \bigotimes \cdot}$$

$$\frac{}{\parr\, \cdot; X^+, \mathbf{L} \vdash \mathbf{R}; \bigotimes X^+} \qquad \frac{A^+; \mathbf{L} \vdash \mathbf{R}}{\parr\, A^+; \mathbf{L} \vdash \mathbf{R}; \bigotimes \cdot} \qquad \frac{\mathbf{L} \vdash \mathbf{R}; A^-}{\parr\, \cdot; \mathbf{L} \vdash \mathbf{R}; \bigotimes A^-} \qquad \frac{}{\parr\, X^-; \mathbf{L} \vdash \mathbf{R}, X^-; \bigotimes \cdot}$$

$$\frac{A^+ \in \mathbf{R} \quad \mathbf{L} \vdash \mathbf{R}; [A^+]}{\mathbf{L} \vdash \mathbf{R}} \qquad \frac{A^- \in \mathbf{L} \quad [A^-]; \mathbf{L} \vdash \mathbf{R}}{\mathbf{L} \vdash \mathbf{R}}$$

Figure 3.6: A pattern-based formulation of PPL focusing proofs

3. In the focusing rules for 1 and $\bot$, and the atomic initial sequents, the rest of the context need not be empty.

Basically, we have made exactly those changes necessary to ensure the admissibility of weakening and contraction on formulas *inside the stable context.* Writing $\vdash_{[c]}$ for provability in this calculus, and letting a and c range over "$a$ssumptions" (A^- or X^+) and "$c$onclusions" (A^+ or X^-):

Weakening (Left)**.**

- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $a, \mathbf{L} \vdash_{[c]} \mathbf{R}$
- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}; [A]$ *then* $a, \mathbf{L} \vdash_{[c]} \mathbf{R}; [A]$
- *If* $[A]; \mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $[A]; a, \mathbf{L} \vdash_{[c]} \mathbf{R}$
- *If* $L; \mathbf{L} \vdash_{[c]} \mathbf{R}; R$ *then* $L; a, \mathbf{L} \vdash_{[c]} \mathbf{R}; R$

Weakening (Right)**.**

- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $\mathbf{L} \vdash_{[c]} \mathbf{R}, c$
- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}; [A]$ *then* $\mathbf{L} \vdash_{[c]} \mathbf{R}, c; [A]$
- *If* $[A]; \mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $[A]; \mathbf{L} \vdash_{[c]} \mathbf{R}, c$
- *If* $L; \mathbf{L} \vdash_{[c]} \mathbf{R}; R$ *then* $L; \mathbf{L} \vdash_{[c]} \mathbf{R}, c; R$

Contraction (Left)**.**

- *If* $a, a, \mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $a, \mathbf{L} \vdash_{[c]} \mathbf{R}$
- *If* $a, a, \mathbf{L} \vdash_{[c]} \mathbf{R}; [A]$ *then* $a, \mathbf{L} \vdash_{[c]} \mathbf{R}; [A]$
- *If* $[A]; a, a, \mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $[A]; a, \mathbf{L} \vdash_{[c]} \mathbf{R}$
- *If* $L; a, a, \mathbf{L} \vdash_{[c]} \mathbf{R}; R$ *then* $L; a, \mathbf{L} \vdash_{[c]} \mathbf{R}; R$

Contraction (Right)**.**

- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}, c, c$ *then* $\mathbf{L} \vdash_{[c]} \mathbf{R}, c$
- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}, c, c; [A]$ *then* $\mathbf{L} \vdash_{[c]} \mathbf{R}, c; [A]$
- *If* $[A]; \mathbf{L} \vdash_{[c]} \mathbf{R}, c, c$ *then* $[A]; \mathbf{L} \vdash_{[c]} \mathbf{R}, c$
- *If* $L; \mathbf{L} \vdash_{[c]} \mathbf{R}, c, c; R$ *then* $L; \mathbf{L} \vdash_{[c]} \mathbf{R}, c; R$

Proof. Immediate by induction on derivations. □

Since we included the structural properties explicitly in our axiomatization above, and we treat them implicitly in the focusing calculus, soundness is not quite as immediate as it was in §3.2.2, but it is nonetheless easy to see.

Theorem 3.4.4 (Soundness of Σ_3-style focusing for PPL)**.**

- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $\mathbf{L} \vdash_c \mathbf{R}$
- *If* $\mathbf{L} \vdash_{[c]} \mathbf{R}; [A]$ *then* $\mathbf{L} \vdash_c \mathbf{R}, A$
- *If* $[A]; \mathbf{L} \vdash_{[c]} \mathbf{R}$ *then* $A, \mathbf{L} \vdash_c \mathbf{R}$
- *If* $L; \mathbf{L} \vdash_{[c]} \mathbf{R}; R$ *then* $L, \mathbf{L} \vdash_c \mathbf{R}, R$

Proof. If we erase structural punctuation, almost every rule becomes an instance of an ordinary sequent calculus rule or trivial. The focusing rules for $\otimes$ and $⅋$ are justified from $\otimes R$ and $⅋L$ by repeated use of CL and CR, and the focusing rules for 1 and $\bot$ from $1R$ and $\bot L$ by repeated use of WL and WR. Likewise, the atomic initial sequents are justified by repeated use of WL and WR. Finally, the two rules in the group "**Neutral** $\to$ **Focus**" become instances of CR and CL, respectively. □

Similarly, the pattern-based formulation of focusing for PPL is almost identical to the one for MALL, except that the rules for proving neutral sequents $\mathbf{L} \vdash \mathbf{R}$ and multiplicative sequents $\parr\, \mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ are modified to build in weakening and contraction. As before, writing $\vdash_{[c^*]}$ for provability in the pattern-based formulation, we have:

Proposition 3.4.5 (Equivalence of focusing calculi)**.**

1. $\mathbf{L} \vdash_{[c^*]} \mathbf{R}; [A^+]$ *iff* $\mathbf{L} \vdash_{[c]} \mathbf{R}; [A^+]$
2. $A^+; \mathbf{L} \vdash_{[c^*]} \mathbf{R}$ *iff* $A^+; \mathbf{L} \vdash_{[c]} \mathbf{R}$
3. $\mathbf{L} \vdash_{[c^*]} \mathbf{R}; A^-$ *iff* $\mathbf{L} \vdash_{[c]} \mathbf{R}; A^+$
4. $[A^-]; \mathbf{L} \vdash_{[c^*]} \mathbf{R}$ *iff* $[A^+]; \mathbf{L} \vdash_{[c]} \mathbf{R}$
5. $\mathbf{L} \vdash_{[c^*]} \mathbf{R}$ *iff* $\mathbf{L} \vdash_{[c]} \mathbf{R}$

Corollary 3.4.6 (Soundness of pattern-based formulation)**.**

1. *If* $\mathbf{L} \vdash_{[c^*]} \mathbf{R}; [A^+]$ *then* $\mathbf{L} \vdash_c \mathbf{R}, A^+$
2. *If* $[A^-]; \mathbf{L} \vdash_{[c^*]} \mathbf{R}$ *then* $A^-, \mathbf{L} \vdash_c \mathbf{R}$
3. *If* $\mathbf{L} \vdash_{[c^*]} \mathbf{R}; a$ *then* $\mathbf{L} \vdash_c \mathbf{R}, a$
4. *If* $c; \mathbf{L} \vdash_{[c^*]} \mathbf{R}$ *then* $c, \mathbf{L} \vdash_c \mathbf{R}$
5. *If* $\mathbf{L} \vdash_{[c^*]} \mathbf{R}$ *then* $\mathbf{L} \vdash_c \mathbf{R}$
6. *If* $\parr\, \mathbf{R}'; \mathbf{L} \vdash_{[c^*]} \mathbf{R}; \bigotimes \mathbf{L}'$ *then* $(\parr\, \mathbf{R}'), \mathbf{L} \vdash_c \mathbf{R}, (\bigotimes \mathbf{L}')$

3.4.3 Classical focusing proofs are unrestricted canonical derivations

This punchline should by now be unsurprising: the pattern-based formulation of focusing for PPL is identical to our presentation of unrestricted canonical derivations with only simple hypotheses, under the same translation as in §3.2.4. As a corollary, we derive initial sequents and cut principles for focusing proofs by translating the corresponding identity and composition principles for unrestricted canonical derivations.

Theorem 3.4.7 (Initial sequents)**.** *The following initial sequents are derivable in* $\vdash_{[c^*]}$*:*

1. *If* $A^+ \in \mathbf{R}$ *then* $A^+; \mathbf{L} \vdash \mathbf{R}$
2. *If* $A^- \in \mathbf{L}$ *then* $\mathbf{L} \vdash \mathbf{R}; A^-$
3. *If* $X^+ \in \mathbf{L}$ *then* $X^+ \vdash \cdot; [X^+]$
4. *If* $X^- \in \mathbf{R}$ *then* $[X^-]; \cdot \vdash X^-$

Theorem 3.4.8 (Cut-admissibility)**.** *The following cuts are admissible in* $\vdash_{[c^*]}$*:*

1. *If* $\mathbf{L} \vdash \mathbf{R}; [A^+]$ *and* $A^+; \mathbf{L} \vdash \mathbf{R}$ *then* $\mathbf{L} \vdash \mathbf{R}$
2. *If* $\mathbf{L} \vdash \mathbf{R}; A^-$ *and* $[A^-]; \mathbf{L} \vdash \mathbf{R}$ *then* $\mathbf{L} \vdash \mathbf{R}$

3. *If* $\parr\,\mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ *and* $\mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'; [A^+]$ *then* $\mathbf{L} \vdash \mathbf{R}; [A^+]$

4. *If* $\parr\,\mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ *and* $[A^-]; \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'$ *then* $[A^-]; \mathbf{L} \vdash \mathbf{R}$

5. *If* $\parr\,\mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ *and* $\mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'; A^-$ *then* $\mathbf{L} \vdash \mathbf{R}; A^-$

6. *If* $\parr\,\mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ *and* $A^+; \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'$ *then* $A^+; \mathbf{L} \vdash \mathbf{R}$

7. *If* $\parr\,\mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ *and* $\parr\,\mathbf{R}''; \mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'; \bigotimes \mathbf{L}''$ *then* $\parr\,\mathbf{R}''; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}''$

8. *If* $\parr\,\mathbf{R}'; \mathbf{L} \vdash \mathbf{R}; \bigotimes \mathbf{L}'$ *and* $\mathbf{L}', \mathbf{L} \vdash \mathbf{R}, \mathbf{R}'$ *then* $\mathbf{L} \vdash \mathbf{R}$

Proof. These are all instances of the identity and composition principles for unrestricted canonical derivations, under the translation guide. □

3.4.4 The completeness theorem

Again we prove the completeness of focusing by way of weak focalization, with the proof almost identical to the one in §3.3.

Lemma 3.4.9 (Weak focalization). *Let* $(\mathfrak{L} \vdash \mathfrak{R}) \longleftrightarrow \Gamma$. *Then* $\mathfrak{L} \vdash_c \mathfrak{R}$ *implies there is a unrestricted canonical derivation of* $\Gamma \vdash \#$.

Proof. The only difference with the proof of Lemma 3.3.1 is that sometimes we need to invoke the weakening and contraction principles. We illustrate with a single example, the $\otimes R$ case:

- **Case** $\otimes$: The proof ends in $\dfrac{\mathfrak{L}_1 \vdash \mathfrak{R}_1, A \quad \mathfrak{L}_2 \vdash \mathfrak{R}_2, B}{\mathfrak{L}_1, \mathfrak{L}_2 \vdash \mathfrak{R}_1, \mathfrak{R}_2, A \otimes B}$, where $(\mathfrak{L}_1 \vdash \mathfrak{R}_1) \longleftrightarrow \Gamma_1$ and $(\mathfrak{L}_2 \vdash \mathfrak{R}_2) \longleftrightarrow \Gamma_2$. By the induction hypothesis, there exist unrestricted canonical derivations (ucds) $C_1 :: (\Gamma_1, \bullet A \vdash \#)$ and $C_2 :: (\Gamma_2, \bullet B \vdash \#)$. Let $\Gamma = \Gamma_1, \Gamma_2, \bullet A \otimes B$. Note that for any $p_1 :: (\Delta_1 \Vdash A)$ and $p_2 :: (\Delta_2 \Vdash B)$, we can build the derivation $C_{(p_1,p_2)} :: (\Gamma, \Delta_1, \Delta_2 \vdash \#)$ as follows:

$$C_{(p_1,p_2)} = \dfrac{\bullet A \otimes B \in \Gamma \qquad \dfrac{\dfrac{\overset{p_1}{\Delta_1 \Vdash A} \quad \overset{p_2}{\Delta_2 \Vdash B}}{\Delta_1, \Delta_2 \Vdash A \otimes B} \quad \overset{Id}{\Gamma, \Delta_1, \Delta_2 \vdash \Delta_1, \Delta_2}}{\Gamma, \Delta_1, \Delta_2 \vdash A \otimes B}}{\Gamma, \Delta_1, \Delta_2 \vdash \#}$$

Then we derive $\Gamma \vdash \#$ with

$$\dfrac{\dfrac{\overset{C_1}{\Gamma_1, \bullet A \vdash \#}}{\Gamma, \bullet A \vdash \#}\,\dagger \qquad \dfrac{\overset{p_1}{\Delta_1 \Vdash A} \longrightarrow \dfrac{\dfrac{\overset{C_2}{\Gamma_2, \bullet B \vdash \#}}{\Gamma, \Delta_1, \bullet B \vdash \#}\,\dagger \quad \dfrac{\overset{p_2}{\Delta_2 \Vdash B} \longrightarrow \overset{C_{(p_1,p_2)}}{\Gamma, \Delta_1, \Delta_2 \vdash \#}}{\Gamma, \Delta_1 \vdash \bullet B}}{\Gamma, \Delta_1 \vdash \#}\,\dagger}{\Gamma \vdash \bullet A}}{\Gamma \vdash \#}\,\dagger$$

where (†) indicates uses of substitution (right into left), and (†) indicates weakening. Again, note that our choice to substitute C_1 and C_2 in this order is arbitrary.

$$\frac{A \in \Gamma}{\Gamma \vdash A}\,hyp \qquad \frac{\Gamma, A \vdash \#}{\Gamma \vdash \sim A}\sim I \qquad \frac{\Gamma \vdash \sim A \quad \Gamma \vdash A}{\Gamma \vdash \#}\sim E$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\wedge E \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\wedge E$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\vee I \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\vee I \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\vee E$$

$$\frac{}{\Gamma \vdash \mathrm{T}}\,\mathrm{T}I \qquad \frac{\Gamma \vdash \mathrm{F}}{\Gamma \vdash C}\,\mathrm{F}E$$

(Note: # is a distinguished atomic formula, different from F.)

Figure 3.7: Natural deduction for conjunction, disjunction, and minimal negation

□

Corollary 3.4.10. *Weak entailment for unrestricted canonical derivations coincides with classical entailment, i.e.,* $A^+ \leq^- B^+$ *(or* $A^- \leq^+ B^-$*) iff* $|A| \supset |B|$ *is a classical theorem.*

Proof. As in the proof of Corollary 3.3.3, then applying Proposition 3.4.3. □

Corollary 3.4.11. *Let* b *be a classical theorem:*

1. *For any positive polarization* A^+ *of* b*, there is an unrestricted canonical derivation of* $\bullet A^+ \vdash \#$
2. *For any negative polarization* A^- *of* b*, there is an unrestricted canonical derivation of* $\cdot \vdash A^-$

3.5 Relating focusing and double-negation translations

Take a look at the two parts of Corollary 3.4.11, remembering how we glossed the different hypothetical judgments in Chapter 2. The conclusion of (1) we read aloud as, "If A is refutable, then contradiction". The conclusion of (2) we read simply as "A is provable", but for a negative notion of proof-by-contradiction. It seems then that focusing proofs , via their one-to-one correspondence with canonical derivations, give us different double-negation interpretations of classical propositions—and the completeness of focusing corresponds to the completeness of these interpretations.

How does this relate to the traditional double-negation translations of classical into intuitionistic or minimal logic? To answer this question, we first relate canonical derivations to proofs in minimal logic. A standard natural deduction[5] for minimal logic is given in Figure 3.7. Consider the following pair of maps $(-)^{m+}$ and $(-)^{m-}$ from polarized to unpolarized formulas:

[5]Note we could also consider a sequent calculus presentation, which would make some aspects of the correspondence with canonical derivations more direct. However, we are anticipating Chapter 4, where we will relate (the Curry-Howard interpretation of) canonical derivations with terms of λ-calculus.

$$\begin{array}{rcl@{\qquad}rcl}
X^{m+} & = & X & X^{m-} & = & \sim X \\
1^{m+} & = & \mathrm{T} & \bot^{m-} & = & \mathrm{T} \\
0^{m+} & = & \mathrm{F} & \top^{m-} & = & \mathrm{F} \\
(A \oplus B)^{m+} & = & A^{m+} \vee B^{m+} & (A \& B)^{m-} & = & A^{m-} \vee B^{m-} \\
(A \otimes B)^{m+} & = & A^{m+} \wedge B^{m+} & (A \parr B)^{m-} & = & A^{m-} \wedge B^{m-} \\
(\overset{+}{\neg} A)^{m+} & = & \sim A^{m+} & (\overset{-}{\neg} A)^{m-} & = & \sim A^{m-} \\
(\downarrow A)^{m+} & = & \sim A^{m-} & (\uparrow A)^{m-} & = & \sim A^{m+} \\
(A - B)^{m+} & = & A^{m+} \wedge B^{m-} & (A \to B)^{m-} & = & A^{m+} \wedge B^{m-}
\end{array}$$

Note that $\sim A$ stands for minimal negation, i.e., negation defined by $\sim A = A \supset \#$, where $\#$ is a distinguished atomic formula (as opposed to the intuitionistic definition $\sim A = A \supset \mathrm{F}$). If we wanted we could be more explicit and write $\sim_{\#} A$, since the atom $\#$ is arbitrary.

We write A^m for the translation of an arbitrary polarity formula (A^{m+} when A is positive, A^{m-} when A is negative). Observe that for the purely positive fragment of PPL, A^m is equal to the forgetful translation $|A|$ we defined in Definition 3.4.1. For the negative fragment, A^m is the De Morgan dual of $|A|$. The translation is extended to assertions and refutations as follows:

$$(A^+)^m = A^{m+} \quad (\bullet A^-)^m = A^{m-} \quad (\bullet A^+)^m = \sim A^{m+} \quad (A^-)^m = \sim A^{m-}$$

The contradiction judgment $\#$ is translated as the distinguished atom $\#$. Frames Δ are translated on the right by treating them as big conjunctions:

$$(\cdot)^m = \mathrm{T} \quad (\Delta_1, \Delta_2)^m = \Delta_1^m \wedge \Delta_2^m$$

On the left, appearing in contexts, frames are translated into lists of assumptions:

$$(\cdot)^m = \cdot \quad (\Delta_1, \Delta_2)^m = (\Delta_1^m, \Delta_2^m)$$

so that polarized contexts can be translated into minimal contexts:

$$(\cdot)^m = \cdot \quad (\Gamma, \Delta)^m = (\Gamma^m, \Delta^m)$$

Theorem 3.5.1 (Soundness of translation into minimal logic)**.** *Any unrestricted canonical derivation of $\Gamma \vdash J$ can be transformed into a minimal logic proof of $\Gamma^m \vdash J^m$.*

Proof. We first establish the following facts:

1. Any A-pattern with frame Δ can be transformed into a minimal logic proof of $\Delta^m \vdash A^m$
2. Suppose that for every A-pattern, there is a minimal logic proof of $\Gamma^m, \Delta^m \vdash J^m$, where Δ is the frame of the pattern. Then there is a minimal logic proof of $\Gamma^m, A^m \vdash J^m$

These are both obvious by inspection of the pattern-formation rules. The theorem follows immediately, by induction on canonical derivations. □

Theorem 3.5.2 (Completeness of translation into minimal logic). *If $\Gamma^m \vdash J^m$ has a minimal logic proof, then there is an unrestricted canonical derivation of $\Gamma \vdash J$.*

Proof. By induction on the minimal natural deduction proof, similar to the proof of weak focalization. We give a few representative cases:

- **Case** $\sim I$: The proof ends in $\dfrac{\Gamma^m, A^m \vdash \#}{\Gamma^m \vdash \sim A^m}$. Note that $\sim A^m$ could be the translation of many different judgments, e.g., $\bullet A^+$ or $\overset{+}{\neg} A$, or A^- or $\bullet \overset{-}{\neg} A$, etc. Without loss of generality we can assume the first case (since it either implies or is dual to the other cases). By appealing to the i.h. we obtain a ucd of $\Gamma, A^+ \vdash \#$, which yields $\Gamma \vdash \bullet A$ by an admissible inference.
- **Case** $\vee E$: The proof ends in $\dfrac{\Gamma^m \vdash A^m \vee B^m \quad \Gamma^m, A^m \vdash J^m \quad \Gamma^m, B^m \vdash J^m}{\Gamma^m \vdash J^m}$. Since $(A \oplus B)^m = A^m \vee B^m$, by (2) there is a ucd of $\Gamma \vdash A \oplus B$, and hence some $\Delta \Vdash A \oplus B$ such that $\Gamma \vdash \Delta$. Since $\Delta \Vdash A \oplus B$ iff $\Delta \Vdash A$ or $\Delta \Vdash B$, we obtain the desired result by first applying pattern substitution on one of the other two premises, and then applying composition.

$\square$

Corollary 3.5.3. *For purely positive A, strong entailment coincides with minimal entailment, i.e., $A \leq^+ B$ iff $|A| \supset |B|$ is a minimal theorem.*

Corollary 3.5.4. *For purely negative A, strong entailment coincides with minimal entailment, i.e., $A \leq^+ B$ iff $|A| \supset |B|$ is a minimal theorem.*

Now, our general recipe for building sound and complete double-negation translations of classical logic into minimal logic can be represented by a diagram:

$$\begin{tikzcd}
 & \text{polarized logic} \arrow[dr, "(-)^m"] & \\
\text{classical logic} \arrow[ur, "(-)^*"] & & \text{minimal logic} \arrow[ll]
\end{tikzcd}$$

Starting from a classical theorem b, we translate it to a polarized formula b^* and apply the completeness of focusing to obtain an appropriate canonical derivation. Call this step $(-)^*$. In the next step $(-)^m$, we apply soundness of the embedding into minimal logic. This establishes that the translation $(-)^m \circ (-)^*$ is complete.[6] Conversely, it is sound simply because minimal logic is included in classical logic, and because $(-)^m \circ (-)^*$ carries b to a formula that is classically equivalent.

By defining different polarizations of the classical connectives, we can reconstruct different double-negation translations. For example, there are two particularly obvious ways to polarize a formula: into the purely positive fragment of PPL, or into the purely negative fragment. The former gives us Glivenko's famous theorem, while the latter gives us a simple but lesser-known double-negation translation due to Lafont, Reus, and Streicher [1993], recently used by Streicher and Kohlenbach [2007] and (independently) by Avigad [2006] to connect Gödel's Dialectica translation to the variant by Schoenfield.[7]

[6]It is important to distinguish the action of $(-)^*$ and $(-)^m$ on classical/PPL formulas from their action on classical proofs and canonical derivations. By $(-)^m \circ (-)^*$, we mean the latter. In general, the translation $(-)^m \circ (-)^*$ will carry b to a formula that may be different from b^{*m} by one or two negations.

[7]The paper by Lafont, Reus, and Streicher gives credit to Krivine [1990] and Girard [1991a] for the inspiration for the translation, although the connection is not completely obvious. The simple version of the translation given below (Theorem 3.5.6) for the full suite of propositional connectives only appears in the recent papers of Streicher and Kohlenbach [2007] and Avigad [2006].

Theorem 3.5.5 (Glivenko [1929])**.** *b is a classical theorem iff $\sim\sim b$ is a minimal theorem.*

Proof. Take $(-)^*$ to be the purely positive polarization:

$$X^* = X^+ \qquad \mathrm{T}^* = 1 \qquad \mathrm{F}^* = 0$$

$$(a \wedge b)^* = a^* \otimes b^* \qquad (a \vee b)^* = a^* \oplus b^* \qquad (\sim a)^* = \overset{+}{\neg}\, a^*$$

We can easily verify that b^* is a polarization of b, and that $b^{*m+} = b$. Assume b is a classical theorem. We apply Corollary 3.4.11(1) to obtain a ucd of $\bullet b^* \vdash \#$, and Theorem 3.5.1 to obtain a minimal logic proof of $\sim b^{*m+} \vdash \#$. Hence $\sim\sim b$ is a minimal logic theorem. In the backward direction, we use the inclusion of minimal logic into classical logic, and that $\sim\sim b$ and b are classically equivalent. $\square$

Theorem 3.5.6 (Streicher and Kohlenbach [2007], Avigad [2006])**.** *Let b' be defined as follows:*

$$X' = \sim X \qquad \mathrm{T}' = \mathrm{F} \qquad \mathrm{F}' = \mathrm{T}$$

$$(a \wedge b)' = a' \vee b' \qquad (a \vee b)' = a' \wedge b' \qquad (\sim a)' = \sim a'$$

Then b is a classical theorem iff $\sim b'$ is a minimal theorem.

Proof. Take $(-)^*$ to be the purely negative polarization:

$$X^* = X^- \qquad \mathrm{T}^* = \top \qquad \mathrm{F}^* = \bot$$

$$(a \wedge b)^* = a^* \& b^* \qquad (a \vee b)^* = a^* \parr b^* \qquad (\sim a)^* = \overset{-}{\neg}\, a^*$$

We can easily verify that b^* is a polarization of b, and that $b^{*m-} = b'$. Assume b is a classical theorem. We apply Corollary 3.4.11(2) to obtain a ucd of $\cdot \vdash b^*$, and Theorem 3.5.1 to obtain a minimal logic proof of $\cdot \vdash \sim b^{*m-}$. Hence $\sim b'$ is a minimal logic theorem. In the backward direction, we use the inclusion of minimal logic into classical logic, and that $\sim b'$ and b are classically equivalent. $\square$

The well-known Gödel-Gentzen translation corresponds to a slightly more involved polarization.

Theorem 3.5.7 (Gödel [1932], Gentzen [1936])**.** *Let b^G be defined as follows:*

$$X^G = \sim\sim X \qquad \mathrm{T}^G = \mathrm{T} \qquad \mathrm{F}^G = \sim \mathrm{T}$$

$$(A \wedge B)^G = A^G \wedge B^G \qquad (A \vee B)^G = \sim(\sim A^G \wedge \sim B^G) \qquad (\sim A)^G = \sim A^G$$

Then b is a classical theorem iff b^G is a minimal theorem.

Proof. Consider the following polarization:

$$X^* = {\downarrow}{\uparrow} X^+ \qquad \mathrm{T}^* = 1 \qquad \mathrm{F}^* = {\downarrow}\bot$$

$$(a \wedge b)^* = a^* \otimes b^* \qquad (a \vee b)^* = {\downarrow}({\uparrow} a^* \parr {\uparrow} b^*) \qquad (\sim a)^* = \overset{+}{\neg}\, a^*$$

We can easily verify $b^{*m} = b^G$. And again, we can immediately see that b^* is indeed a polarization of b, and that if b is a classical theorem then there is a ucd of $\bullet b^* \vdash \#$. But then the embedding into minimal logic only lets us conclude $\sim\sim b^G$, rather than b^G. If we could strengthen $\bullet b^* \vdash \#$ to $\cdot \vdash b^*$, we would obtain our result. But what justifies that step?

Definition 3.5.8. *We say that A is* **tagless** *if there is exactly one A-pattern, and if that pattern's frame only has hypotheses of the form $\bullet B^+$ or C^-.*

For example, $\downarrow A \otimes \downarrow B$ is tagless, but $\downarrow A \oplus \downarrow B$ and $X^+ \otimes Y^+$ are not.

Lemma 3.5.9. *Let A be tagless. If there is a ucd of $\Gamma, \bullet A^+ \vdash \#$ (resp. $\Gamma, A^- \vdash \#$) then there is a ucd of $\Gamma \vdash A^+$ (resp. $\Gamma \vdash \bullet A^-$).*

Proof. Assume A is positive (the negative case is dual). Let Δ be the frame of the unique A proof-pattern. To derive $\Gamma \vdash A^+$, we must show $\Gamma \vdash \Delta$, which reduces to showing $\Gamma \vdash \bullet B^+$ and $\Gamma \vdash C^-$ for every hypothesis $\bullet B^+ \in \Delta$ and $C^- \in \Delta$ (there are no other forms of hypothesis in Δ). Consider the case of a positive hypothesis $\bullet B^+ \in \Delta$ (the negative case is dual). To derive $\Gamma \vdash \bullet B^+$, we must show that $\Gamma, \Delta' \vdash \#$ for every $\Delta' \Vdash B^+$. Now, by assumption we have that $\Gamma, \bullet A^+ \vdash \#$, which we can weaken to $\Gamma, \Delta', \bullet A^+ \vdash \#$. So, it suffices to show that $\Gamma, \Delta' \vdash \bullet A^+$ and apply composition. But since A is tagless, this reduces to showing $\Gamma, \Delta', \Delta \vdash \#$ for Δ as above. We complete the derivation like so:

$$\dfrac{\bullet B^+ \in \Delta \qquad \dfrac{\overset{p'}{\Delta' \Vdash B^+} \qquad \overset{Id}{\Gamma, \Delta', \Delta \vdash \Delta'}}{\Gamma, \Delta', \Delta \vdash B^+}}{\Gamma, \Delta', \Delta \vdash \#}$$

□

Therefore we can finish the proof of completeness of the Gödel-Gentzen translation by observing that the polarization b^* defined above is always tagless. As usual, soundness of the translation is obvious because b^G is classically equivalent to b. □

3.6 Related Work

Most of the results of this section about focusing proofs have appeared in some form or another in prior work, at least in spirit—only the presentation in terms of canonical derivations is new, and the connection to minimal logic perhaps made clearer. The original motivation for introducing polarities [Girard, 1991a, 1993] was to better understand the classical double-negation translations, and Danos, Joinet, and Schellinx [1997] explored this view in depth, defining a polarized sequent calculus $\mathbf{LK}^{tq}$, and studying the behavior of alternate polarizations of classical proofs. Laurent [2002] gave a similar analysis in his dissertation. Our approach was only a slight twist, since we combine polarization with focusing, and so can view the completeness theorem for focusing as implying the completeness of different double-negation translations, by way of the embedding into minimal logic.

Since Andreoli's original work, focusing has been used very successfully as a proof search procedure in automated theorem provers, both for linear and intuitionistic logic [Howe, 1998, Chaudhuri, 2006, McLaughlin and Pfenning, 2008].

Chapter 4

Proofs as programs

Purely applicative languages are often said to be based on a logical system called the lambda calculus, or even to be "syntactically sugared" versions of the lambda calculus... However, as we will see, although an unsugared applicative language is syntactically equivalent to the lambda calculus, there is a subtle semantic difference. Essentially, the "real" lambda calculus implies a different "order of application" (i.e., normal-order evaluation) than most applicative programming languages.

—John Reynolds [1972]

In this chapter I will go back to our motivating analogy, and explain how to read derivations in polarized logic as a programming language. As it turns out, the different forms of logical inference correspond directly to syntactic categories already familiar from the theory of functional programming languages. For example, a proof pattern corresponds to what we ordinarily think of as "pattern" in functional programming, i.e., a tree of constructors, with variables at the leaves. A proof of a positive proposition corresponds to a *value* under eager semantics, decomposed as a pattern and a substitution for the pattern's variables. A refutation of a positive proposition corresponds to a call-by-value *continuation,* defined by a map from patterns to expressions, i.e., by "pattern-matching". The negative story is dual and somewhat less intuitive, but corresponds closely with (and gives a more refined analysis of) call-by-name and "lazy" evaluation.

These concepts are familiar from the *semantics* of programming languages, but by applying the Curry-Howard correspondence to derivations of polarized logic we reconstruct them as *syntax.* Perhaps the overarching lesson to draw from this is that syntax and semantics should not always be treated independently, because there are interactions going both ways. A historical example of this interplay is the *continuation-passing-style* transform, a syntactic transformation which, as Reynolds [1972] and Plotkin [1975] observed, determines an evaluation strategy for a program. One way to understand the results of this section is that the language we extract from polarized logic intrinsically enforces continuation-passing-style, and just as we saw that different polarizations correspond to different double-negation translations of classical logic, we can see the polarity of a type as corresponding to a choice of different (call-by-value or call-by-name) CPS translations of the λ-calculus. From the composition principles for canonical derivations (or equivalently, from the cut-admissibility theorem for focusing proofs), we extract a procedure for evaluating programs that is entirely deterministic—the answer to the question of eager vs. lazy evaluation is *encoded in types,* rather than being a global property of the language.

The first part of this chapter is just an exercise in transliteration, showing how to annotate logical derivations with a syntax of terms, and then expressing the identity and composition principles on this syntax. Although we are merely reformulating the definitions and results of Chapter 2, our aim here is to build up a programming language and some operational intuitions behind it. We also define a simple notion of syntactic or *definitional* equality on these terms, and show that the terms corresponding to the identity and composition principles deserve the name (i.e., they satisfy suitably formulated unit and associativity properties).

Once we have this core language, we can then go on to consider richer notions of computation. In particular, we will consider extensions of the language with different kinds of *effects.* To begin we include only two very simple effects, Ω (the diverging computation) and $\mho$ (the aborting computation), which have analogues in Girard's ludics [2001]. Since closed programs can now yield two different observable results, it makes sense to consider a notion of *observational* equivalence. We state this in terms of an environment semantics for evaluating closed programs, which logically corresponds to a procedure for eliminating multiple cuts embedded within a proof, rather than just a single cut. We consider the relationship between definitional equality and observational equivalence, proving the perhaps surprising theorem that in the presence of the two effects Ω and $\mho$ *plus a counter,* any two syntactically distinct terms in the core language can be observationally distinguished. This generalizes a result by Girard, who showed that to distinguish two *affine* terms (more literally: ludics' "designs"), the effects $\mho$ and Ω are enough—and it gives us some confidence that we really do have a canonical notion of syntax.

We also briefly discuss an untyped version of the language, and its relationship to the typed version. As with the usual λ-calculus, we can think of the untyped language as really *uni*-typed (or actually bi-typed, if we include both positive and negative polarities). In our case, though, the correspondence is more direct than usual, because whereas the usual translation from untyped to typed syntax involves addition of coercions, here the coercions were already present, mediating between the different syntactic categories.

Finally, we conclude the chapter by making precise the correspondence between polarization and continuation-passing style, showing how both call-by-value and call-by-name λ-calculus are embedded within our language $\mathcal{L}$, as a particular mode of programming.

4.1 Type-free notations for typeful derivations

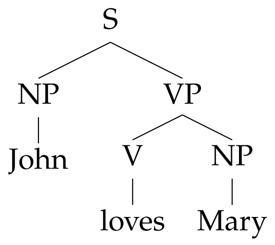
Before beginning we should say just a few more words about what we are hoping to accomplish. At least before we extend the language with effects, there is a technical sense in which we won't do anything we haven't already done in Chapter 2, because we view programs as literally the same thing as derivations in polarized logic. Reynolds [2000] calls this equation of logical derivations with programs an *intrinsic* definition of a typed language, because it only assigns meaning to well-typed programs. This is also sometimes called a "Church" view of typing, since it matches Alonzo Church's formulation of the simply-typed λ-calculus [1940]. On the other hand, an *extrinsic* definition begins with a raw, untyped syntax, and then defines types as properties of these untyped terms, preserved by operations such as reduction. This is sometimes called a "Curry" view of typing, after the work of Haskell Curry [Curry and Feys, 1958].

In programming languages theory, there is often a bias towards the extrinsic interpretation, because it accords with the intuition that programs carry a computational content irrespective of their type. In fact, a crucial property of strongly-typed functional programming languages like ML and Haskell is that they can be given a *type-free* operational semantics, i.e., programs can be evaluated without any run-time type information. But this is not inconsistent with an intrinsic view of typing. While the intrinsic approach equates terms with derivations involving types, it is not the case that terms have to *mention* these types. So our first goal in this chapter is to develop different type-free notations for typeful derivations.

Consider an analogy from linguistics. Suppose we have a context-free grammar for a (fairly small) subset of English:

$1 : \text{S} \to \text{NP VP}$
$2 : \text{VP} \to \text{V NP}$
$3 : \text{NP} \to \text{John}$
$4 : \text{NP} \to \text{Mary}$
$5 : \text{V} \to \text{loves}$

To show that "John loves Mary" is a well-formed sentence, we can exhibit the following parse tree:



The parse tree doesn't say which grammar rules are being applied at each node, but we can indicate them with a more explicit tree (written upside-down for convenience, in inference-rule notation, and where $(\cdot)$ represents concatenation):

$$\dfrac{\dfrac{\dfrac{\text{John}}{\text{NP}}\,(3) \qquad \dfrac{\dfrac{\dfrac{\text{loves}}{\text{V}}\,(5) \quad \dfrac{\text{Mary}}{\text{NP}}\,(4)}{\text{V NP}}\,(\cdot)}{\text{VP}}\,(2)}{\text{NP VP}}\,(\cdot)}{\text{S}}\,(1)$$

But now the terminals and non-terminals annotating the nodes of the tree are redundant, because they can be reconstructed from the rules. We may as well write the derivation like so:

$$\dfrac{3 \quad \dfrac{5 \quad 4}{2}}{1}$$

or linearly as the expression $1(3, 2(5, 4))$. This representation is type-free, even though we can mechanically reconstruct all of the types from the definition of the grammar.

Although our canonical derivations have more complex structure than parse trees, we will follow more or less the same procedure to come up with a type-free notation. In the next sections we will explain how to build a type-free notation for arbitrary derivations, while still taking this as an *intrinsic* definition of a programming language. We will then show how to express the composition principles for canonical derivations directly on this type-free notation, validating the intuition that the run-time execution of programs does not require type information, although we can still use types to *reason* about this execution. (This will not be the end of our take on the Curry vs. Church debate, however. In Chapter 6, we will explain how to define extrinsic *refinement types* that further classify intrinsically well-typed programs.)

4.2 $\mathcal{L}^+$: A call-by-value language

Just as we started Chapter 2 by describing a proof-biased approach to logic—and later explained how to interpret it as a *fragment* of a larger, polarized logic—we will start this chapter by describing a "value-biased", or *call-by-value* language $\mathcal{L}^+$, and then show how to view it as a fragment of a larger language $\mathcal{L}$ that freely mixes call-by-value and call-by-name evaluation.

4.2.1 Continuation frames and value patterns

We begin by recalling some of the definitions of §2.1.1, and recasting them in Curry-Howard terms. Rather than speaking of positive propositions, we will now speak of positive *types* A^+, B^+, C^+. As in previous chapters, we only write the polarity marker $(-)^+$ for emphasis, and sometimes omit it to relax the notation. Recall that in this fragment, frames Δ consist of lists of refutation holes $\bullet A_1^+, \dots, \bullet A_n^+$. We now call these **continuation holes**.

Likewise, whereas we previously referred to derivations of $\Delta \Vdash A^+$ as *proof patterns*, we now call them **value patterns**. Recall the rules for deriving $\Delta \Vdash A^+$:

$$\frac{}{\bullet A^+ \Vdash \overset{\pm}{\neg} A}$$

$$\frac{}{\cdot \Vdash 1} \qquad \frac{\Delta_1 \Vdash A \quad \Delta_2 \Vdash B}{\Delta_1, \Delta_2 \Vdash A \otimes B}$$

$$\frac{\Delta \Vdash A}{\Delta \Vdash A \oplus B} \qquad \frac{\Delta \Vdash B}{\Delta \Vdash A \oplus B} \qquad \text{(no rule for 0)}$$

Now we assign labels to the rules:

$$\frac{}{\bullet A^+ \Vdash \overset{\pm}{\neg} A}\ \_$$

$$\frac{}{\cdot \Vdash 1}\ () \qquad \frac{\Delta_1 \Vdash A \quad \Delta_2 \Vdash B}{\Delta_1, \Delta_2 \Vdash A \otimes B}\ \mathsf{pair}$$

$$\frac{\Delta \Vdash A}{\Delta \Vdash A \oplus B}\ \mathsf{inl} \qquad \frac{\Delta \Vdash B}{\Delta \Vdash A \oplus B}\ \mathsf{inr}$$

These labels give us a type-free notation for patterns. For example, $\mathsf{pair}(\_, \mathsf{inl}\,\_)$ and $\mathsf{pair}(\_, \mathsf{inr}\,\_)$ are two $\neg A \otimes (\neg B \oplus \neg C)$-patterns with frames $\bullet A, \bullet B$ and $\bullet A, \bullet C$, respectively. Since pair-patterns are used pretty frequently, we write (p_1, p_2) as shorthand for $\mathsf{pair}(p_1, p_2)$. We also implicitly associate unary pattern constructors to the right, so for example we can write $\mathsf{inl}\,\mathsf{inl}\,\_$ as shorthand for $\mathsf{inl}\,(\mathsf{inl}\,\_)$.

As in Chapter 2, we will use patterns in a fairly open-ended way, without relying on the properties of particular connectives except to construct particular examples. Here we can introduce some additional types that didn't really have interesting logical counterparts in Chapter 2. For example, we define patterns for booleans and natural numbers:

$$\frac{}{\cdot \Vdash \mathbf{B}}\ \mathsf{tt} \qquad \frac{}{\cdot \Vdash \mathbf{B}}\ \mathsf{ff} \qquad\qquad \frac{}{\cdot \Vdash \mathbf{N}}\ \mathsf{z} \qquad \frac{\Delta \Vdash \mathbf{N}}{\Delta \Vdash \mathbf{N}}\ \mathsf{s}$$

as well as patterns for a paradoxical type $\mathbf{D}$:

$$\frac{\Delta \Vdash \mathbf{N}}{\Delta \Vdash \mathbf{D}}\ \mathsf{dn} \qquad \frac{\Delta \Vdash {}^{\pm}\mathbf{D}}{\Delta \Vdash \mathbf{D}}\ \mathsf{dk}$$

The type $\mathbf{D}$ is meant to include both natural numbers and $\mathbf{D}$-continuations. Note that this *is* a valid inductive definition of $\mathbf{D}$-patterns, but that $\mathbf{D}$ itself is not well-founded in the sense of the definition ordering (§2.1.2), since $\mathbf{D} \prec \mathbf{D}$. Later, we will use this type to construct non-terminating programs.

Finally, let's observe that value patterns really do correspond to patterns in the usual functional programming sense, with the proviso that the latter go "as deep as possible", i.e., up to continuations. When we want to emphasize this proviso, we say that value patterns are **maximal.** One consequence of maximality, e.g., is that $\mathbf{N}$-patterns are in one-to-one correspondence with numerals.

Notation. *We write $\overline{n}$ as syntactic sugar for the $\mathbf{N}$-pattern corresponding to the n^{th} numeral, e.g., $\overline{0} = \mathsf{z}$, $\overline{1} = \mathsf{s}\,\mathsf{z}$, $\overline{2} = \mathsf{s}\,\mathsf{s}\,\mathsf{z}$, etc. Note that every $\mathbf{N}$-pattern has an empty frame, i.e., has no holes for continuations.*

4.2.2 Annotated frames, contexts and binding

Of course the reader may object, "But patterns in functional programming languages bind variables!" Well patterns in $\mathcal{L}^+$ do as well, but we have to think a bit carefully about what variables are. Recall, we gave simple inductive rules for deciding containment $\Delta_1 \in \Delta_2$:

$$\frac{}{\Delta \in \Delta} \qquad \frac{\Delta \in \Delta_1}{\Delta \in \Delta_1, \Delta_2} \quad \frac{\Delta \in \Delta_2}{\Delta \in \Delta_1, \Delta_2}$$

Now, suppose we assign labels to these rules:

$$\frac{}{\Delta \in \Delta}\ \mathsf{here} \qquad \frac{\Delta \in \Delta_1}{\Delta \in \Delta_1, \Delta_2}\ \mathsf{left} \quad \frac{\Delta \in \Delta_2}{\Delta \in \Delta_1, \Delta_2}\ \mathsf{right}$$

Then a *derivation* of a containment relationship $\Delta_1 \in \Delta_2$ is a sequence of these constructors defining a particular path through Δ_2, and can be seen as a "de Bruijn index" for a variable [de Bruijn, 1972]. The way we associate Δ_2 matters for how we build this index. For example, the derivation of $\bullet B \in (\bullet A, (\bullet B, \bullet C))$—the index of a continuation *variable*—would be annotated $\mathsf{right}\ \mathsf{left}\ \mathsf{here}$.

However, programming in this style becomes tedious very quickly. We would rather have actual names to refer to the continuation holes inside of frames, and not care about the associativity of frames. So, let us introduce some new notation.

Definition 4.2.1 (Annotated frames)**.** *An* **annotated frame** *is a frame with labelled leaves, subject to the usual variables conventions: we assume all of the labels within an annotated frame are disjoint, and we can freely α-convert labels. In particular, we write an annotated frame of* **continuation variables** *as $\kappa_1 : \bullet A_1, \ldots, \kappa_n : \bullet A_n$, where the κ_i are distinct.*

Now, we can give an alternative notation for patterns, using annotated frames. In particular, we annotate the negation pattern-rule with a continuation variable:

$$\frac{}{\kappa : \bullet A^+ \Vdash \overset{+}{\neg} A}\ \kappa$$

The other rules remain the same, but we can reinterpret them as operating on annotated frames. For example, consider the pair pattern-rule:

$$\frac{\Delta_1 \Vdash A \quad \Delta_2 \Vdash B}{\Delta_1, \Delta_2 \Vdash A \otimes B}\ \mathsf{pair}$$

Because of the requirement that the labels in an annotated frame are distinct, the rule now has an implicit side-condition, that the two subpatterns do not repeat variable names. (Note that this corresponds to the usual linearity restriction on pattern-matching in most functional programming languages.)

To give an example, the two $\neg A \otimes (\neg B \oplus \neg C)$-patterns we wrote above could be annotated with explicit variable names as $\mathsf{pair}(\kappa_1, \mathsf{inl}\,\kappa_2)$ and $\mathsf{pair}(\kappa_1, \mathsf{inr}\,\kappa_2)$. Their annotated frames are $\kappa_1 : \bullet A, \kappa_2 : \bullet B$ and $\kappa_1 : \bullet A, \kappa_2 : \bullet C$.

When defining the terms of $\mathcal{L}^+$ below, we will work with both annotated and unannotated frames. However, a *context* is always a list of annotated frames. That is, a term inside a context can mention variables, which are *bound by* the context. (This is one reason why it is useful to conceptually distinguish frames and contexts.)

4.2.3 Values, continuations, substitutions, expressions

With a definition of patterns (and frames) in hand, we can now "make a language for ourselves". Recall the four hypothetical judgments:

$$\Gamma \vdash A \qquad \Gamma \vdash \bullet A$$

$$\Gamma \vdash \Delta \qquad \Gamma \vdash \#$$

and the rules for building canonical derivations of these judgments within the positive, atom-free, simple fragment of PPL:

$$\frac{\Delta \Vdash A^+ \quad \Gamma \vdash \Delta}{\Gamma \vdash A^+} \qquad \frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A^+}$$

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash (\Delta_1, \Delta_2)} \qquad \frac{\bullet A^+ \in \Gamma \quad \Gamma \vdash A^+}{\Gamma \vdash \#}$$

We now interpret these derivations as terms of a programming language.

Definition 4.2.2 (Terms)**.** *A derivation $t :: (\Gamma \vdash J)$ is called a* **term.** *Terms may be further classified as follows:*

- *A* **value** *is a derivation $V :: (\Gamma \vdash A)$*
- *A* **continuation** *is a derivation $K :: (\Gamma \vdash \bullet A)$*
- *A* **substitution** *is a derivation $\sigma :: (\Gamma \vdash \Delta)$*
- *An* **expression** *is a derivation $E :: (\Gamma \vdash \#)$*

Sometimes we say "A-value", "A-continuation", etc., for added specificity. Again, the interesting part is the type-free notation for terms, and the operational intuition that goes along with it. We examine the rules for each judgment in turn:

- A^+: This judgment has a single introduction rule:

$$\frac{\overset{p}{\Delta \Vdash A^+} \quad \overset{\sigma}{\Gamma \vdash \Delta}}{\Gamma \vdash A^+}$$

  The rule combines a value pattern p and a substitution σ for the frame of the pattern, and we write the result as $p[\sigma]$. If we think in terms of annotated frames, the intuition behind this rule is expressed by the slogan,

  a value is a pattern under a substitution

  This holds intuitively in call-by-value languages like ML, by a fairly trivial "factorization lemma". For example, the ML value

```
(fn x => x*x, fn x => x-3)
```

  can be factored as the pattern

```
(f,g)
```

  composed with the substitution

```
let val f = fn x => x*x
    val g = fn x => x-3
in
```

  On the other hand, here we really only care about the structure of the pattern, and not the variable names, so we should read the rule as taking an unannotated frame. Either way, the utility of this factorization is that values are given a uniform representation, which we will exploit to great effect when defining the operational semantics of $\mathcal{L}^+$.

- $\bullet A^+$: This judgment has a single introduction rule:

$$\frac{\overset{p}{\Delta \Vdash A^+} \quad \longrightarrow \quad \overset{E_p}{\Gamma, \Delta \vdash \#}}{\Gamma \vdash \bullet A^+}$$

  Encoded in the rule is the slogan that

  a continuation is a map from patterns to expressions

  This slogan has some deep syntactic and semantic ramifications, so let's explore them carefully. First, the rule gives us license to define continuations by *pattern-matching*, just as we do in languages like ML. For example, we build a $\mathbf{N}$-continuation K by listing the cases

$$\begin{array}{rcl} K\ \mathsf{z} & = & E_0 \\ K\ \mathsf{s}\,\mathsf{z} & = & E_1 \\ K\ \mathsf{s}\,\mathsf{s}\,\mathsf{z} & = & E_2 \\ & \vdots & \end{array}$$

which we might also express more concisely as

$$K\ \overline{n} \quad = \quad E_n$$

A $\mathbf{D}$-continuation might be defined by the map

$$\begin{array}{rcl} K\ \mathsf{dn}\,\overline{n} & = & E_n \\ K\ \mathsf{dk}\,\kappa & = & E_\kappa \end{array}$$

where E_κ can use the bound continuation variable κ, relying on the annotated view of frames.

The fact that the rule quantifies over all A-patterns builds in the typical side-condition (usually checked but not always enforced) that pattern-matching is *exhaustive.* It is worth noting that although the practical benefits of pattern-matching notation are frequently recognized, it is often seen as a matter of surface syntax, as "syntactic sugar" for more verbose elimination constructs that explain what is really going on at a deeper level. We are taking the opposite view by treating pattern-matching as a primitive notion in syntax.

Second, the fact that continuations are defined by *maximal* pattern-matching has the consequence that they are *strict.* For example, to define a $\mathbf{D}$-continuation, we define its behavior on any value matching one of the patterns $\mathsf{dn}\,\mathsf{z}$, $\mathsf{dn}\,\mathsf{s}\,\mathsf{z}$, $\mathsf{dn}\,\mathsf{s}\,\mathsf{s}\,\mathsf{z}$, etc., or $\mathsf{dk}\,\kappa$. In particular, we leave *undefined* its behavior on a diverging computation of a $\mathbf{D}$, or on a computation that gives dn as an outermost constructor but then diverges while computing a $\mathbf{N}$.

Finally, the slogan conveys that a continuation is *nothing more* than a map from patterns to expressions. Starting from this definition, we cannot help but treat continuations extensionally, by their behavior on (maximal) value patterns. We do not care about the details of how these maps are defined. As we will see, this extensional view dramatically simplifies the presentation of the operational semantics of our programming language relative to typical presentations for comparable languages, and also forces our hand in defining the right notion of equality on continuations.

- Δ: We build a substitution simply by concatenating other substitutions (which ultimately are composed of continuations):

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\overset{\sigma_1}{\Gamma \vdash \Delta_1} \quad \overset{\sigma_2}{\Gamma \vdash \Delta_2}}{\Gamma \vdash (\Delta_1, \Delta_2)}$$

Because we can treat frames modulo associativity and unit (relying on the fact that the leaves of the frame are referenced by labels rather than paths), we can correspondingly think of these substitution constructors as monoidal operations, i.e., the concatenation of two substitutions (σ_1, σ_2) is an associative operator, and the empty substitution $\cdot$ is its unit. So, any substitution can simply be written as a list of continuations, $\sigma = (K_1, \ldots, K_n)$. This is for the *unannotated* view of frames. The annotated view associates each continuation with a continuation variable, $\sigma = (K_1/\kappa_1, \ldots, K_n/\kappa_n)$. But these two views are freely interchangeable, by the following easy observation:

Observation 4.2.3. *Any substitution $(K_1, \ldots, K_n) :: (\Gamma \vdash (\bullet A_1, \ldots, \bullet A_n))$ determines a map from continuation variables $\kappa : \bullet A \in (\kappa_1 : \bullet A_1, \ldots, \kappa_n : \bullet A_n)$ to continuations $K :: (\Gamma \vdash \bullet A)$, and conversely, given such a map we can build a substitution. We write $\sigma(\kappa)$ for this action of a Δ-substitution on a continuation variable in (an annotation of) Δ.*

- $\#$: There is a single rule for establishing contradiction in the positive fragment of PPL:

$$\frac{\kappa : \bullet A^+ \in \Gamma \quad \overset{V}{\Gamma \vdash A^+}}{\Gamma \vdash \#}$$

  This rule builds an expression $\kappa\ V$ by pairing an A-value with an A-continuation variable. Intuitively, in terms of conventional operational semantics, the expression $\kappa\ V$ is interpreted as passing the value V to the continuation denoted by κ at runtime. This is what Reynolds [1972] calls a "serious expression", because the resulting computation might do "serious" things (for example, diverge), according to κ's whim. Of course, in our language this is as yet only a metaphor for what κ could do *in potential*—we have not yet explained how to write programs that diverge, or that do anything interesting for that matter.

Before discussing the properties of $\mathcal{L}^+$ in detail, let us make a few easy observations, and examine a few simple programs.

Proposition 4.2.4 (Weakening)**.** *If $t :: (\Gamma \vdash J)$ then $t :: (\Gamma(\Delta) \vdash J)$.*

Proof. Trivial, by observing that our type-free notation is unaffected by the addition of extra frames into the context. Note, though, that we are relying on the fact that we have actual variables—with a de Bruijn approach, weakening requires index "shifting". □

Proposition 4.2.5. *The value-constructing rules*

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \mathsf{INL} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \mathsf{INR} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \otimes B}\ \mathsf{PAIR}$$

are admissible, defined by the transformations

$$\begin{aligned} \mathsf{INL}\,(p[\sigma]) &= (\mathsf{inl}\,p)[\sigma] \\ \mathsf{INR}\,(p[\sigma]) &= (\mathsf{inr}\,p)[\sigma] \\ \mathsf{PAIR}(p_1[\sigma_1], p_2[\sigma_2]) &= (p_1, p_2)[(\sigma_1, \sigma_2)] \end{aligned}$$

Proposition 4.2.6. *More generally, we can view standard value-constructors as syntactic sugar for combinators which operate on patterns and substitutions. Let c be an n-ary pattern rule:*

$$\frac{\Delta_1 \Vdash A_1 \quad \ldots \quad \Delta_n \Vdash A_n}{\Delta_1, \ldots, \Delta_n \Vdash B}\ \mathsf{c}$$

Then the value-constructing rule

$$\frac{\Gamma \vdash A_1 \quad \ldots \quad \Gamma \vdash A_n}{\Gamma \vdash B}\ \mathsf{C}$$

is admissible, defined by the operation $\mathsf{C}(p_1[\sigma_1], \ldots, p_n[\sigma_n]) = c(p_1, \ldots, p_n)[(\sigma_1, \ldots, \sigma_n)]$.

Proposition 4.2.7. *Any A-continuation K can be treated as a $\dashv$ A-value, by placing it in a singleton substitution, i.e., by building*

$$\_[K]$$

Proposition 4.2.8. *Any A-value V can be lifted to a $\dashv$ A-continuation that immediately applies its argument to V, i.e., the continuation K defined by*

$$K\ \kappa = \kappa\ V$$

As an exercise, the reader can try reconstructing the canonical derivations corresponding to each of the above terms, in the more verbose notation of Chapter 2.

Example 4.2.9. $\mathcal{L}^+$ forces us into writing programs in continuation-passing style. Where in a direct style language we would typically define a function $A \to B$, in $\mathcal{L}^+$ we define a *continuation transformer* from B continuations to A continuations. For example, to represent boolean conjunction, we can define a $\mathbf{B} \otimes \mathbf{B}$-continuation and_κ indexed by a continuation variable $\kappa : \bullet\mathbf{B}$, which takes in a pair of booleans and throws their binary product to κ. Likewise, we can define a $\mathbf{B}$-continuation not_κ, which takes in a boolean and throws its complement to κ. These continuations are defined by the following maps from value patterns to expressions (following Proposition 4.2.6 above, we write TT and FF as syntactic sugar for $\mathsf{tt}[\cdot]$ and $\mathsf{ff}[\cdot]$ respectively):

$$\begin{array}{rcl}
\mathit{and}_\kappa\ (\mathsf{tt}, \mathsf{tt}) & = & \kappa\ \mathsf{TT} \\
\mathit{and}_\kappa\ (\mathsf{tt}, \mathsf{ff}) & = & \kappa\ \mathsf{FF} \\
\mathit{and}_\kappa\ (\mathsf{ff}, \mathsf{tt}) & = & \kappa\ \mathsf{FF} \\
\mathit{and}_\kappa\ (\mathsf{ff}, \mathsf{ff}) & = & \kappa\ \mathsf{FF} \\
\\
\mathit{not}_\kappa\ \mathsf{tt} & = & \kappa\ \mathsf{FF} \\
\mathit{not}_\kappa\ \mathsf{ff} & = & \kappa\ \mathsf{TT}
\end{array}$$

Alternatively, we can define closed continuations that take the continuation variable κ as an extra component of the pattern. For example, we can define a closed $(\mathbf{B} \otimes \mathbf{B}) \otimes \neg\mathbf{B}$-continuation and^*:

$$\begin{array}{rcl}
\mathit{and}^*\ ((\mathsf{tt}, \mathsf{tt}), \kappa) & = & \kappa\ \mathsf{TT} \\
\mathit{and}^*\ ((\mathsf{tt}, \mathsf{ff}), \kappa) & = & \kappa\ \mathsf{FF} \\
\mathit{and}^*\ ((\mathsf{ff}, \mathsf{tt}), \kappa) & = & \kappa\ \mathsf{FF} \\
\mathit{and}^*\ ((\mathsf{ff}, \mathsf{ff}), \kappa) & = & \kappa\ \mathsf{FF}
\end{array}$$

■

Proposition 4.2.10. *Any* $\mathbf{N}$*-pattern* $\overline{n}$ *combined with the empty substitution builds a* $\mathbf{N}$*-value under any context, that is,* $\overline{n}[\cdot] :: (\Gamma \vdash \mathbf{N})$.

Example 4.2.11. We can define continuation transformers representing addition and multiplication on natural numbers. We build $\mathbf{N} \otimes \mathbf{N}$-continuations plus_κ and times_κ, indexed by a continuation variable $\kappa : \bullet\mathbf{N}$, defined by the following maps from value patterns to expressions:

$$\begin{aligned} \mathit{plus}_\kappa\ (\overline{n_1}, \overline{n_2}) &= \kappa\ \overline{n_1 + n_2}[\cdot] \\ \mathit{times}_\kappa\ (\overline{n_1}, \overline{n_2}) &= \kappa\ \overline{n_1 \times n_2}[\cdot] \end{aligned}$$

For example, $\mathit{plus}_\kappa\ (\overline{2}, \overline{3}) = \kappa\ \overline{5}[\cdot]$ and $\mathit{times}_\kappa\ (\overline{2}, \overline{3}) = \kappa\ \overline{6}[\cdot]$.

Alternatively, we can define closed $(\mathbf{N} \otimes \mathbf{N}) \otimes \neg\mathbf{N}$-continuations plus^* and times^* which take the continuation variable as an extra component of the pattern:

$$\begin{aligned} \mathit{plus}^*\ ((\overline{n_1}, \overline{n_2}), \kappa) &= \kappa\ \overline{n_1 + n_2}[\cdot] \\ \mathit{times}^*\ ((\overline{n_1}, \overline{n_2}), \kappa) &= \kappa\ \overline{n_1 \times n_2}[\cdot] \end{aligned}$$

■

The intention of Example 4.2.11 is hopefully clear—but the reader might have doubts about what it means formally, or how to generalize from it. Obviously, the definition of the maps $\mathit{plus}_\kappa, \mathit{times}_\kappa$, etc., presupposes some basic arithmetic. But then what, precisely, was meant when I wrote that "a continuation is a map from patterns to expressions"? Are these maps arbitrary set-theoretic functions? Recursive functions? Partial recursive? Because the space of value patterns is infinite for some types (such as $\mathbf{N}$ and $\mathbf{N} \otimes \mathbf{N}$), these can all be different classes of functions. For the simple examples above, it seems that defining continuations by recursive functions will do. But will it suffice in general?

And there is another (perhaps overlooked) ambiguity in our definition of terms: it is circular. Values are built out of substitutions, which are built out of continuations, which are built out of expressions, which are built out of values. So should we read it as an inductive definition? Do we suppose that terms are built out of finite applications of these rules? Again, this was the case in all of the examples above, but will it always work? Note that this question already arose for canonical derivations in the logical setting, but there we used the well-foundedness of the definition ordering to restrict the height of derivations.

As we will see, in order to preserve the identity and composition principles for canonical derivations in this operational setting—in particular where we can no longer rely on the definition ordering being well-founded—we have to give negative answers to both of these questions. We will adopt the following conventions:

1. Continuations are defined by partial recursive functions from value patterns to expressions.

2. Terms can be non-well-founded.

But these clarifications of the syntax of the language lead to an obvious next question,

4.2.4 Is this really syntax?

This somewhat ill-formed philosophical question requires a somewhat ill-formed philosophical answer. So let me offer one. Computer science has a long a tradition of building upon higher and higher levels of abstraction—both in terms of the domain of research (e.g., studying basic tree data structures, which are then used to study sorting and searching algorithms, which are then used to study process scheduling, etc.), and in terms of the construction of the computer itself. Olivier Danvy explains the latter situation well:

> Overall, a computer system is constructed inductively as a (finite) tower of interpreters, from the micro-code all the way up to the graphical user interface. Compilers and partial evaluators were invented to collapse interpretive levels because too many levels make a computer system impracticably slow. The concept of meta levels therefore is forced on computer scientists: I cannot make my program work, but maybe the bug is in the compiler? Or is it in the compiler that compiled the compiler? Maybe the misbehaviour is due to a system upgrade? Do we need to reboot? and so on. Most of the time, this kind of conceptual regression is daunting even though it is rooted in the history of the system at hand, and thus necessarily finite.[1]

In similar fashion, there has been a gradual progression towards higher levels of abstraction in what computer scientists view as legitimate descriptions of the syntax of a programming language. Consider how Alonzo Church began his definition of the λ-calculus:

> We select a particular list of symbols, consisting of the symbols {, }, (,), λ, [,], and an enumerably infinite set of symbols a, b, c, . . . to be called *variables.* And we define the word *formula* to mean any finite sequence of symbols out of this list. The terms *well-formed formula, free variable,* and *bound variable* are then defined by induction as follows. . . [Church, 1936]

In 1936, "syntax" meant strings, i.e., finite sequences of marks on a piece of paper or chalkboard, and Church gives a rather cumbersome description of the legal ways of forming strings representing λ-terms. A more convenient way of specifying well-formed strings was devised by John Backus and refined by Peter Naur, originally called Backus Normal Form and now known as Backus-Naur Form—basically a notation for context-free grammars. This allowed viewing syntax slightly more abstractly, as derivations in a context-free grammar (in other words as parse trees). But consider John McCarthy's words from 1963:

> The Backus normal form that is used in the ALGOL report, describes the morphology of ALGOL programs in a synthetic manner. Namely, it describes how the various kinds of program are built up from their parts. This would be better for translating into ALGOL than it is for the more usual problem of translating from ALGOL. The form of syntax we shall now describe differs from the Backus normal form in two ways. First, it is analytic rather than synthetic; it tells how to take a program apart, rather than how to put it together. Second, it is abstract in that it is independent of the notation used to represent, say sums, but only affirms that they can be recognized and taken apart. [McCarthy, 1963]

[1]Entry for "Self-interpreter", written by Olivier Danvy, in Appendix A to [Girard, 2001].

Most modern textbooks on compilers and programming languages follow McCarthy in making a distinction between *concrete syntax*—the well-formed strings of the programming language, with all the necessary curly braces, semicolons, etc., described by a BNF grammar—and *abstract syntax*. Abstract syntax still represents a program by a labelled tree, but a simpler one, without irrelevant information such as, say, whether statements are enclosed in square brackets or round parentheses. To be sure, this information only becomes irrelevant *after* we have already parsed the program's string encoding into a tree. And parsing is not a completely trivial problem. But once we have the parse tree, it becomes a distraction to have this extra information around if we want to *do* anything with the syntax, i.e., give it semantics. Many textbooks will only devote a couple paragraphs to concrete syntax, before moving on to abstract syntax.

What I am proposing is to take an even more abstract view of syntax, particularly allowing for *computation in syntax.* Not only does doing so highlight some neat symmetries, but it also makes it much easier to describe and reason about the operational behavior of programs, as we will see below. On the other hand, just as the abstract syntax tree representation of programs brushes aside some real issues (namely, parsing), so too does this functional representation. Rest assured, we will examine some of these issues in Chapter 5. Suffice it to say, if we build these higher and higher levels of abstraction, we have to be willing to compile them away.

4.2.5 Equality, operational semantics, and effects: overview

In his paper on "Notions of computations and monads", Eugenio Moggi contrasts *operational*, *denotational*, and *logical* approaches to reasoning about equivalence of programs:

- *The operational approach starts from an operational semantics, e.g., a partial function mapping every program. . . to its resulting value (if any), which induces a congruence relation on open terms called operational equivalence. . . Then the problem is to prove that two terms are operationally equivalent.*

- *The denotational approach gives an interpretation of the (programming) language in a mathematical structure, the intended model. Then the problem is to prove that two terms denote the same object in the intended model.*

- *The logical approach gives a class of possible models for the (programming) language. Then the problem is to prove that two terms denotes the same object in all possible models.*[2]

After explaining some of the shortcomings of the operational and denotational approaches, Moggi then goes on to introduce a logical approach to modelling computation, in categories with monads.

Of course, Moggi's definition of "logical" is biased towards the model-theoretic rather than the proof-theoretic view of logic. One of the aims of the following sections is to show how to give the operational approach a logical interpretation of the latter sort, deriving an operational semantics as a particular, simple cut-elimination algorithm for canonical derivations. Our other aim is, like Moggi, to use this semantics to explore different notions of computational effects.

We begin by translating the composition and identity principles for canonical derivations, defined in Chapter 2, to the notation of $\mathcal{L}^+$ terms. These principles are comparable to the standard notions of β-reduction and η-expansion in the λ-calculus, except insofar as terms are already in

[2][Moggi, 1991]

normal form. The composition principles (analogous to iterated β-reduction) are binary operations on terms, while the identity principles (analogous to iterated η-expansion) are particular terms. In this sense, the theory we derive is similar in spirit to categorical semantics: although we do not formally interpret terms as arrows of a category, we do show that the composition and identity principles satisfy (suitably formulated) associativity and unit properties.

But it is only an equational theory, rather than a realistic model of evaluation. For example, to define composition in general we must compose terms suspended within continuations, what is sometimes referred to as "evaluation under a lambda". To obtain a more realistic operational semantics, we investigate a special case of composition, iterated a finite number of times:

$$\frac{\cdot \vdash \Delta_1 \quad \Delta_1 \vdash \Delta_2 \quad \dots \quad \Delta_1, \dots, \Delta_{n-1} \vdash \Delta_n \quad \Delta_1, \dots, \Delta_n \vdash \#}{\cdot \vdash \#}$$

It happens that this n-ary composition principle is easier to define than the general, binary composition principles. Operationally, it corresponds to executing an open expression ($E :: (\Delta_1, \dots, \Delta_n \vdash \#)$) within a closed environment of substitutions ($\sigma_i :: (\Delta_1, \dots, \Delta_{i-1} \vdash \Delta_i)$, for $i = 1..n$). In this way, the operational semantics for our programming language arises naturally from its logical interpretation, as a simple cut-elimination algorithm. On the other hand, this n-ary composition principle also seems logically paradoxical: it results in a derivation of $\cdot \vdash \#$, i.e., a closed proof of contradiction!

Another way of viewing this situation is that the pure language $\mathcal{L}^+$, derived via the Curry-Howard correspondence, allows only one kind of expression: throwing a value to the continuation denoted by a variable. And what can that continuation do? Only throw a value to another continuation, which likewise must throw a value to another continuation, and so on. Thus the only way we can ever hope to instantiate the operational semantics is by building a program that loops forever, the environment growing indefinitely, i.e., a *circular* proof of contradiction. In other words, divergence is the only possible observable result of a closed program.

This fact could be used to dismiss the Curry-Howard interpretation of polarized logic as trivial, or alternatively (the view I support, of course) that it provides just the right starting point, a clean slate for investigating richer models of computation. Thus, the remainder of the section considers how to extend $\mathcal{L}^+$ with additional forms of observable behavior, or *side-effects.* Taking a cue from Girard's ludics, we begin by investigating the simplest possible side-effect: immediate failure. Now equipped with two forms of observable results, we can already define a non-trivial notion of observational equivalence, and relate it to syntactic, or *definitional* equality. One property proved by Girard [2001], which may seem counterintuitive, is that these two side-effects are sufficient to distinguish (i.e., prove observationally distinct) any two definitionally distinct "designs". Girard's designs are very similar to *affine* expressions in our language—expressions in which continuation variables are applied at most once—and we translate his result to that case. In the general case, more observations are needed: we explain how to extend the language with integer state, and show how this suffices to distinguish observationally any two $\mathcal{L}^+$ expressions that are definitionally distinct. We take this as one indication that $\mathcal{L}^+$ is indeed the right framework for investigations of computational effects under call-by-value.

4.2.6 Definitional equality

In order to reason about the properties of composition and of observational equivalence, we first have to clarify what we mean by "syntactic" or *definitional equality.* For two terms $t_1, t_2 :: (\Gamma \vdash J)$, we write $t_1 =_{\Gamma \vdash J} t_2$ (or just $t_1 = t_2$ when the form of judgment is clear from context) to indicate that t_1 and t_2 are the same term. The rules of definitional equality simply follow the rules of term construction:

$$\frac{p :: (\Delta \Vdash A^+) \quad \sigma_1 =_{\Gamma \vdash \Delta} \sigma_2}{p[\sigma_1] =_{\Gamma \vdash A^+} p[\sigma_2]} \qquad \frac{p :: (\Delta \Vdash A^+) \quad \longrightarrow \quad K_1(p) =_{\Gamma, \Delta \vdash \#} K_2(p)}{K_1 =_{\Gamma \vdash \bullet A^+} K_2}$$

$$\frac{}{\cdot =_{\Gamma \vdash \cdot} \cdot} \qquad \frac{\sigma_1 =_{\Gamma \vdash \Delta_1} \sigma_1' \quad \sigma_2 =_{\Gamma \vdash \Delta_2} \sigma_2'}{(\sigma_1, \sigma_2) =_{\Gamma \vdash (\Delta_1, \Delta_2)} (\sigma_1', \sigma_2')} \qquad \frac{\kappa : \bullet A^+ \in \Gamma \quad V_1 =_{\Gamma \vdash A^+} V_2}{\kappa\ V_1 =_{\Gamma \vdash \#} \kappa\ V_2}$$

This notion of equality has a more extensional flavor than typical notions of definitional equality, but that is forced on us by the functional representation of syntax. Implicit in these rules is that we have notions of equality for continuation variables and value patterns: equality of patterns is just equality of trees (relying on the unannotated view of frames) while equality of variables is α-equivalence (relying on the annotated view of frames). Finally, note that since we take continuations to be defined by *partial*-recursive functions from patterns to expressions, the rule for equality implicitly requires that both continuation maps terminate on the same set of patterns. Moreover, because we allow terms to be non-well-founded, we must also allow non-well-founded equality derivations.

Proposition 4.2.12 (Reflexivity)**.** *For all $t :: (\Gamma \vdash J)$, we have $t = t$.*

Proof. Immediate by recursion on t. □

Proposition 4.2.13 (Symmetry)**.** *If $t_1 = t_2$ then $t_2 = t_1$.*

Proof. Immediate by recursion on the derivation of $t_1 = t_2$. □

Proposition 4.2.14 (Transitivity)**.** *If $t_1 = t_2$ and $t_2 = t_3$ then $t_1 = t_3$.*

Proof. Immediate by recursion on the two equality derivations. □

4.2.7 Identity

The derivations of the two identity principles in §2.1.4 correspond to two particular terms. For any positive continuation variable $\kappa : \bullet A^+ \in \Gamma$, we build the *identity A-continuation Id_κ* $:: (\Gamma \vdash \bullet A^+)$, and likewise for any frame $\Delta \in \Gamma$, we build the *identity Δ-substitution $Id_{[\Delta]}$* $:: (\Gamma \vdash \Delta)$. The notation $[\Delta]$ is used here to denote the labels in a frame, and similarly we write $[p]$ for the labels in a pattern. Then we can build the identity continuation and the identity substitution according to the following mutually recursively definitions:

$$Id_\kappa\ p = \kappa\ (p[Id_{[p]}]) \qquad Id_{\cdot} = \cdot \qquad Id_{[\Delta_1, \Delta_2]} = (Id_{[\Delta_1]}, Id_{[\Delta_2]})$$

The reader can verify that these definitions correspond precisely to the derivations in §2.1.4. As per Theorem 2.1.11, these derivations are *not* necessarily well-founded, if the definition ordering is not well-founded. However, we can verify that these definitions are *productive*, that is, we can compute the structure of Id_κ and $Id_{[\Delta]}$ to arbitrary, possibly infinite precision. This is analogous to the situation in untyped λ-calculus, where η-expansion is represented by infinite Böhm trees.

4.2.8 Composition

For any positive value $V :: (\Gamma \vdash A^+)$ and positive continuation $K :: (\Gamma \vdash \bullet A^+)$, we can build an expression $K \bullet V :: (\Gamma \vdash \#)$ corresponding to the composition (reduction) principle. Likewise, for any term $t :: (\Gamma(\Delta) \vdash J)$ and Δ-substitution σ, we can build a term $t[\sigma] :: (\Gamma \vdash J)$ corresponding to the composition (substitution) principle. We build them according to the following mutually recursive definitions:

$$\begin{aligned}
K \bullet p[\sigma] &= K(p)[\sigma] \\
(\kappa\ V)[\sigma] &= \begin{cases} K \bullet (V[\sigma]) & \text{if } \sigma(\kappa) = K \\ \kappa\ (V[\sigma]) & \text{if } \kappa \notin \text{dom}(\sigma) \end{cases} \\
(p[\sigma_0])[\sigma] &= p[\sigma_0[\sigma]] \\
\cdot[\sigma] &= \cdot \\
(\sigma_1, \sigma_2)[\sigma] &= (\sigma_1[\sigma], \sigma_2[\sigma]) \\
K[\sigma] &= p \mapsto K(p)[\sigma]
\end{aligned}$$

Here $\sigma(\kappa)$ refers to the action of a substitution on variables in its frame, as defined in Observation 4.2.3. The notation $p \mapsto \ldots$ stands for a map over patterns (in the above, defining the continuation that sends any p to the expression $K(p)[\sigma]$). Again, the reader can verify that these definitions correspond precisely to the derivations of the composition principles in §2.1.4. But note the derivations for composition are neither well-founded in general, nor productive: in particular, the definitions of $K \bullet V$ and $E[\sigma]$ might never yield even the outermost variable κ of an expression. For notational convenience, we reify this possibility as a "pseudo-expression" Ω (pronounced "diverge"). Ω is also useful for making (partial-recursively defined) continuation maps total, letting us write $K(p) = \Omega$ when K diverges on pattern p. For the purpose of definitional equality, Ω is only equal to itself.

As an example of a composition that diverges, suppose we define a $\mathbf{D}$-continuation K by

$$K\ (\mathsf{dk}\,\kappa) \quad = \quad \kappa\ (\mathsf{DK}\,K)$$

(Recall that $\mathsf{DK}\,K$ is syntactic sugar for $\mathsf{dk}\_[K]$. We must also define $K\ (\mathsf{dn}\,\overline{n})$ for K to be exhaustive over $\mathbf{D}$-patterns, but the definition is irrelevant here.) Then $K \bullet \mathsf{DK}\,K = \Omega$.

4.2.9 Properties of composition

Lemma 4.2.15 (Unit laws)**.**

1. *$t[Id_{[\Delta]}] = t$*
   where $t :: (\Gamma, \Delta \vdash J)$

2. *$Id_\kappa \bullet V = \kappa\ V$*
   where $V :: (\Gamma \vdash A)$ and $\kappa : \bullet A \in \Gamma$

3. *$Id_{[\Delta]}[\sigma] = \sigma$*
   where $\sigma :: (\Gamma \vdash \Delta)$

Proof. We derive the three equalities by mutual recursion. For equality (1), we examine the form of t. In most cases we appeal to (1) on subterms of t and then derive the equality. When $t = \kappa\ V$, we have $\kappa\ V[Id_{[\Delta]}] = Id_\kappa \bullet V = \kappa\ V$ by (2). For equality (2), we know V is of the form $p[\sigma]$ for some $p :: (\Delta \Vdash A)$, and we have $Id_\kappa \bullet p[\sigma] = \kappa\ (p[Id_{[p]}])[\sigma] = \kappa\ (p[Id_{[p]}[\sigma]]) = \kappa\ (p[\sigma])$, with the last step by (3). Finally, to show equality (3), we recur over the structure of Δ, the interesting case being when it is a singleton $\kappa : \bullet B$. Then σ has the form (K/κ) for some K, and $Id_{[\Delta]}$ has the form (Id_κ/κ). To show $Id_\kappa[(K/\kappa)]$ and K are equal, we must apply the two sides to arbitrary value patterns $p' :: (\Delta' \Vdash B)$, and show they are equal at $\Gamma, \Delta' \vdash \#$. We have that

$$\begin{aligned} Id_\kappa[(K/\kappa)](p') &= (\kappa\ p'[Id_{[p']}])[(K/\kappa)] \\ &= K \bullet p'[(Id_{[p']}[(K/\kappa)])] \\ &= K \bullet p'[Id_{[p']}] && (*) \\ &= K(p')[Id_{[p']}] \\ &= K(p') && (1) \end{aligned}$$

where in $(*)$ we use the fact that $\kappa \notin [p'] = [\Delta']$. □

Lemma 4.2.16.

1. $(K \bullet V)[\sigma] = K[\sigma] \bullet V[\sigma]$
   where $\sigma :: (\Gamma \vdash \Delta)$ *and* $V :: (\Gamma, \Delta \vdash A)$ *and* $K :: (\Gamma, \Delta \vdash \bullet A)$

2. $(t[\sigma_2])[\sigma_1] = (t[\sigma_1])[\sigma_2[\sigma_1]]$
   where $\sigma_1 :: (\Gamma \vdash \Delta_1)$ *and* $\sigma_2 :: (\Gamma, \Delta_1 \vdash \Delta_2)$ *and* $t :: (\Gamma, \Delta_1, \Delta_2 \vdash J)$

Proof. For (1), we know V is of the form $p[\sigma_0]$, and we have that $(K \bullet p[\sigma_0])[\sigma] = (K(p)[\sigma_0])[\sigma] = (K(p)[\sigma])[\sigma_0[\sigma]] = K[\sigma] \bullet p[\sigma_0[\sigma]] = K[\sigma] \bullet (p[\sigma_0])[\sigma]$, with the second equality by (2). For (2), we examine the form of t. In most cases we appeal to (2) on subterms of t and derive the equality. When $t = \kappa\ V$ for some $\kappa \in \Delta_2$, we have

$$\begin{aligned} ((\kappa\ V)[\sigma_2])[\sigma_1] &= (\sigma_2(\kappa) \bullet V[\sigma_2])[\sigma_1] \\ &= \sigma_2(\kappa)[\sigma_1] \bullet (V[\sigma_2])[\sigma_1] && (1) \\ &= \sigma_2(\kappa)[\sigma_1] \bullet (V[\sigma_1])[\sigma_2[\sigma_1]] && (2) \\ &= (\kappa\ V[\sigma_1])[\sigma_2[\sigma_1]] \end{aligned}$$

□

Proposition 4.2.17 (Commutation). *Let* $\sigma_1 :: (\Gamma \vdash \Delta_1)$ *and* $\sigma_2 :: (\Gamma \vdash \Delta_2)$ *and* $t :: (\Gamma, \Delta_1, \Delta_2 \vdash J)$. *Then* $t[(\sigma_1, \sigma_2)] = t[\sigma_1][\sigma_2] = t[\sigma_2][\sigma_1]$.

Proof. Immediate. □

Corollary 4.2.18 (Associativity). *Let* $\sigma_1 :: (\Gamma \vdash \Delta_1)$ *and* $\sigma_2 :: (\Gamma, \Delta_1 \vdash \Delta_2)$ *and* $t :: (\Gamma, \Delta_1, \Delta_2 \vdash J)$. *Then* $(t[\sigma_1])[\sigma_2] = t[(Id_{[\Delta_2]}, \sigma_1)[\sigma_2]]$.

Proof. Combining the above facts we have

$$\begin{aligned}
t[(Id_{[\Delta_2]}, \sigma_1)[\sigma_2]] &= t[(Id_{[\Delta_2]}[\sigma_2], \sigma_1[\sigma_2])] && \text{(def.)}\\
&= t[(\sigma_2, \sigma_1[\sigma_2])] && \text{(Lemma 4.2.15(3))}\\
&= (t[\sigma_2])[\sigma_1[\sigma_2]] && \text{(Prop. 4.2.17)}\\
&= (t[\sigma_1])[\sigma_2] && \text{(Lemma 4.2.16(2))}
\end{aligned}$$

□

4.2.10 Complex variables

In Chapter 2 we introduced a notion of *complex hypotheses*, whose only role was to simplify the presentation of canonical derivations. For the same purpose, we can add them to $\mathcal{L}^+$. The idea is that a hypothesis $x : A$ in the context takes the place of explicit quantification over A-patterns. We call x a **complex value variable**. As a simple motivating example, suppose for instance that we want to define the projection functions on pairs as continuation transformers. For arbitrary positive types A and B, given continuation variables $\kappa_1 : \bullet A$ and $\kappa_2 : \bullet B$, we define the projections by the following maps on $A \otimes B$-patterns:

$$\begin{aligned}
\pi_1\ (p_1, p_2) &= \kappa_1\ (p_1[Id_{[p_1]}])\\
\pi_2\ (p_1, p_2) &= \kappa_2\ (p_2[Id_{[p_2]}])
\end{aligned}$$

Because the projection functions work generically over A and B, matching on p_1 and p_2 only to reconstruct them, we could write π_1 and π_2 more concisely using complex variables:

$$\begin{aligned}
\pi_1\ (x_1, x_2) &= \kappa\ Id_{x_1}\\
\pi_2\ (x_1, x_2) &= \kappa\ Id_{x_2}
\end{aligned}$$

Basically we would like to write the functions using *shallow* pattern-matching, rather than pattern-matching all the way down. This is really just syntactic sugar—the two versions of the functions behave exactly the same way, as they would in a strict language like ML—but it is convenient syntactic sugar. Formally, we include in $\mathcal{L}^+$ only a single construct related to complex hypotheses, which case-analyzes them away:

$$\text{case } x \text{ of } (p \mapsto t_p) = \frac{\overset{p}{\Delta \Vdash A^+} \quad \longrightarrow \quad \overset{t_p}{\Gamma(\Delta) \vdash J}}{\Gamma(x : A^+) \vdash J}$$

There is no explicit construct to introduce complex value variables, but we can *define* one via pattern substitution:

Proposition 4.2.19 (Pattern substitution). *If $t :: (\Gamma(x : A) \vdash J)$ and $p :: (\Delta \Vdash A)$ then $t[p/x]$ ::*

$(\Gamma(\Delta) \vdash J)$, *where* $t[p/x]$ *is defined as follows* (t^* *indicates a pattern-indexed term,* $t^* = p \mapsto t_p$)

$$\begin{aligned}
(\text{case } x \text{ of } t^*)[p/x] &= t^*(p) \\
(\text{case } y \text{ of } t^*)[p/x] &= \text{case } y \text{ of } (p' \mapsto t^*(p')[p/x]) \\
(p'[\sigma])[p/x] &= p'[\sigma[p/x]] \\
\cdot[p/x] &= \cdot \\
(\sigma_1, \sigma_2)[p/x] &= (\sigma_1[p/x], \sigma_2[p/x]) \\
K[p/x] &= (p' \mapsto K(p')[p/x]) \\
(\kappa\ V)[p/x] &= \kappa\ (V[p/x])
\end{aligned}$$

This is more or less the usual notion of capture-avoiding substitution, and importantly, unlike the composition principles, does not involve any "serious" computation (in Reynolds' sense).

To see how this give us license to build terms using shallow pattern-matching, consider again the above definitions. To build an $A \otimes B$-continuation in context Γ, we must give a map from patterns $p :: (\Delta \Vdash A \otimes B)$ to expressions in context Γ, Δ. By the definition of $A \otimes B$, this is equivalent to defining a map from patterns $p_1 :: (\Delta_1 \Vdash A)$ and $p_2 :: (\Delta_2 \Vdash B)$ to expressions in $\Gamma, \Delta_1, \Delta_2$. But by pattern substitution, this is equivalent to constructing a single expression in context $\Gamma, x_1 : A, x_2 : B$. So the two versions of π_1 and π_2 really are the same, with an implicit use of pattern substitution in the definitions by shallow pattern-matching.

Although complex value variables are basically syntactic sugar, we must still check that they cohere with our definitions of equality, identity, and composition. Two terms are equal in a context with complex hypotheses, if they are equal under all pattern substitutions:

$$\frac{\overset{p}{\Delta \Vdash A^+} \quad \longrightarrow \quad t_1[p/x] =_{\Gamma(\Delta)\vdash J} t_2[p/x]}{t_1 =_{\Gamma(x:A^+)\vdash J} t_2}$$

Given a complex value variable x, we construct the identity value Id_x by case analysis:

$$Id_x = \text{case } x \text{ of } p \mapsto p[Id_{[p]}]$$

And likewise, to compose two terms in a context with complex hypotheses ($V :: (\Gamma(x : B^+) \vdash A^+)$ and $K :: (\Gamma(x : B^+) \vdash \bullet A^+)$, or $t :: (\Gamma(x : B^+)(\Delta) \vdash J)$ and $\sigma :: (\Gamma(x : B^+) \vdash \Delta)$), we first do a case-analysis:

$$\begin{aligned}
K \bullet V &= \text{case } x \text{ of } (p \mapsto K[p/x] \bullet V[p/x]) \\
t[\sigma] &= \text{case } x \text{ of } (p \mapsto (t[p/x])[\sigma[p/x]])
\end{aligned}$$

Finally, we can substitute a value $V = p[\sigma]$ for a complex hypothesis x within a term t, simply by performing the pattern substitution $[p/x]$ followed by σ, i.e., $t[V/x] = (t[p/x])[\sigma]$.

4.2.11 Type isomorphisms

With a notion of identity and composition of terms, we can give a definition of *type isomorphism,* which is useful for formalizing our intuition that two types provide the same information.

Notation. *For two continuation transformers* $\phi :: (\kappa : \bullet B \vdash \bullet A)$ *and* $\psi :: (\kappa : \bullet C \vdash \bullet B)$*, we write* $\psi \circ \phi$ *as shorthand for* $\phi[\psi/\kappa] :: (\kappa : \bullet C \vdash \bullet A)$*. Similarly, for two value transformers* $\phi^* :: (x : A \vdash B)$ *and* $\psi^* :: (x : B \vdash C)$*, we write* $\psi^* \circ \phi^*$ *as shorthand for* $\psi^*[\phi^*/x] :: (x : A \vdash C)$*.*

Definition 4.2.20 (Type isomorphism)**.** *We say that two positive types A^+ and B^+ are isomorphic ($A \approx B$) if there exist a pair of continuation transformers $\phi :: (\kappa : \bullet B \vdash \bullet A)$ and $\psi :: (\kappa : \bullet A \vdash \bullet B)$ which are inverses, i.e., such that:*

1. $\psi \circ \phi =_{\kappa:\bullet A \vdash \bullet A} Id_\kappa$
2. $\phi \circ \psi =_{\kappa:\bullet B \vdash \bullet B} Id_\kappa$

Recall that in §2.3.3 we also defined a "positive" notion of entailment $A \vdash B$, in addition to the negative notion $\bullet B \vdash \bullet A$. We could do likewise here, and define another notion of type isomorphism by the existence of a pair of value transformers which are inverses. However, because of the requirement of *isomorphism*, this positive notion is equivalent to the negative one.

Proposition 4.2.21. *$A \approx B$ iff there exist a pair of value transformers $\phi^* :: (x : A \vdash B)$ and $\psi^* :: (x : B \vdash A)$ which are inverses, i.e., such that:*

1. $\psi^* \circ \phi^* =_{x:A \vdash A} Id_x$
2. $\phi^* \circ \psi^* =_{x:B \vdash B} Id_x$

Proof. The backwards direction is analogous to Proposition 2.3.9: we take $\phi = x \mapsto \kappa\ \phi^*$ and $\psi = x \mapsto \kappa\ \psi^*$. In the forwards direction, because $\psi \circ \phi = x \mapsto \kappa\ Id_x$ and $\phi \circ \psi = x \mapsto \kappa\ Id_x$, we know (by the definition of composition) that ϕ and ψ must be of the form $\phi = x \mapsto \kappa\ \phi^*$ and $\psi = x \mapsto \kappa\ \psi^*$ for some ϕ^* and ψ^*. Then because $\psi \circ \phi = x \mapsto \kappa\ (\psi^* \circ \phi^*)$ and $\phi \circ \psi = x \mapsto \kappa\ (\phi^* \circ \psi^*)$, we know that ϕ^* and ψ^* must be inverses. □

Proposition 4.2.22. *$\approx$ is reflexive, symmetric, and transitive.*

Proof. Symmetry is immediate, while reflexivity and transitivity are a consequence of the unit laws and associativity (Lemmas 4.2.15 and 4.2.18). Explicitly, we have $A \approx A$ for all A, because $Id_\kappa :: (\kappa : \bullet A \vdash \bullet A)$, and $Id_\kappa \circ Id_\kappa = Id_\kappa$. Suppose $A \approx B$ and $B \approx C$. Then there exist $\phi_1 :: (\kappa : \bullet B \vdash \bullet A)$ and $\psi_1 :: (\kappa : \bullet A \vdash \bullet B)$ such that $\phi_1 \circ \psi_1 = Id_\kappa$ and $\psi_1 \circ \phi_1 = Id_\kappa$, and likewise $\phi_2 :: (\kappa : \bullet C \vdash \bullet B)$ and $\psi_2 :: (\kappa : \bullet B \vdash \bullet C)$ such that $\phi_1 \circ \psi_2 = Id_\kappa$ and $\psi_2 \circ \phi_2 = Id_\kappa$. Then $(\phi_1 \circ \phi_2) :: (\kappa : \bullet C \vdash \bullet A)$ and $(\psi_2 \circ \psi_1) :: (\kappa : \bullet A \vdash \bullet C)$, and

$$\begin{aligned}
(\phi_1 \circ \phi_2) \circ (\psi_2 \circ \psi_1) &= \phi_1 \circ (\phi_2 \circ \psi_2) \circ \psi_1 \\
&= \phi_1 \circ Id_\kappa \circ \psi_1 \\
&= \phi_1 \circ \psi_1 \\
&= Id_\kappa \\
(\psi_2 \circ \psi_1) \circ (\phi_1 \circ \phi_2) &= \psi_2 \circ (\psi_1 \circ \phi_1) \circ \phi_2 \\
&= \psi_2 \circ Id_\kappa \circ \phi_2 \\
&= \psi_2 \circ \phi_2 \\
&= Id_\kappa
\end{aligned}$$

Hence $A \approx C$. □

Proposition 4.2.23 (cf. Laurent [2005])**.** *The following isomorphisms of positive types hold:*

$$A \otimes (B \otimes C) \approx (A \otimes B) \otimes C \qquad A \otimes B \approx B \otimes A \qquad A \approx A \otimes 1$$

$$A \oplus (B \oplus C) \approx (A \oplus B) \oplus C \qquad A \oplus B \approx B \oplus A \qquad A \approx A \oplus 0$$

$$A \otimes (B \oplus C) \approx (A \otimes B) \oplus (A \otimes C) \qquad A \otimes 0 \approx 0$$

$$\overset{\pm}{\neg}(A \oplus B) \approx \overset{\pm}{\neg} A \otimes \overset{\pm}{\neg} B \qquad \overset{\pm}{\neg}\, 0 \approx 1$$

Proof. Routine calculations. □

4.2.12 Environment semantics

As noted in the introduction to this section, the composition and identity principles for terms provide an equational theory comparable to $\beta\eta$-conversion for the λ-calculus, but not a realistic account of evaluation. We now describe a more conventional operational semantics based on *environments*. We deliberately exclude complex hypotheses from the environment semantics, because they do not play an interesting computational role.

Definition 4.2.24 (Environments). *An* **environment** γ *for context* Γ *is built as follows:*

$$\frac{}{\cdot}\ \mathsf{emp} \qquad \frac{\overset{\gamma}{\Gamma} \quad \overset{\sigma}{\Gamma \vdash \Delta}}{\Gamma, \Delta}\ \mathsf{bind}$$

In other words, an environment is an ordered list of substitutions, each of which can reference variables bound by prior substitutions. We use the notation $(\gamma;\sigma)$ *as an abbreviation for* $\mathsf{bind}(\gamma,\sigma)$*, and* $(\sigma_1;\ldots;\sigma_n)$ *as an abbreviation for* $\mathsf{bind}(\ldots\mathsf{bind}(\mathsf{bind}(\mathsf{emp},\sigma_1),\sigma_2)\ldots,\sigma_n)$.

Proposition 4.2.25. *Let* $\gamma = (\sigma_1;\ldots;\sigma_n)$ *be a* Γ*-environment, for* $\Gamma = \Delta_1,\ldots,\Delta_n$*, and let* $t :: (\Gamma \vdash J)$*. Then* $(((t[\sigma_n])[\sigma_{n-1}])\ldots)[\sigma_1] :: (\cdot \vdash J)$.

Proof. By n applications of the composition (substitution) principle, as we can illustrate like so:

$$\dfrac{\overset{\sigma_1}{\cdot \vdash \Delta_1} \qquad \dfrac{\overset{\sigma_2}{\Delta_1 \vdash \Delta_2} \qquad \overset{\displaystyle \dfrac{\overset{\sigma_n}{\Delta_1,\ldots,\Delta_{n-1} \vdash \Delta_n} \quad \overset{t}{\Delta_1,\ldots,\Delta_n \vdash J}}{\Delta_1,\ldots,\Delta_{n-1} \vdash J}}{\overset{\iddots}{\Delta_1,\Delta_2 \vdash J}}}{\Delta_1 \vdash J}}{\cdot \vdash J}$$

□

Notation. *We write* $t[\gamma]$ *for the* n*-fold composition* $(((t[\sigma_n])[\sigma_{n-1}])\ldots)[\sigma_1]$ *defined in Prop. 4.2.25.*

Proposition 4.2.25 tells us that a Γ-environment γ and an expression $E :: (\Gamma \vdash \#)$ can be combined to build a closed expression $E[\gamma]$. But rather than evaluating $E[\gamma]$ by performing this n-fold composition, we will define a small-step operational semantics to compute it directly.

Proposition 4.2.26. *Given a* Γ*-environment* $\gamma = (\sigma_1;\ldots;\sigma_n)$ *and a continuation variable* $\kappa : \bullet A \in \Gamma$*, there is some* σ_i *such that* $\sigma_i(\kappa) :: (\Gamma \vdash \bullet A)$.

Proof. Since $\kappa : \bullet A \in \Gamma$, it must be in Δ_i for some $1 \leq i \leq n$. Since $\sigma_i :: (\Delta_1, \ldots, \Delta_{i-1} \vdash \Delta_i)$, we have $\sigma_i(\kappa) :: (\Delta_1, \ldots, \Delta_{i-1} \vdash \bullet A)$ by definition. Hence $\sigma_i(\kappa) :: (\Gamma \vdash \bullet A)$ by weakening. $\square$

Notation. *We write* $\mathsf{lookup}(\gamma, \kappa) :: (\Gamma \vdash \bullet A)$ *for the continuation* $\sigma_i(\kappa)$ *selected by Prop. 4.2.26, or when we are feeling terse, simply* $\gamma(\kappa)$*. Explicitly,* $\mathsf{lookup}(\gamma, \kappa)$ *is defined as follows:*

$$\mathsf{lookup}(\mathsf{bind}(\gamma, \sigma), \kappa) = \begin{cases} \sigma(\kappa) & \kappa \in \mathrm{dom}(\sigma) \\ \mathsf{lookup}(\gamma, \kappa) & \kappa \in \mathrm{dom}(\gamma) \end{cases}$$

Definition 4.2.27 (Programs)**.** *A* **program** *is either a pair* $\langle \gamma \mid E \rangle$ *of a* Γ*-environment* γ *and an expression* $E :: (\Gamma \vdash \#)$*, or a triple* $\langle \gamma \mid K \mid V \rangle$ *of a* Γ*-environment* γ*, a continuation* $K :: (\Gamma \vdash \bullet A)$ *and a value* $V :: (\Gamma \vdash A)$*. We use the letter* P *to range over programs.*

Definition 4.2.28 (Environment semantics)**.** *A small-step* **environment semantics** *is a relation* $P \rightsquigarrow P'$ *between programs. A* **result** *is a program* P *such that* $P \not\rightsquigarrow P'$ *for any* P'*, or* Ω*. The small-step semantics induces a big-step evaluation relation* $P \Downarrow R$ *between programs and results, defined inductively as follows:*

$$\frac{P \rightsquigarrow P' \quad P' \Downarrow R}{P \Downarrow R} \qquad \frac{P \textit{ result}}{P \Downarrow P}$$

We reify the possibility that a program diverges by writing $P \Downarrow \Omega$*.*

For clarity, we will distinguish the language $\mathcal{L}^+$ as described thus far, as well as its *total* fragment, from further extensions of $\mathcal{L}^+$ with effects.

Definition 4.2.29 (Purity and totality)**.** *We refer to arbitrary terms of* $\mathcal{L}^+$ *(i.e., possibly non-well-founded derivations involving positive types, where continuations are defined by partial-recursive functions, and including the pseudo-expression* Ω *representing divergence) as* **pure** *terms. We refer to* $\mathcal{L}^+$ *terms with continuations defined by total-recursive functions, and without the pseudo-expression* Ω*, as* **total** *terms.*

The small-step environment semantics for total $\mathcal{L}^+$ is given by a pair of rules:

$$\begin{aligned} \langle \gamma \mid \kappa\ V \rangle &\quad\rightsquigarrow\quad \langle \gamma \mid \mathsf{lookup}(\gamma, \kappa) \mid V \rangle && (\mathrm{lookup}^+) \\ \langle \gamma \mid K \mid p[\sigma] \rangle &\quad\rightsquigarrow\quad \langle \mathsf{bind}(\gamma, \sigma) \mid K(p) \rangle && (\mathrm{bind/call}^+) \end{aligned}$$

In pure $\mathcal{L}^+$, we add a single rule for the pseudo-expression Ω:

$$\langle \gamma \mid \Omega \rangle \quad\rightsquigarrow\quad \langle \gamma \mid \Omega \rangle \qquad (\mathrm{loop})$$

Clearly, the two rules lookup^+ and $\mathrm{bind/call}^+$ can be refactored into a single rule:

$$\langle \gamma \mid \kappa\ (p[\sigma]) \rangle \quad\rightsquigarrow\quad \langle \mathsf{bind}(\gamma, \sigma) \mid \mathsf{lookup}(\gamma, \kappa)(p) \rangle \qquad (\mathrm{go}^+)$$

which is sometimes expedient when we don't care about the intermediate transition. This compound rule can be glossed as,

> *To execute* $\kappa\ (p[\sigma])$ *in environment* γ*, first look up the continuation* K *the environment associates with* κ*, then add* σ *to the environment, and finally execute the expression* $K(p)$

Because this is an *intrinsic* semantics—that is, it is defined on programs that are well-typed by construction—a type safety theorem in the standard sense is unnecessary. However, we can state stronger versions of the usual "progress" and "preservation" lemmas.

Lemma 4.2.30 (Intrinsic progress). *For all programs P in pure $\mathcal{L}^+$, there exists a P' such that $P \rightsquigarrow P'$.*

Proof. Immediate. All pure expressions $E :: (\Gamma \vdash \#)$ are either Ω (we can invoke loop) or of the form $E = \kappa\ V$, where $\kappa : \bullet A \in \Gamma$ and $V :: (\Gamma \vdash A)$ (we can invoke lookup^+). Likewise, all pure values $V :: (\Gamma \vdash A)$ are of the form $V = p[\sigma]$, where $p :: (\Delta \Vdash A)$ and $\sigma :: (\Gamma \vdash \Delta)$ (we can invoke $\mathrm{bind/call}^+$). □

The intrinsic progress lemma has a surprising consequence: in pure $\mathcal{L}^+$, there is only one possible result (Ω), and in total $\mathcal{L}^+$ there are none!

Lemma 4.2.31. $t[\mathsf{bind}(\gamma, \sigma)] = (t[\gamma])[\sigma[\gamma]]$

Proof. By Lemma 4.2.16. □

Lemma 4.2.32 (Intrinsic preservation). *For pure $\mathcal{L}^+$, if $\langle \gamma \mid E \rangle \rightsquigarrow \langle \gamma \mid K \mid V \rangle \rightsquigarrow \langle \gamma' \mid E' \rangle$, then $E[\gamma] = K \bullet V[\gamma] = E'[\gamma']$.*

Proof. The transitions are by the lookup^+ and $\mathrm{bind/call}^+$ rules, so that $E = \kappa\ V$, $V = p[\sigma]$, $K = \gamma(\kappa)$, $\gamma' = (\gamma; \sigma)$, and $E' = K(p)$. We have:

$$\begin{aligned}(\kappa\ V)[\gamma] &= \gamma(\kappa)[\gamma] \bullet V[\gamma] \\ &= (\gamma(\kappa)(p)[\gamma])[\sigma[\gamma]] \\ &= \gamma(\kappa)(p)[(\gamma; \sigma)]\end{aligned}$$

with the last step by Lemma 4.2.31. □

We can apply intrinsic preservation to show that the environment semantics indeed implements the desired behavior of n-fold composition.

Corollary 4.2.33 (Functionality). *For pure $\mathcal{L}^+$, if $\langle \gamma \mid E \rangle \Downarrow R$ then $E[\gamma] = R$.*

4.2.13 Observational equivalence

The environment semantics leads to a natural definition of *observational equivalence.*

Definition 4.2.34 (Observational equivalence). *Let $E_1, E_2 :: (\Gamma \vdash \#)$ be two expressions. We say that E_1 and E_2 are* **observationally equivalent** *(written $E_1 \cong E_2$) if for all Γ-environments γ and all results R, we have $\langle \gamma \mid E_1 \rangle \Downarrow R$ iff $\langle \gamma \mid E_2 \rangle \Downarrow R$.*

It is easy to see that definitional equality implies observational equivalence.

Definition 4.2.35 (Equality of environments). *Two Γ-environments $\gamma = (\sigma_1; \ldots; \sigma_n)$ and $\gamma' = (\sigma_1'; \ldots; \sigma_n')$ are definitionally equal (written $\gamma =_\Gamma \gamma'$) if $\sigma_i = \sigma_i'$ for all $1 \leq i \leq n$.*

Proposition 4.2.36. *If $\gamma =_\Gamma \gamma'$ then $\gamma(\kappa) =_{\Gamma, \Delta \vdash \#} \gamma'(\kappa)$ for all $\kappa : \bullet A \in \Gamma$.*

Lemma 4.2.37 (Congruence). *If $\langle \gamma_1 \mid E_1 \rangle \Downarrow R$ and $E_1 = E_2$ and $\gamma_1 = \gamma_2$ then $\langle \gamma_2 \mid E_2 \rangle \Downarrow R$.*

Proof. For pure $\mathcal{L}^+$, we know the derivation must have the form:

$$\frac{\langle \gamma_1 \mid E_1 \rangle \rightsquigarrow \langle \gamma_1' \mid E_1' \rangle \quad \langle \gamma_1' \mid E_1' \rangle \Downarrow R}{\langle \gamma_1 \mid E_1 \rangle \Downarrow R}$$

with the transition $\langle \gamma_1 \mid E_1 \rangle \rightsquigarrow \langle \gamma_1' \mid E_1' \rangle$ either by loop or decomposed by the go^+ compound rule. In the former case we have $E_1 = E_2 = R = \Omega$, and the statement is trivial. In the latter case, we know $E_1 = \kappa\ (p[\sigma_1])$, $E_1' = \gamma_1(\kappa)(p)$, and $\gamma_1' = (\gamma_1; \sigma_1)$. Since $E_1 = E_2$, we know that $E_2 = \kappa\ (p[\sigma_2])$ for some $\sigma_2 = \sigma_1$. Then $\langle E_2 \mid \gamma_2 \rangle \rightsquigarrow \langle (\gamma_2; \sigma_2) \mid \gamma_2(\kappa)(p) \rangle$ by go^+. Since $(\gamma_1; \sigma_1) = (\gamma_2; \sigma_2)$ by definition, and $\gamma_1(\kappa)(p) = \gamma_2(\kappa)(p)$ by Proposition 4.2.36, we have $\langle E_2 \mid \gamma_2 \rangle \Downarrow R$ by appeal back to congruence. □

As an immediate result of congruence, we have:

Theorem 4.2.38. *If $E_1 = E_2$ then $E_1 \cong E_2$.*

The more interesting question is, how coarse is observational equivalence? We already remarked that in pure $\mathcal{L}^+$, there is only one possible result (divergence), so the relation $\cong$ is indeed trivial. But as we add more kinds of observations to the language, we get a progressively finer notion of observational equivalence on pure expressions, until it eventually coincides with definitional equality.

4.2.14 Immediate failure, and the chronicle representation

We extend $\mathcal{L}^+$ with a single rule:

$$\frac{}{\Gamma \vdash \#}\ \mho$$

The expression $\mho$ is pronounced "fail". We give no additional transition rules, so that $\mho$ is a possible result, which can be thought of as representing some sort of safety violation: an uncaught exception, an abort, a hardware crash, etc. Obviously, $\mho$ is a *new* kind of result, distinct from Ω, so formally we declare that $\mho$ is only definitionally equal to itself. We write $\mathcal{L}^+_\mho$ for this extension of $\mathcal{L}^+$, and $\cong_\mho$ for the derived observational equivalence relation on expressions of $\mathcal{L}^+_\mho$. Note that this also gives us a new equivalence relation on pure $\mathcal{L}^+$ expressions, by restriction.

Logically, the rule $\mho$ corresponds to immediately asserting a contradiction under any set of assumptions: a plainly inconsistent reasoning principle. Girard [2001] introduced the same rule, which he calls *daimon*, in the setting of ludics. There, he proved a statement (the Separation Theorem, an analogue of Boehm's theorem for λ-calculus [1968]) which essentially says that Ω and $\mho$ are sufficient effects to distinguish observationally any two "designs" which are definitionally distinct. Here, we show that this holds for *affine* expressions.

Definition 4.2.39 (Affineness)**.** *We say that a term $t :: (\Gamma \vdash J)$ is* **affine** *when*

1. *For every subterm $\kappa\ V$ of t, κ is not free in V, and*
2. *For every subterm (σ_1, σ_2) of t, σ_1 and σ_2 have disjoint free variables.*

In order to prove this result about definitional equality of affine expressions, as well as to pave the way for the result about arbitrary expressions in the next section, we introduce (analogues of) several technical notions from ludics, starting with that of *chronicle.*

Definition 4.2.40 (Chronicles)**.** *A* **proper action** *is a tuple (κ, p) of a continuation variable κ and a pattern p of the same type, i.e., $\kappa : \bullet A$ and $p :: (\Delta \Vdash A)$ for some A and Δ. An* **improper action** *is either Ω or $\mho$. We use α to range over actions, proper or improper. A* **chronicle** *$\mathfrak{c}$ is a sequence of actions $\mathfrak{c} = \alpha_1 \cdots \alpha_n$.*

Girard defines chronicles near the start of "Locus Solum", in order to construct the basic proof objects of ludics (designs) as special sets of chronicles. Because we already have a working language of proofs ($\mathcal{L}^+$ and its extension $\mathcal{L}^+_\mho$), we shall view chronicles more as derived artifacts. Concretely, our definition omits many of the side-conditions that are part of the definition of chronicles in ludics—instead, we can show that they are derived properties of chronicles in the *chronicle representation* of a term.

Notation. *We write (κ, p, σ) as shorthand for $\kappa\ (p[\sigma])$, and $\sigma\ \kappa'\ p'$ as shorthand for $\sigma(\kappa')(p')$.*

Definition 4.2.41 (Chronicle representation)**.** *Let t be an expression or substitution in $\mathcal{L}^+_\mho$. We define a set of chronicles $|t|$ (called the* **chronicle representation** *of t) as follows:*

$$\frac{}{\cdot \in |t|}$$

$$\frac{}{\Omega \in |\Omega|} \qquad \frac{}{\mho \in |\mho|}$$

$$\frac{\mathfrak{c} \in |\sigma|}{(\kappa, p) \cdot \mathfrak{c} \in |(\kappa, p, \sigma)|} \qquad \frac{\mathfrak{c} \in |\sigma\ \kappa'\ p'|}{(\kappa', p') \cdot \mathfrak{c} \in |\sigma|}$$

If $\alpha \cdot \mathfrak{c} \in |t|$, we say that α is **positive** *if t is an expression, or* **negative** *if t is a substitution. Thus, the odd-numbered (even-numbered) actions of a chronicle in an expression (substitution) are positive, and even-numbered (odd-numbered) actions are negative.*

We can see that some of the conditions placed on chronicles in ludics are obvious properties of chronicles in the chronicle representation of a $\mathcal{L}^+_\mho$ expression or substitution.

Proposition 4.2.42 (Propriety)**.** *For $\mathfrak{c} \in |t|$, all actions before the last are proper.*

Proposition 4.2.43 (Positive/negative actions)**.** *Let $\mathfrak{c} \in |t|$, where t is either an expression $E :: (\Gamma \vdash \#)$ or a substitution $\sigma :: (\Gamma \vdash \Delta'_1)$. Suppose we label the (possibly empty) sequence of proper actions in $\mathfrak{c}$ by:*

$$\begin{array}{ll} (\kappa_1, p_1) \cdot (\kappa'_1, p'_1) \cdot (\kappa_2, p_2) \cdots & \text{if } t = E \\ (\kappa'_1, p'_1) \cdot (\kappa_2, p_2) \cdot (\kappa'_2, p'_2) \cdots & \text{if } t = \sigma \end{array}$$

Then there is some collection of frames $\Delta'_1, \Delta_2, \Delta'_2, \ldots$ and types $A_1, A'_1, A_2, A'_2, \ldots$, such that for all proper actions (κ_i, p_i) and (κ'_i, p'_i) in $\mathfrak{c}$:

$$\begin{array}{rl} \kappa_i : \bullet A_i \in \Gamma, \Delta_2, \ldots, \Delta_i & p_i :: (\Delta'_i \Vdash A_i) \\ \kappa'_i : \bullet A'_i \in \Delta'_i & p'_i :: (\Delta_{i+1} \Vdash A'_i) \end{array}$$

We should remark that the condition on chronicles in ludics that "foci" (continuation variables) be pairwise distinct does *not* hold for chronicles coming from arbitrary $\mathcal{L}^+_\mho$ terms. By renaming of variables we can assume that the κ'_i coming from *negative* actions are distinct from all other κ'_j and κ_j. However, the condition that $\kappa_i \neq \kappa_j$ for $i \neq j$ requires the affineness restriction. We can verify some additional properties of chronicle representations, which in ludics are part of the definition of designs.

Proposition 4.2.44 (Arborescence)**.** *Chronicle representations are closed under restriction. Thus we can say that $|t|$ is generated by its* **maximal chronicles***, i.e., $\mathfrak{c} \in |t|$ with no extension in $|t|$.*

Proposition 4.2.45 (Coherence)**.** *If $\mathfrak{c}, \mathfrak{c}' \in |t|$, then either one extends the other, or they first differ on negative actions, that is, $\mathfrak{c} = \mathfrak{c}_0 \cdot \alpha \cdot \mathfrak{c}_1$, $\mathfrak{c}' = \mathfrak{c}_0 \cdot \alpha' \cdot \mathfrak{c}_2$, with $\alpha \neq \alpha'$ negative.*

Proposition 4.2.46 (Positivity)**.** *The last action of a maximal chronicle must be positive. Moreover, if it comes from a pure term, it must be proper or Ω. If it comes from a total term, it must be proper.*

Example 4.2.47. Consider the $(\mathbf{N} \otimes \mathbf{N}) \otimes \neg\mathbf{N}$-continuation *plus** defined in Example 4.2.11. The chronicle representation of the singleton substitution $(plus^*/\kappa')$ is generated by chronicles of the form

$$(\kappa', ((\overline{n}_1, \overline{n}_2), \kappa_2)) \cdot (\kappa_2, \overline{n_1 + n_2})$$

The expression $\kappa_1\ (\kappa'[(plus^*/\kappa')])$ (where κ_1 is some $\neg((\mathbf{N} \otimes \mathbf{N}) \otimes \neg\mathbf{N})$-continuation variable in the context) has maximal chronicles of the form

$$(\kappa_1, \kappa') \cdot (\kappa', ((\overline{n}_1, \overline{n}_2), \kappa_2)) \cdot (\kappa_2, \overline{n_1 + n_2})$$

■

The main reason we are interested in chronicle representations is that they give us a better handle on how one term *approximates* (or fails to approximate) another term.

Definition 4.2.48 (Definitional approximation)**.** *For two terms $t_1, t_2 :: (\Gamma \vdash J)$ of $\mathcal{L}^+_{\mho}$, we say that t_1* **approximates** *t_2 (written $t_1 \leq t_2$) if t_2 is obtained from t_1 by replacing some expressions Ω by expressions of the form $\kappa\ V$, and some expressions of the form $\kappa\ V$ by $\mho$. In other words, the approximation relation $\leq$ is generated the same way as the equality relation $=$, except that we take Ω and $\mho$ as least and greatest expressions, respectively.*

Proposition 4.2.49. *$t_1 = t_2$ iff $t_1 \leq t_2$ and $t_2 \leq t_1$*

Definition 4.2.50. *We impose an ordering on positive actions:*

$$\Omega < (\kappa, p) < \mho$$

We take distinct proper actions (κ_1, p_1) and (κ_2, p_2) to be incomparable. We write $\alpha \leq \alpha'$ if either $\alpha < \alpha'$ or $\alpha = \alpha'$, and $\alpha \not\leq \alpha'$ if neither of these holds.

Note that we have to be a bit careful about what we mean when we say that (κ_1, p_1) and (κ_2, p_2) are *distinct*: in the appropriate context, either $\kappa_1 \neq \kappa_2$ (modulo renaming) or $p_1 \neq p_2$ (ignoring the names of variables).

Proposition 4.2.51. *If $t_1 \leq t_2$ then for any $\mathfrak{c} \in |t_1|$, either $\mathfrak{c} \in |t_2|$, or else there is some prefix $\mathfrak{c}_0 \cdot \alpha$ of $\mathfrak{c}$, where α is positive and $\alpha < \alpha'$, such that $\mathfrak{c}_0 \cdot \alpha' \in |t_2|$.*

Proof. By induction on the derivation of $\mathfrak{c} \in |t_1|$. □

Proposition 4.2.52. *If $t_1 \not\leq t_2$ then there is some $\mathfrak{c}_0$ and positive actions $\alpha \not\leq \alpha'$, such that $\mathfrak{c}_0 \cdot \alpha \in |t_1|$ and $\mathfrak{c}_0 \cdot \alpha' \in |t_2|$.*

Proof. By recursion on t_1 and t_2. From the assumption that $t_1 \not\lesssim t_2$, we must have one of the situations below. Note that for pure terms, only cases (3)–(6) are relevant, while for total expressions only (4)–(6).

1. $t_1 = \mho$, $t_2 = \Omega$: take $\mathfrak{c}_0 = \cdot$, $\alpha = \mho$, $\alpha' = \Omega$
2. $t_1 = \mho$, $t_2 = (\kappa, p, \sigma)$: take $\mathfrak{c}_0 = \cdot$, $\alpha = \mho$, $\alpha' = (\kappa, p)$
3. $t_1 = (\kappa, p, \sigma)$, $t_2 = \Omega$: take $\mathfrak{c}_0 = \cdot$, $\alpha = (\kappa, p)$, $\alpha' = \Omega$
4. $t_1 = (\kappa, p_1, \sigma_1)$, $t_2 = (\kappa_2, p_2, \sigma_2)$ where $\kappa_1 \neq \kappa_2$ or $p_1 \neq p_2$: take $\mathfrak{c}_0 = \cdot$, $\alpha = (\kappa_1, p_1)$, $\alpha' = (\kappa_2, p_2)$
5. $t_1 = (\kappa, p, \sigma_1)$, $t_2 = (\kappa, p, \sigma_2)$ where $\sigma_1 \not\lesssim \sigma_2$: from $\sigma_1 \not\lesssim \sigma_2$, we obtain $\mathfrak{c}_0'$ and $\alpha \not\lesssim \alpha'$ such that $\mathfrak{c}_0' \cdot \alpha \in |\sigma_1|$ and $\mathfrak{c}_0' \cdot \alpha' \in |\sigma_2|$. Take $\mathfrak{c}_0 = (\kappa, p) \cdot \mathfrak{c}_0'$.
6. $t_1 = \sigma_1$, $t_2 = \sigma_2$, where $\sigma_1 \not\lesssim \sigma_2$: by definition, there is some κ' and p' such that $\sigma_1\ \kappa'\ p' \not\lesssim \sigma_2\ \kappa'\ p'$, and hence we can obtain $\mathfrak{c}_0'$ and $\alpha \not\lesssim \alpha'$ such that $\mathfrak{c}_0' \cdot \alpha \in |\sigma_1\ \kappa'\ p'|$ and $\mathfrak{c}_0' \cdot \alpha' \in |\sigma_2\ \kappa'\ p'|$. Take $\mathfrak{c}_0 = (\kappa', p') \cdot \mathfrak{c}_0'$.

□

We apply Proposition 4.2.52 to prove the main result.

Lemma 4.2.53. *Let $E_1, E_2 :: (\Gamma \vdash \#)$ be two affine $\mathcal{L}^{+}_{\mho}$ expressions, and suppose that $E_1 \not\lesssim E_2$. Then there exists an environment γ in $\mathcal{L}^{+}_{\mho}$ such that $\langle E_1 \mid \gamma \rangle \Downarrow \mho$ and $\langle E_2 \mid \gamma \rangle \Downarrow \Omega$.*

Proof. For notational convenience, let's relabel these expressions $E^* = E_1$ and $E^\dagger = E_2$. Then there is some $\mathfrak{c}_0 \cdot \alpha^* \in |E^*|$ and $\mathfrak{c}_0 \cdot \alpha^\dagger \in |E^\dagger|$, where $\alpha^* \not\lesssim \alpha^\dagger$. By propriety, we know that $\mathfrak{c}_0$ must be a sequence of only proper actions $(\kappa_1, p_1) \cdot (\kappa_1', p_1') \cdots (\kappa_n, p_n) \cdot (\kappa_n', p_n')$, together with the appropriate conditions on the contexts $\Gamma, \Delta_1', \Delta_2, \ldots, \Delta_{n+1}$ expressed in Prop. 4.2.43. Now, the idea is that we build the Γ-environment γ so that when paired with E^* or $E^\dagger$, it "walks through" these proper actions, until it gets to either α^* or $\alpha^\dagger$, which are distinguishable.

Without loss of generality, we can view Γ as an annotated frame $\Delta_1 = \Gamma$, and build γ as a closed Δ_1-substitution σ_1' rather than an environment. We construct σ_1' together with a series of closed substitutions $\sigma_2', \ldots, \sigma_{n+1}'$, such that

$$\sigma_i' :: (\cdot \vdash \Delta_i)$$

This maintains the invariant that for any substitutions $\sigma_1, \ldots, \sigma_k$, where

$$\sigma_i :: (\Delta_1, \Delta_1', \ldots, \Delta_{i-1}', \Delta_{i-1}', \Delta_i \vdash \Delta_i')$$

we have that $(\sigma_1'; \sigma_2; \sigma_2'; \ldots; \sigma_k; \sigma_{k+1}')$ is a $(\Delta_1, \Delta_1', \Delta_2, \ldots, \Delta_k', \Delta_{k+1})$-environment.

We start by setting

$$\sigma_1'\ \kappa_1\ p_1 = (\kappa_1', p_1', \sigma_2')$$

leaving σ_2' unspecified for now. Since $(\kappa_1, p_1) \cdot (\kappa_1', p_1') \cdots \in |E^*|, |E^\dagger|$, we know that both

$$E^* = (\kappa_1, p_1, \sigma_1^*) \quad \text{and} \quad \sigma_1^*\ \kappa_1'\ p_1' = E_2^*$$

and

$$E^\dagger = (\kappa_1, p_1, \sigma_2^\dagger) \quad \text{and} \quad \sigma_2^\dagger\ \kappa_1'\ p_1' = E_2^\dagger$$

for some $\sigma_1^*, \sigma_1^\dagger$ and some $E_2^*, E_2^\dagger$ such that $(\kappa_2, p_2) \cdot (\kappa_2', p_2') \cdots \in |E_2^*|, |E_2^\dagger|$. So, we have both

$$\langle \sigma_1' \mid E_1 \rangle \rightsquigarrow \langle (\sigma_1'; \sigma_1^*) \mid (\kappa_1', p_1', \sigma_2') \rangle \rightsquigarrow \langle (\sigma_1'; \sigma_1^*; \sigma_2') \mid E_2^* \rangle$$

and

$$\langle \sigma_1' \mid E_2 \rangle \rightsquigarrow \left\langle (\sigma_1'; \sigma_1^\dagger) \mid (\kappa_1', p_1', \sigma_2') \right\rangle \rightsquigarrow \left\langle (\sigma_1'; \sigma_1^\dagger; \sigma_2') \mid E_2^\dagger \right\rangle$$

by go^+ transitions. Now, if κ_2 were guaranteed to be in Δ_2, we could go on to define

$$\sigma_2'\ \kappa_2\ p_2 = (\kappa_2', p_2', \sigma_3')$$

However, we only know (Prop. 4.2.43) that κ_2 is in the domain of Δ_1, Δ_2. Thus we set

$$(\sigma_1', \sigma_2')\ \kappa_2\ p_2 = (\kappa_2', p_2', \sigma_3')$$

with the intended meaning that if κ_2 is bound in Δ_1, we set $\sigma_1'\ \kappa_2\ p_2$, and if it is bound in Δ_2 we set $\sigma_2'\ \kappa_2\ p_2$. And so on, for each $1 \leq i \leq n$, we require that

$$(\sigma_1', \ldots, \sigma_i')\ \kappa_i\ p_i = (\kappa_i', p_i', \sigma_{i+1}')$$

Note that for this definition to make sense, it is crucial that **the pairs** (κ_i, p_i) **are distinct**, because otherwise we might give conflicting entries for the substitution maps. The affineness restriction is a sufficient condition for this, since it guarantees that the κ_i are distinct. We are also leaving many of the entries in the domain of the σ_i' unspecified, because they are not covered by any proper action (κ_i, p_i): we can set these entries to arbitrary expressions (in the appropriate context), for example to Ω or $\mho$.

Finally, after building $\sigma_1', \ldots, \sigma_n'$ to "walk through" the common chronicle $\mathfrak{c}_0$ in both E^* and $E^\dagger$, we arrive at the actions $\alpha^* \not\lesssim \alpha^\dagger$. These we can distinguish by setting

$$(\sigma_1', \ldots, \sigma_{n+1}')\ \kappa^*\ p^* = \mho$$

if $\alpha^* = (\kappa^*, p^*)$ is a proper action, and likewise

$$(\sigma_1', \ldots, \sigma_{n+1}')\ \kappa^\dagger\ p^\dagger = \Omega$$

if $\alpha^\dagger = (\kappa^\dagger, p^\dagger)$ is a distinct proper action. Note that because $\alpha^* \not\lesssim \alpha^\dagger$, if α^* is improper it must be $\mho$, and if $\alpha^\dagger$ is improper it must be Ω. By repeating the above argument, we can verify that

$$\langle \sigma_1' \mid E^* \rangle \rightsquigarrow^* \langle (\sigma_1'; \sigma_2^*; \sigma_2') \mid E_2^* \rangle \rightsquigarrow^* \left\langle (\sigma_1'; \sigma_1^*; \sigma_2'; \ldots; \sigma_n^*; \sigma_{n+1}') \mid E_{n+1}^* \right\rangle \Downarrow \mho$$
$$\left\langle \sigma_1' \mid E^\dagger \right\rangle \rightsquigarrow^* \left\langle (\sigma_1'; \sigma_1^\dagger; \sigma_2') \mid E_2^\dagger \right\rangle \rightsquigarrow^* \left\langle (\sigma_1'; \sigma_1^\dagger; \sigma_2'; \ldots; \sigma_n^\dagger; \sigma_{n+1}') \mid E_{n+1}^\dagger \right\rangle \Downarrow \Omega$$

where $\sigma_1^*, \ldots, \sigma_n^*, E_{n+1}^*$ and $\sigma_1^\dagger, \ldots, \sigma_n^\dagger, E_{n+1}^\dagger$ are some subterms of E^* and $E^\dagger$ respectively, and where $\alpha^* \in E_{n+1}^*$, $\alpha^\dagger \in E_{n+1}^\dagger$. □

Theorem 4.2.54 (Affine separation)**.** *For pure affine expressions, $E_1 = E_2$ iff $E_1 \cong_\mho E_2$.*

Proof. By Theorem 4.2.38 and Lemma 4.2.53. □

Note that Lemma 4.2.53 actually allows us to make this statement stronger: for two *arbitrary* (not necessarily pure) $\mathcal{L}^+_{\mho}$ expressions, definitional equality coincides with observational equivalence. But while this is true here, in general we won't ask for such a coincidence, because it requires making sure we have the right equational theory for effectful terms. Doing this for arbitrary effects lies outside the scope of this thesis. Instead, we have the more modest goal of verifying that definitional equality is the right equational theory for pure terms *in the presence* of arbitrary effects.

As we highlighted in the proof, the affine restriction is crucial for the result in $\mathcal{L}^+_{\mho}$. We can demonstrate this explicitly by exhibiting two syntactically distinct but observationally indistinguishable expressions.

Proposition 4.2.55. *There exist two pure, non-affine expressions E_1, E_2 such that $E_1 \neq E_2$ but $E_1 \cong_{\mho} E_2$.*

Proof. We define E_1 and E_2 in the one-variable context $\kappa : \bullet\neg 1$:

$$E_1 = \kappa\ (\_[(() \mapsto \Omega)])$$
$$E_2 = \kappa\ (\_[(() \mapsto E_1)])$$

To gloss these expressions in words, E_1 calls κ with a 1-continuation (treated as a $\neg 1$-value) that when invoked immediately diverges, while E_2 calls κ with a 1-continuation that when invoked executes E_1. The chronicle representation of E_1 is generated by a single maximal chronicle (picking an arbitrary name for the continuation supplied by the value $\_[(() \mapsto \Omega)]$):

$$(\kappa, \kappa') \cdot (\kappa', ()) \cdot \Omega$$

and that of E_2 by the following:

$$(\kappa, \kappa') \cdot (\kappa', ()) \cdot (\kappa, \kappa'') \cdot (\kappa'', ()) \cdot \Omega$$

Note that $E_1 \neq E_2$ (although $E_1 \leq E_2$). Also note that the two actions (κ, κ') and (κ, κ'') in $|E_2|$ are two occurrences of the *same* action, since the pattern components are compared ignoring variable names.

Now, consider any possible environment γ for $\kappa : \bullet\neg 1$. In $\mathcal{L}^+_{\mho}$ there are just three possibilities:

1. $\gamma\ \kappa\ \kappa' = \Omega$
2. $\gamma\ \kappa\ \kappa' = (\kappa', (), \cdot)$
3. $\gamma\ \kappa\ \kappa' = \mho$

In situations (1) and (3), it is obvious that $\langle \gamma \mid E_1 \rangle$ and $\langle \gamma \mid E_2 \rangle$ yield the same result (Ω and $\mho$, respectively). In case (2), we have that

$$\begin{array}{rcl}\langle \gamma \mid E_1 \rangle & \rightsquigarrow & \langle (\gamma; (() \mapsto \Omega/\kappa')) \mid (\kappa', (), \cdot) \rangle \\ & \rightsquigarrow & \langle (\gamma; (() \mapsto \Omega/\kappa'); \cdot) \mid \Omega \rangle \\ & \Downarrow & \Omega \end{array}$$

but also

$$\begin{array}{rcl} \langle \gamma \mid E_2 \rangle & \rightsquigarrow & \langle (\gamma; (() \mapsto E_1/\kappa')) \mid (\kappa', (), \cdot) \rangle \\ & \rightsquigarrow & \langle (\gamma; (() \mapsto E_1/\kappa'); \cdot) \mid E_1 \rangle \\ & \rightsquigarrow & \langle (\gamma; (() \mapsto E_1/\kappa'); \cdot; (() \mapsto \Omega/\kappa'')) \mid (\kappa'', (), \cdot) \rangle \\ & \rightsquigarrow & \langle (\gamma; (() \mapsto E_1/\kappa'); \cdot; (() \mapsto \Omega/\kappa''); \cdot) \mid \Omega \rangle \\ & \Downarrow & \Omega \end{array}$$

Hence $E_1 \cong_{\mho} E_2$. □

On the other hand, conceptually the argument in Lemma 4.2.53 did not really require very much. Any two distinct expressions $E^* \neq E^\dagger$ must contain respective chronicles $\mathfrak{c}_0 \cdot \alpha^* \neq \mathfrak{c}_0 \cdot \alpha^\dagger$, and so all we needed were:

1. Two distinct results to distinguish the last actions α^* and $\alpha^\dagger$
2. An environment that crawls through the common prefix $\mathfrak{c}_0$

For (1), we used the two distinct results available in $\mathcal{L}^+_{\mho}$: divergence and immediate failure. For (2), we remained within the total fragment of $\mathcal{L}^+$, but relied on the affineness restriction, which ensured that all proper positive actions (κ_i, p_i) and (κ_j, p_j) were distinct, and hence could be given unique, well-defined entries (respectively $(\kappa'_i, p'_i, \sigma'_{i+1})$ and $(\kappa'_j, p'_j, \sigma'_{j+1})$) in the environment.

So where does that leave us? To build the chronicle-crawling environment in the general case, the idea is that we just include an additional piece of state as a *counter*, updated during the course of evaluation. By querying this counter, the environment can return different expressions in response to distinct occurrences of the same positive action.

4.2.15 Ground state, and the separation theorem

To encode a counter, we will use a slightly more general effect: ground (integer) state. We extend $\mathcal{L}^+$ with a pair of operations for building expressions:

$$\frac{\mathbb{N} \longrightarrow \Gamma \vdash \#}{\Gamma \vdash \#}\ \mathsf{read} \qquad \frac{\mathbb{N} \qquad \Gamma \vdash \#}{\Gamma \vdash \#}\ \mathsf{write}$$

The operational intuition here is standard: $\mathsf{read}(E_i)_{i \in \mathbb{N}}$ reads the value i of an integer variable and then executes E_i, while $\mathsf{write}(j, E)$ writes the value j to the variable before executing E. To make this intuition precise, we first have to extend our notion of environment.

Definition 4.2.56 (Stateful environments)**.** *A* **stateful environment** *is an ordinary environment γ paired with an integer j, which we write $\mathsf{st}(j, \gamma)$. The* bind *operation is extended to stateful environments by the equation $\mathsf{bind}(\mathsf{st}(j, \gamma), \sigma) = \mathsf{st}(j, \mathsf{bind}(\gamma, \sigma))$, and the* lookup *operation by the equation $\mathsf{lookup}(\mathsf{st}(j, \gamma), \kappa) = \mathsf{lookup}(\gamma, \kappa)$.*

With this updated definition of environments and of the environment operations, we only need to add two rules to define our small-step semantics for state:

$$\begin{array}{rcll} \langle \mathsf{st}(i, \gamma) \mid \mathsf{read}(E_i)_{i \in \mathbb{N}} \rangle & \rightsquigarrow & \langle \mathsf{st}(i, \gamma) \mid E_i \rangle & (\mathsf{read}_i) \\ \langle \mathsf{st}(i, \gamma) \mid \mathsf{write}(j, E) \rangle & \rightsquigarrow & \langle \mathsf{st}(j, \gamma) \mid E \rangle & (\mathsf{write}_j) \end{array}$$

We call this extension $\mathcal{L}^+_{\mho\mathsf{r/w}}$, and write $\cong_{\mho\mathsf{r/w}}$ for the derived observational equivalence relation. Of course, we have to verify that $\mathcal{L}^+_{\mho\mathsf{r/w}}$ doesn't break the type safety and congruence properties of $\mathcal{L}^+_{\mho}$, but this is more or less immediate, because the new versions of bind and lookup preserve all the properties of the old versions, and the read_i and write_j rules are trivial. The reader may notice that we have not fixed a notion of definitional equality for read and write expressions. In this case, we could try to come up with a sensible definition. However, as I explained above, in general the question of axiomatizing equality for effectful terms lies outside the scope of this thesis. Because adding read and write to $\mathcal{L}^+_{\mho}$ creates no new results, we have already fully determined the observational equivalence relation $\cong_{\mho\mathsf{r/w}}$, and can thus already compare it to definitional equality on pure terms of $\mathcal{L}^+$.

Theorem 4.2.57 (Separation)**.** *Let E_1, E_2 be arbitrary pure expressions. Then $E_1 = E_2$ iff $E_1 \cong_{\mho\mathsf{r/w}} E_2$.*

Proof. In the forward direction, we apply Theorem 4.2.38. In the backward direction, to distinguish $E_1 \neq E_2$ with some stateful environment, we extend the construction of Lemma 4.2.53 to read the *index of the current positive action* as input. Recall, before we required that

$$(\sigma'_1, \dots, \sigma'_i)\ \kappa_i\ p_i = (\kappa'_i, p'_i, \sigma'_{i+1})$$

for all proper positive actions (κ_i, p_i), $1 \leq i \leq n$ in the common prefix $\mathfrak{c}_0$, as well as that

$$(\sigma'_1, \dots, \sigma'_{n+1})\ \kappa^*\ p^* = \mho$$

and

$$(\sigma'_1, \dots, \sigma'_{n+1})\ \kappa^\dagger\ p^\dagger = \Omega$$

if the distinct final actions were proper, $\alpha^* = (\kappa^*, p^*), \alpha^\dagger = (\kappa^\dagger, p^\dagger)$. We now update these conditions:

$$\begin{aligned}
(\sigma'_1, \dots, \sigma'_i)\ \kappa_i\ p_i \quad\quad\ \ i\ &= (i+1, \kappa'_i, p'_i, \sigma'_{i+1}) \\
(\sigma'_1, \dots, \sigma'_{n+1})\ \kappa^*\ p^*\ (n+1)\ &= \mho \\
(\sigma'_1, \dots, \sigma'_{n+1})\ \kappa^\dagger\ p^\dagger\ (n+1)\ &= \Omega
\end{aligned}$$

where a clause (j, κ, p, σ) stands for the expression $\mathsf{write}(j, \kappa\ (p[\sigma]))$, and a clause $\sigma\ \kappa\ p\ j = E$ has the intended reading that $\sigma(\kappa)(p) = \mathsf{read}(E_i)_{i\in\mathbb{N}}$ for some collection of expressions $(E_i)_{i\in\mathbb{N}}$ such that $E_j = E$ (and is otherwise arbitrary). The reader can verify that this is a legitimate definition of the σ'_i, irrespective of whether the expressions are affine, because the counter ensures there are no conflicting entries. We then execute E_1 and E_2 in the environment $\mathsf{st}(1, \sigma'_1)$. The same argument as in Lemma 4.2.53 shows that E_1 evaluates to $\mho$ and E_2 to Ω. □

Example 4.2.58. Let E_1 and E_2 be as in the proof of Proposition 4.2.55. Recall that E_1 contains maximal chronicle

$$(\kappa, \kappa') \cdot (\kappa', ()) \cdot \Omega$$

and E_2 contains

$$(\kappa, \kappa') \cdot (\kappa', ()) \cdot (\kappa, \kappa'') \cdot (\kappa'', ()) \cdot \Omega$$

Then we can distinguish E_1 and E_2 in the environment $\mathsf{st}(1, \sigma_1')$ and initial state 1, where

$$\begin{aligned} \sigma_1' \ \kappa \ \kappa' \ 1 &= (2, \kappa', (), \cdot) \\ \sigma_1' \ \kappa \ \kappa' \ 2 &= \mho \end{aligned}$$

We have that

$$\begin{aligned} \langle \mathsf{st}(1, \sigma_1') \mid E_1 \rangle \quad &\leadsto^* \quad \langle \mathsf{st}(1, \mathsf{bind}(\sigma_1', (() \mapsto \Omega/\kappa'))) \mid (2, \kappa', (), \cdot) \rangle \\ &\leadsto \quad \langle \mathsf{st}(2, \mathsf{bind}(\mathsf{bind}(\sigma_1', (() \mapsto \Omega/\kappa')), \cdot)) \mid \Omega \rangle \\ &\Downarrow \quad \Omega \end{aligned}$$

$$\begin{aligned} \langle \mathsf{st}(1, \sigma_1') \mid E_2 \rangle \quad &\leadsto^* \quad \langle \mathsf{st}(1, \mathsf{bind}(\sigma_1', (() \mapsto E_1/\kappa'))) \mid (2, \kappa', (), \cdot) \rangle \\ &\leadsto \quad \langle \mathsf{st}(2, \mathsf{bind}(\mathsf{bind}(\sigma_1', (() \mapsto E_1/\kappa')), \cdot)) \mid E_1 \rangle \\ &\leadsto \quad \langle \mathsf{st}(2, \mathsf{bind}(\mathsf{bind}(\mathsf{bind}(\sigma_1', (() \mapsto E_1/\kappa')), \cdot), (() \mapsto \Omega/\kappa''))) \mid \mho \rangle \\ &\Downarrow \quad \mho \end{aligned}$$

And so E_1 and E_2 are observationally distinct in $\mathcal{L}^+_{\mho\mathsf{r/w}}$. ∎

4.3 $\mathcal{L}$: A language with mixed evaluation order

Now that we have defined $\mathcal{L}^+$ and explored it at some length, we should have no problem generalizing to the language $\mathcal{L}$, which is the type-free notation for arbitrary canonical derivations mixing positive and negative types. For simplicity, we still omit atomic types, though the interested reader should have no trouble reconstructing them from the development in §2.3.2.

4.3.1 Continuation patterns and value frames

We begin by describing how to define negative types by their *continuation patterns.* Unannotated frames Δ can now contain negative **value holes** A^-, while annotated frames can contain **value variables** $x : A^-$.

Recall the rules from §2.2 for building refutation patterns:

$$\frac{}{A^- \Vdash \bullet \mathrel{\bar{=}} A}$$

$$\frac{}{\cdot \Vdash \bullet \bot} \qquad \frac{\Delta_1 \Vdash \bullet A \quad \Delta_2 \Vdash \bullet B}{\Delta_1, \Delta_2 \Vdash \bullet A \invamp B}$$

$$\frac{\Delta \Vdash \bullet A}{\Delta \Vdash \bullet A \& B} \qquad \frac{\Delta \Vdash \bullet B}{\Delta \Vdash \bullet A \& B} \quad \text{(no rule for } \top)$$

Now we assign labels to the rules:

$$\frac{}{A^- \Vdash \bullet \mathrel{\bar{=}} A}\ -$$

$$\frac{}{\cdot \Vdash \bullet \bot}\ [] \qquad \frac{\Delta_1 \Vdash \bullet A \quad \Delta_2 \Vdash \bullet B}{\Delta_1, \Delta_2 \Vdash \bullet A \invamp B}\ \mathsf{copair}$$

$$\frac{\Delta \Vdash \bullet A}{\Delta \Vdash \bullet A \& B}\ \mathsf{fst;} \qquad \frac{\Delta \Vdash \bullet B}{\Delta \Vdash \bullet A \& B}\ \mathsf{snd;}$$

These labels give us a type-free notation for (refutation/continuation) patterns. For example, copair(_, fst;_) and copair(_, snd;_) are two $\neg A \parr (\neg B \& \neg C)$-patterns with frames A, B and A, C, respectively.

Again, we can also rewrite the rule for $\neg$ with an explicit variable name:

$$\frac{}{x : A^{-} \Vdash \bullet \neg A}\ x$$

Then the above patterns could be written as copair(x_1, fst;x_2) and copair(x_1, snd;x_2).

We use the letter d to range over continuation patterns. Like we did with value pattern constructors, we associate unary continuation pattern constructors to the right, so that for example fst;snd;x is shorthand for fst;(snd;x).

Again, we don't attach importance to this particular collection of continuation patterns. As one interesting example of a negative datatype outside the propositional fragment, we can introduce the type of streams of A's (where A is negative), with the following continuation patterns:

$$\frac{\Delta \Vdash \bullet A}{\Delta \Vdash \bullet \mathcal{S}A}\ \mathsf{hd;} \qquad \frac{\Delta \Vdash \bullet \mathcal{S}A}{\Delta \Vdash \bullet \mathcal{S}A}\ \mathsf{tl;}$$

Continuation patterns do not have direct counterparts in mainstream functional languages, so it takes some time to get an intuition for them—but the idea is that they axiomatize the possible observations on values of negative type. To observe a stream we can either observe its head or observe its tail, for instance.

4.3.2 Terms

Given a collection of value and continuation patterns, we construct the language $\mathcal{L}$ by including all the term-forming rules of $\mathcal{L}^{+}$, and adding a few more to deal with negative types.

The rules for building negative values and continuations are precisely dual to the rules for building their positive counterparts. Accordingly, we can come up with some new slogans. Let's examine the rule for building negative values:

$$\frac{\overset{d}{\Delta \Vdash \bullet A^{-}} \quad \longrightarrow \quad \overset{E_d}{\Gamma, \Delta \vdash \#}}{\Gamma \vdash A^{-}}$$

The slogan behind this rule is that

a negative value is a map from continuations patterns to expressions

or in a slightly punchier version,

a negative value is a map from observations to actions

The operational intuition behind this slogan is that negative values are *computed on demand,* as the environment makes different observations. Consider, for example the type $\mathcal{S}\bot$. Since $\bot$ has only one observation [], observations on $\mathcal{S}\bot$ are of the form hd;[], tl;hd;[], tl;tl;hd;[], etc., which we can gloss as "get the first element and then stop", "get the second element and then stop", "get the third element and then stop", etc. We build a $\mathcal{S}\bot$-value V by specifying which expression to execute for each of these continuation patterns:

$$\begin{array}{rcl} V\ \mathsf{hd};[] & = & E_0 \\ V\ \mathsf{tl};\mathsf{hd};[] & = & E_1 \\ V\ \mathsf{tl};\mathsf{tl};\mathsf{hd};[] & = & E_2 \\ & \vdots & \end{array}$$

Incidentally, the reader may notice a resemblance between $\mathcal{S}\bot$-values and **N**-continuations. As we did with positive continuations, we allow negative values to be defined by partial recursive maps, reifying the possibility that $V(d)$ diverges as $V(d) = \Omega$.

We construct negative continuations as follows:

$$\frac{\overset{d}{\Delta \Vdash \bullet A^-} \quad \overset{\sigma}{\Gamma \vdash \Delta}}{\Gamma \vdash \bullet A^-}$$

writing the result as $d[\sigma]$. Thus we say that

a negative continuation is a continuation pattern under a substitution

Again, this is a somewhat unfamiliar interpretation of continuations: the "rest of the computation" for a negative value can always be put into a normal form that can be pattern-matched against.[3] For example, by this interpretation, there can never be any $\top$-continuations even in the presence of non-termination and effects—contrast this with the two different 1-continuations we can build using Ω and $\mho$. Again, it takes some time to build an intuition here. As we will see below, the really interesting negative continuations arise for types that mix negative with positive polarity.

Substitutions in $\mathcal{L}$ are constructed in the same way as in $\mathcal{L}^+$, except that they can contain negative values as well as positive continuations. We write $\sigma(x)$ for the action of a Δ-substitution on a value variable in Δ. Finally, a value variable can be used by pairing it with a negative continuation:

$$\frac{x : A^- \in \Gamma \quad \overset{K}{\Gamma \vdash \bullet A^-}}{\Gamma \vdash \#}$$

which we write as $x\ K$. Operationally, $x\ K$ is interpreted as passing the continuation K to the value denoted by x at runtime, a convention underlying control operators such as callcc, as well as the call-by-*name* CPS transform.

4.3.3 Mixed polarity types

The most interesting features of $\mathcal{L}$ arise from mixing positive and negative polarity. At the heart of this interaction are the shift connectives $\downarrow$ and $\uparrow$. The patterns for the shifts are trivial, and we give them in their variable-annotated versions:

[3] In the literature on call-by-value/call-by-name duality, these inductively constructed continuations are sometimes called "covalues" [Wadler, 2003].

$$\frac{}{x : A^- \Vdash \downarrow A}\ x \qquad \frac{}{\kappa : \bullet A^+ \Vdash \bullet \uparrow A}\ \kappa$$

Intuitively, the value pattern rule for $\downarrow$ says that we *cannot* decompose a negative value into more basic parts—we can only bind it to a variable. The rule for $\uparrow$ expresses the dual principle, that we cannot decompose a positive continuation.

For example, the negative type $\mathcal{S}(\uparrow\mathbf{B})$ can be thought of as representing infinite boolean sequences (sometimes called the Cantor space). Its continuation patterns are $\mathsf{hd};\kappa$, $\mathsf{tl};\mathsf{hd};\kappa$, $\mathsf{tl};\mathsf{tl};\mathsf{hd};\kappa$, etc., which all bind a single continuation variable $\kappa : \bullet\mathbf{B}$. Abstractly, κ stands for what to do with the nth boolean in the stream. Thus, we can represent an infinite, alternating sequence of 0s and 1s as a $\mathcal{S}(\uparrow\mathbf{B})$-value, by the following definition:

$$\begin{aligned} V\ \mathsf{hd};\kappa &= \kappa\ \mathsf{TT} \\ V\ \mathsf{tl};\mathsf{hd};\kappa &= \kappa\ \mathsf{FF} \\ V\ \mathsf{tl};\mathsf{tl};\mathsf{hd};\kappa &= \kappa\ \mathsf{TT} \\ V\ \mathsf{tl};\mathsf{tl};\mathsf{tl};\mathsf{hd};\kappa &= \kappa\ \mathsf{FF} \\ &\vdots \end{aligned}$$

Of course, in the presence of effects, this type contains more than just the Cantor space. For example, rather than passing a value to the continuation variable, V could diverge (Ω) or abort ($\mho$).

The positive type $\downarrow\uparrow\mathbf{N} \otimes \downarrow\uparrow\mathbf{N}$ can be thought of as representing a pair of *suspended* $\mathbf{N}$-computations. It has a single pattern, (x_1, x_2), which binds a pair of variables $x_1 : \uparrow\mathbf{N}$, $x_2 : \uparrow\mathbf{N}$. For instance,

$$(\_,\_)[(\kappa \mapsto \kappa\ \mathsf{Z}, \kappa \mapsto \Omega)]$$

is a positive value containing a pair of suspended computations: when invoked with a $\mathbf{N}$-continuation, the first computation always returns zero, the second diverges.

The shifts also let us define some interesting new datatypes. For example, we can define the type of *lazy lists* as the negative type $\mathcal{LL}'A = \uparrow\mathcal{LL}A$, where the positive type $\mathcal{LL}A$ is defined by the following value patterns:

$$\frac{}{\cdot \Vdash \mathcal{LL}A}\ \mathsf{nil} \qquad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \downarrow\mathcal{LL}'A}{\Delta_1, \Delta_2 \Vdash \mathcal{LL}A}\ \mathsf{cons}$$

Note that all $\mathcal{LL}A$ patterns have the form nil or $\mathsf{cons}(p, x)$, because the second premise of the cons rule can only be satisfied by the trivial pattern for $\downarrow$ (setting $\Delta_2 = x : \mathcal{LL}'A$). It is important to realize that lazy lists (ubiquitous in Haskell and encodable in ML) are *not* the same thing as the streams we defined above, having very different patterns. One dimension where they differ, for instance, is that values of type $\mathcal{LL}'A$ can represent lists of A's of finite length, whereas values of type $\mathcal{S}\uparrow A$ (note the polarity shift because A is positive) can only represent infinite lists. More generally, we can view Haskell-style *lazy sums* as represented by negative types of the form $\uparrow(\downarrow A \oplus \downarrow B)$.

In addition to the shifts, we give patterns for the connectives $\to$ and $-$:

$$\frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash \bullet A \to B}\ \mathsf{app} \qquad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash A - B}\ \mathsf{coapp}$$

The app rule expresses the following principle about the mode of use of a function type:

To use a value of type $A \to B$, we must provide an argument A, and use the result B

Continuation patterns for $A \to B$ thus take the form $\mathsf{app}(p, d)$, where p is an A-value pattern, and d is a B-continuation pattern. Because app patterns are common, we use the shorthand $p@d = \mathsf{app}(p, d)$. We treat @ as right associative, so that we can apply the Currying convention:

Observation 4.3.1. *A function of type $(A_1 \otimes \ldots \otimes A_n) \to B$ (with the A_i positive and B negative) is essentially the same as a function of type $A_1 \to \cdots \to A_n \to B$, the former having continuation patterns of shape $(p_1, \ldots, p_n)@d$, the latter of shape $p_1@\ldots@p_n@d$. (The two types are isomorphic, in the sense of Definition 4.3.6 below.)*

Example 4.3.2. Although in the traditional setting of λ-calculus we build functions using the $\lambda x.(-)$ construct, in practical functional languages like ML and Haskell we invariably use pattern-matching. It is worthwhile to see how definition of functions by pattern-matching emerges naturally in $\mathcal{L}$ from the definition of the function type by its continuation patterns.

Consider the continuation transformer and_κ from Example 4.2.9. We defined it is a positive continuation accepting type $\mathbf{B} \otimes \mathbf{B}$, assuming κ is a continuation variable accepting $\mathbf{B}$. But now we can alternatively define it as negative value of type $(\mathbf{B} \otimes \mathbf{B}) \to \uparrow\mathbf{B}$:

$$\begin{aligned}
\mathit{and}\ (\mathsf{tt}, \mathsf{tt})@\kappa &= \kappa\ \mathsf{TT} \\
\mathit{and}\ (\mathsf{tt}, \mathsf{ff})@\kappa &= \kappa\ \mathsf{FF} \\
\mathit{and}\ (\mathsf{ff}, \mathsf{tt})@\kappa &= \kappa\ \mathsf{FF} \\
\mathit{and}\ (\mathsf{ff}, \mathsf{ff})@\kappa &= \kappa\ \mathsf{FF}
\end{aligned}$$

or as a Curried function of type $\mathbf{B} \to \mathbf{B} \to \uparrow\mathbf{B}$:

$$\begin{aligned}
\mathit{and}\ \mathsf{tt}@\mathsf{tt}@\kappa &= \kappa\ \mathsf{TT} \\
\mathit{and}\ \mathsf{tt}@\mathsf{ff}@\kappa &= \kappa\ \mathsf{FF} \\
\mathit{and}\ \mathsf{ff}@\mathsf{tt}@\kappa &= \kappa\ \mathsf{FF} \\
\mathit{and}\ \mathsf{ff}@\mathsf{ff}@\kappa &= \kappa\ \mathsf{FF}
\end{aligned}$$

Modulo the shift, these types look more like the conventional types of *and*. And modulo the continuation-passing, these definitions look just like its traditional definition by pattern-matching. There is a conceptual shift, though, in that we are defining the function by *matching against its observations,* rather than on its arguments. This does not make much difference syntactically, however, precisely because the boolean arguments are part of the observation, and the observation on the boolean result must be treated abstractly as a continuation variable. ■

As we said before, we treat the collection of types in $\mathcal{L}$ as open-ended, and we could go on to consider many more interesting types by defining new patterns. Indeed, we *should* do this if we are using $\mathcal{L}$ as a real programming language, as opposed to just studying its high-level features. When we program in languages like ML and Haskell, we often define new datatypes to solve new problems, rather than trying to encode them with a fixed set of existing datatypes. Even if two types are structurally equivalent, it can be useful to maintain a conceptual distinction between them. For example, in a program manipulating red-black trees, we might introduce a type color containing the two patterns red and black. Although color is isomorphic to $\mathbf{B}$ and we can go back and forth between them, having these domain-specific tags is a helpful mnemonic device. If instead we always had to write $\mathtt{tt}$ and $\mathtt{ff}$ where we meant red and black, we would no doubt end up wasting too much time trying to remember which was which.

The virtue of our pattern-based formulation of $\mathcal{L}$ is that the precise collection of patterns *doesn't matter* for the high-level properties of the language. Thus there is no cost for defining new types when we program in $\mathcal{L}$. That said, in this thesis we *are* mainly interested in studying $\mathcal{L}$'s high-level features, and its relationship to traditional foundational languages like the λ-calculus. Indeed, it is worthwhile to take a brief look at the other end of the spectrum, where instead of building up a rich collection of types, we suffice with two.

4.3.4 Untyped, or "uni-typed"? Better: bi-typed!

An age-old source of rancor in programming languages is the status of the untyped λ-calculus relative to its typed cousins. On the one hand, as Church showed, untyped λ-calculus is a computationally universal language capable of representing arbitrary partial recursive functions, whereas simply-typed (or polymorphic) λ-calculus is not (since all programs terminate). On the other hand, after the addition of recursive types it becomes a simple exercise to encode arbitrary untyped programs by means of a single "universal type" $U = \mu X.(X \to X)$. Thus it is sometimes said that the untyped λ-calculus is really *uni-typed*, i.e., it is a special case of typed programming with only one type.[4] But on the third hand, untyped λ-calculus programs aren't literally *the same thing* as programs with the recursive type U. To translate an arbitrary untyped term into typed λ-calculus, at various places in the term we may have to insert coercions transforming subterms of type U to type $U \to U$, and vice versa. For example, to encode the untyped term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ we write

$$(\lambda x.(c\ x)\ x)\ d(\lambda x.(c\ x)\ x)$$

where c is the coercion from U to $U \to U$ and d is the reverse coercion. It could be argued that these coercions don't amount to much, or alternatively that making them explicit clarifies the original program's sense... but in any case they're *there* in the typed term, where they *aren't* in the untyped term, and aren't needed.

The point of these shallow remarks is just to motivate the observation that in $\mathcal{L}$, the status of untyped programs is significantly easier to understand, because the language definition is already generic with respect to types. Imagine, as a first step, building an untyped variant of $\mathcal{L}^+$. We begin by defining contexts that merely bind some continuation variables, without specifying their type:

$$\Delta ::= \kappa \mid \cdot \mid (\Delta_1, \Delta_2)$$

Then we define untyped value patterns as derivations of $\Delta \Vdash +$, which is defined just as we defined $\Delta \Vdash A^+$ but without any types, e.g., with rules like the following:

$$\frac{}{\kappa \Vdash +}\,\kappa \qquad \frac{}{\cdot \Vdash +}\,() \qquad \frac{\Delta_1 \Vdash + \quad \Delta_2 \Vdash +}{\Delta_1, \Delta_2 \Vdash +}\,\mathsf{pair} \qquad \frac{\Delta \Vdash +}{\Delta \Vdash +}\,\mathsf{inl} \quad \frac{\Delta \Vdash +}{\Delta \Vdash +}\,\mathsf{inr}$$

Finally, we use these untyped patterns to define the untyped terms of our language, basically as we did in the typed case. For example, if we forget types, the rule of refutation,

[4]See, e.g., [Harper, Chapter 21, "The Untyped λ-calculus"].

$$\frac{\overset{p}{\Delta \Vdash +} \quad \longrightarrow \quad \overset{E_p}{\Gamma, \Delta \vdash \#}}{\Gamma \vdash \bullet +}$$

builds a continuation $K = p \mapsto E_p$, given a map from untyped value patterns to expressions.[5]

Now, what we have to notice here is that really we aren't defining a *new* language—all we are doing is instantiating $\mathcal{L}^+$ with a different set of patterns, which coincidentally all belong to the same type. We might even call this type "$+$". Concretely, we do not need to define a new operational semantics for the untyped language—it is simply an instance of the typed semantics—and all the results of §4.2.5 (such as the Separation Theorem) continue to hold without need for new proof. Thus, different from the situation we saw above for λ-calculus, untyped $\mathcal{L}^+$ programs are *literally* typed $\mathcal{L}^+$ programs with the single type $+$. If we continue to play this game for all of $\mathcal{L}$, we see that "untyped $\mathcal{L}$" is really just the special case where all value patterns introduce the positive type $+$, and all continuation patterns the negative type $-$ (hence "bi-typed"), without any need for defining a new language.

What is going on here? Well in fairness to typed λ-calculus, we should point out that the reason additional coercions aren't necessary for type-checking untyped $\mathcal{L}^+/\mathcal{L}$ terms is because they are already there. For example, as noted in Propositions 4.2.7 and 4.2.8, in typed $\mathcal{L}^+$ there are explicit coercions transforming A-continuations into $\neg A$-values (replace K by $\_[K]$), and A-values into $\neg A$-continuations (replace V by $\kappa \mapsto \kappa\ V$). In untyped $\mathcal{L}^+$, these coercions don't have any action on types, but they must still be explicit, because values and continuations are different syntactic categories.

So have we really gained any footing in the debate over λ-calculus, typed vs. un-? In a sense, our pattern-based language definition argues for principles that should please *both* sides.[6] On the one hand, we are reiterating the position that coercions really are necessary for understanding the computational meaning of untyped λ-terms, and force this into the syntax by maintaining separate syntactic categories of positive/negative values and continuations. But on the other hand, we are also supporting the position that computation can be understood *before* the level of types, in the sense that it can be defined generically for all types of a given polarity.

4.3.5 $\mathcal{L}$ equality, semantics and effects: overview

In the following sections we complete the definition of $\mathcal{L}$ by defining its semantics, both equationally (definitional equality, the identity and composition principles), and operationally (the environment semantics). Since we already explored these concepts at length for $\mathcal{L}^+$, here we proceed rapidly, only describing the additional (negative polarity) cases needed to extend the definitions to $\mathcal{L}$, and confirming that the main properties of the equational and operational theories still hold.

4.3.6 Definitional equality

For the positive fragment, definitional equality of $\mathcal{L}$-terms is defined just as for $\mathcal{L}^+$-terms. We include the following additional rules for dealing with the new terms in the language:

[5]Because there are infinitely many untyped patterns and we may only want to consider a subset when defining a continuation, it becomes even more important to have the expression $\mho$, which we can use as a default entry.

[6]Or alternatively displease both, depending on the result of the glass half-full vs. half-empty debate.

$$\frac{d :: (\Delta \Vdash \bullet A^-) \quad \sigma_1 =_{\Gamma \vdash \Delta} \sigma_2}{d[\sigma_1] =_{\Gamma \vdash \bullet A^-} d[\sigma_2]} \qquad \frac{d :: (\Delta \Vdash \bullet A^-) \quad \longrightarrow \quad V_1(d) =_{\Gamma, \Delta \vdash \#} V_2(d)}{V_1 =_{\Gamma \vdash A^-} V_2}$$

$$\frac{x : A^- \in \Gamma \quad K_1 =_{\Gamma \vdash \bullet A^-} K_2}{x\ K_1 =_{\Gamma \vdash \#} x\ K_2}$$

We read these rules with conventions analogous to those for the positive fragment:

- Equality of continuations patterns is implicitly defined as equality of trees (ignoring names of variables), and equality of value variables is α-equivalence.
- The rule for equality of values implicitly requires that both value maps terminate on the same set of continuation patterns.
- We allow non-well-founded derivations of equality.

Proposition 4.3.3. *$\mathcal{L}$ equality is reflexive, symmetric, and transitive.*

4.3.7 Identity

For any negative value variable $x : A^- \in \Gamma$, we define the *identity A-value* $Id_x :: (\Gamma \vdash A^-)$ by the action $Id_x\ d = x\ (d[Id_{[d]}])$.

4.3.8 Composition

For any negative value $V :: (\Gamma \vdash A^-)$ and negative continuation $K :: (\Gamma \vdash \bullet A^-)$, we define the composite expression $V \bullet K :: (\Gamma \vdash \#)$ by

$$V \bullet d[\sigma] = V(d)[\sigma]$$

For any $\mathcal{L}$ term $t :: (\Gamma(\Delta) \vdash J)$ and Δ-substitution σ, we build the term $t[\sigma] :: (\Gamma \vdash J)$ by augmenting the definition from §4.2.8 with the following extra cases:

$$\begin{aligned} d[\sigma_0][\sigma] &= d[\sigma_0[\sigma]] \\ V[\sigma] &= d \mapsto V(d)[\sigma] \\ (x\ K)[\sigma] &= \begin{cases} V \bullet K[\sigma] & \text{if } \sigma(x) = V \\ x\ (K[\sigma]) & \text{if } x \notin \mathrm{dom}(\sigma) \end{cases} \end{aligned}$$

4.3.9 Properties of composition

Proposition 4.3.4 (Unit laws). *The three unit laws of Lemma 4.2.15 continue to hold in $\mathcal{L}$, and moreover:*

4. *$Id_x \bullet K = x\ K$*
   where $K :: (\Gamma \vdash \bullet A)$ and $x : A \in \Gamma$

Proposition 4.3.5 (Associativity). *The associativity equation $((t[\sigma_1])[\sigma_2] = t[(Id_{[\Delta_2]}, \sigma_1)[\sigma_2]]$, where $\sigma_1 :: (\Gamma \vdash \Delta_1)$ and $\sigma_2 :: (\Gamma, \Delta_1 \vdash \Delta_2)$ and $t :: (\Gamma, \Delta_1, \Delta_2 \vdash J)$) continues to hold in $\mathcal{L}$.*

Proof. These are both trivial generalizations of the arguments in §4.2.9. □

4.3.10 Type isomorphisms

We can extend the notion of type isomorphism to negative types:

Definition 4.3.6. *We say that two negative types A^- and B^- are isomorphic ($A \approx B$) if there exist a pair of value transformers $\phi :: (x : B \vdash A)$ and $\psi :: (x : A \vdash B)$ which are inverses, i.e.:*

1. $\phi[\psi/x] =_{x:A\vdash A} Id_x$
2. $\psi[\phi/x] =_{x:B\vdash \bullet B} Id_x$

Proposition 4.3.7. *The following isomorphisms of negative types hold:*

$$A \parr (B \parr C) \approx (A \parr B) \parr C \qquad A \parr B \approx B \parr A \qquad A \approx A \parr \bot$$

$$A \& (B \& C) \approx (A \& B) \& C \qquad A \& B \approx B \& A \qquad A \approx A \& \top$$

$$A \parr (B \& C) \approx (A \parr B) \& (A \parr C) \qquad A \parr \top \approx \top$$

$$\neg(A \& B) \approx \neg A \parr \neg B \qquad \neg \top \approx \bot$$

Proposition 4.3.8. *The following isomorphisms of mixed polarity types hold:*

$$A \to B \to C \approx (A \otimes B) \to C \qquad C^- \approx 1 \to C \qquad C^+ \approx C - \bot$$

$$A \to (B \& C) \approx (A \to B) \& (A \to C) \qquad (A \oplus B) \to C \approx (A \to C) \& (B \to C)$$

$$\downarrow A \otimes \downarrow B \approx \Downarrow(A \& B) \qquad \stackrel{+}{\neg} A \approx \Downarrow(A \to \bot) \qquad A - \uparrow B \approx A \otimes \stackrel{+}{\neg} B$$

$$\uparrow A \parr \uparrow B \approx \Uparrow(A \oplus B) \qquad \neg A \approx \Uparrow(1 - A) \qquad \downarrow A \to B \approx \neg A \parr B$$

$$A^+ \approx 1 - (A \to \bot) \qquad A^- \approx (1 - A) \to \bot$$

4.3.11 Environment semantics

Environments for $\mathcal{L}$ are defined just like $\mathcal{L}^+$-environments (§4.2.12), as lists of substitutions $\gamma = (\sigma_1; \dots \sigma_n)$, except that the substitutions σ_i now can contain mappings for value variables. Generalizing Proposition 4.2.26, given a Γ-environment γ and a value variable $x : A^- \in \Gamma$, we can lookup the variable in the environment to get a negative value $\mathsf{lookup}(\gamma, x) :: (\Gamma \vdash A^-)$. We then define the small-step environment semantics for arbitrary $\mathcal{L}$ programs by including two additional rules:

$$\begin{array}{rcll} \langle \gamma \mid x\ K \rangle & \rightsquigarrow & \langle \gamma \mid \mathsf{lookup}(\gamma, x) \mid K \rangle & (\text{lookup}^-) \\ \langle \gamma \mid V \mid d[\sigma] \rangle & \rightsquigarrow & \langle \mathsf{bind}(\gamma, \sigma) \mid V(d) \rangle & (\text{bind/call}^-) \end{array}$$

which can also be refactored as a single rule:

$$\langle \gamma \mid x\ (d[\sigma]) \rangle \quad \rightsquigarrow \quad \langle \mathsf{bind}(\gamma, \sigma) \mid \mathsf{lookup}(\gamma, x)(d) \rangle \qquad (\text{go}^-)$$

We adopt the same terminological conventions for $\mathcal{L}$ as we did for $\mathcal{L}^+$: we identify the *total* fragment by forbidding the use of Ω or $\mho$ and requiring the use of total recursive functions in the definition of negative values/positive continuations, and we identify the *pure* fragment by allowing Ω and partial recursive functions but forbidding effects like $\mho$.

Again, we can state strong versions of the usual progress and preservation lemmas.

Lemma 4.3.9 (Intrinsic progress)**.** *For all programs P in pure $\mathcal{L}$, there exists a P' such that $P \rightsquigarrow P'$.*

Lemma 4.3.10 (Intrinsic preservation)**.** *For pure $\mathcal{L}$, if $\langle \gamma \mid E \rangle \rightsquigarrow \langle \gamma' \mid E' \rangle$ then $E[\gamma] = E'[\gamma']$.*

Corollary 4.3.11 (Functionality)**.** *For pure $\mathcal{L}$, if $\langle \gamma \mid E \rangle \Downarrow R$ then $E[\gamma] = R$.*

The proofs of these properties are trivial generalizations of the proofs in §4.2.12.

4.3.12 Observational equivalence and the separation theorems

Observational equivalence is defined just as before: two $\mathcal{L}$ expressions $E_1, E_2 :: (\Gamma \vdash \#)$ are observationally equivalent ($E_1 \cong E_2$) if they yield the same result in all Γ-environments. The relationship between observational equivalence and definitional equality in $\mathcal{L}$ is essentially unchanged from the situation in $\mathcal{L}^+$. Again, it is easy to see that $\cong$ is a congruence, and somewhat less direct but still not too difficult to establish that given enough effects, any two syntactically distinct expressions may be observationally distinguished.

Theorem 4.3.12. *If $E_1 = E_2$ then $E_1 \cong E_2$.*

Theorem 4.3.13 (Affine separation)**.** *For pure affine expressions of $\mathcal{L}$, $E_1 = E_2$ iff $E_1 \cong_{\mho} E_2$.*

Theorem 4.3.14 (Separation)**.** *For arbitrary pure expressions of $\mathcal{L}$, $E_1 = E_2$ iff $E_1 \cong_{\mho\mathsf{r/w}} E_2$.*

By $\cong_{\mho}$ and $\cong_{\mho\mathsf{r/w}}$ we of course mean observational equivalence in the extensions of $\mathcal{L}$ with immediate failure/ground state. The proofs of these theorems are trivial generalizations of the analogous results for $\mathcal{L}^+$, though we must first extend the chronicle representation to arbitrary $\mathcal{L}$ expressions and substitutions by including the following additional (straightforward) definitions:

$$\frac{\mathfrak{c} \in |\sigma|}{(x, d) \cdot \mathfrak{c} \in |(x, d, \sigma)|} \qquad \frac{\mathfrak{c} \in |\sigma\ x'\ d'|}{(x', d') \cdot \mathfrak{c} \in |\sigma|}$$

4.4 Polarization and CPS translations

Near the end of Chapter 3, we drew a diagram explaining how to derive different double-negation translations of classical logic via polarization and focusing:

$$\begin{tikzcd}
 & \text{polarized logic} \arrow[dr, bend left, "(-)^m"] & \\
\text{classical logic} \arrow[ur, bend left, "(-)^*"] & & \text{minimal logic} \arrow[ll, bend left]
\end{tikzcd}$$

Given a proof of a classical proposition b, we can polarize b any way we like as a PPL proposition $A = b^*$ (with $|A| = b$), and via the completeness of focusing obtain a focusing proof of A. This focusing proof can also be interpreted as a canonical derivation (depending on the polarity of A, either asserting $\cdot \vdash A^-$, or $\kappa : \bullet A^+ \vdash \#$), which can then be translated directly into a proof of an unpolarized proposition in (the conjunction-disjunction-negation fragment of) minimal logic. The composition of these two steps yields different double-negation translations of classical into minimal logic, depending on how we choose to polarize the original classical theorem.

In this section, we reconsider the same picture from a computational standpoint:

$$\begin{tikzcd} & \mathcal{L} \arrow[dr, bend left, "(-)^m"] & \\ \lambda\text{-calculus} \arrow[ur, bend left, "(-)^*"] & & \text{CPS } \lambda\text{-calculus} \arrow[ll] \end{tikzcd}$$

How to read this diagram? Suppose we have a simply-typed term $M : \tau$ of λ-calculus with sums and products. We can polarize τ any way we like, e.g., interpreting products as $\otimes$ or $\&$, inserting shifts $\downarrow$ and $\uparrow$ in random places, etc. The focusing completeness theorem says that we can obtain a focusing proof of the polarized type A, which we can then interpret as a term of $\mathcal{L}$ (depending on the polarity of A, either a closed negative value or an expression with one free positive continuation variable). Finally, this $\mathcal{L}$-term can be reinterpreted as a simply-typed λ-calculus term in continuation-passing style, resulting in a CPS translation of the original λ-term. Again, we can then see different CPS translations as arising from different polarizations—although the translation isn't *entirely* determined by polarization, because (as already noted in Chapter 3) there is a bit of ambiguity in the computational content of focalization, particularly whether to use left-to-right or right-to-left evaluation of products.[7]

4.4.1 From λ-calculus to $\mathcal{L}^+$ and back

Rather than jumping in with full generality, we will begin with a simple example where the polarization is purely positive, and where the result of chasing the diagram is a well-known call-by-value CPS transformation. A similar result was presented by Führmann and Thielecke [2004], who showed how to derive this particular CPS transform as a composition of two more primitive translations (motivated by the categorical semantics of call-by-value, though, rather than by proof theory). The reader might revisit Theorem 3.5.5 to see the proof-theoretic statement closely corresponding to this purely positive polarization.

We take as our starting point the λ-calculus with sums and products (without atoms, but those could be easily added if we included them in $\mathcal{L}^+$). Types are given by the following grammar:

$$\tau ::= \tau_1 \supset \tau_2 \mid \tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2 \mid \mathrm{T} \mid \mathrm{F}$$

and terms by the following, with the standard typing rules:

$$\begin{aligned} M ::= \;& \lambda x.M \mid M_1\ M_2 \\ & \mid (M_1, M_2) \mid \pi_1\ M \mid \pi_2\ M \\ & \mid \iota_1\ M \mid \iota_2\ M \mid \mathrm{case}(M, y.M_1, z.M_2) \\ & \mid () \mid \mathrm{abort}(M) \end{aligned}$$

As our polarization $(-)^*$, we interpret all the connectives positively:

[7] One aspect of this analogy that may need a bit of clarification is why the left corner of the logical diagram contains classical logic (and in particular the focusing completeness theorem applied to classical sequent calculus) while the left corner of the computational diagram contains λ-calculus (which is supposed to correspond to natural deduction for intuitionistic logic). To be clear, we can apply the translation $\lambda \to \mathcal{L}$ generally for λ-calculus *with effects.* The historical genesis of the CPS translations, after all, was to account for the behavior of different λ-calculus evaluation strategies in the presence of non-termination and side-effects. In particular, we could extend the translation $\lambda \to \mathcal{L}$ to λ-calculus with the control operator callcc, corresponding to Peirce's Law and classical natural deduction. As for our choice to prove the completeness theorem for sequent calculus rather than natural deduction: that was only to relate to the historical view of focusing as a sequent calculus search procedure. The computational content of the two versions of the theorem (SC vs. ND) are not identical, but very similar.

$$(\tau_1 \supset \tau_2)^* = {}^{\pm}(\tau_1^* \otimes {}^{\pm}\tau_2^*) \quad (\tau_1 \wedge \tau_2)^* = \tau_1^* \otimes \tau_2^* \quad (\tau_1 \vee \tau_2)^* = \tau_1^* \oplus \tau_2^* \quad \mathrm{T}^* = 1 \quad \mathrm{F}^* = 0$$

It is trivial to verify that this is indeed a polarization (recalling Definition 3.4.1), in particular because $\tau_1 \supset \tau_2$ and $\sim(\tau_2 \wedge \sim\tau_2)$ are classically equivalent. Now, the first result we establish is really an instance of the focusing completeness theorem, and describes how to translate any λ-calculus term into a corresponding term of $\mathcal{L}^+$:

Theorem 4.4.1. *For any λ-term $\Gamma \vdash M : \tau$, there is a corresponding total $\mathcal{L}^+$-expression $E = M^*\ \kappa$, where $E :: (\Gamma^*, \kappa : \bullet\tau^* \vdash \#)$.*

Proof. To make the translation more readable, we use the notation "E where $\kappa = K$" as sugar for the composition $E[(K/\kappa)]$. On variables and function abstraction/application, the translation is defined as follows:

$$\begin{aligned} x^*\ \kappa &= \kappa\ \mathit{Id}_x \\ (\lambda x.M)^*\ \kappa &= \kappa\ ((x, \kappa') \mapsto M^*\ \kappa') \\ (M_1\ M_2)^*\ \kappa &= M_1^*\ \kappa_1 \text{ where } \kappa_1 = (\kappa' \mapsto \\ &\qquad M_2^*\ \kappa_2 \text{ where } \kappa_2 = (x \ \mapsto \\ &\qquad\quad \kappa'\ \mathsf{PAIR}(\mathit{Id}_x, \mathit{Id}_\kappa))) \end{aligned}$$

(Note that in the $\lambda x.M$ case, we are leaving implicit the coercion treating the $\tau_1^* \otimes \neg\tau_2^*$-continuation as a $\neg(\tau_1^* \otimes \neg\tau_2^*)$-value, and in the $M_1\ M_2$ case, we are making use of the combinator PAIR defined in Proposition 4.2.5.) On pairing and projection the translation is as follows:

$$\begin{aligned} (M_1, M_2)^*\ \kappa &= M_1^*\ \kappa_1 \text{ where } \kappa_1 = (x \mapsto \\ &\qquad M_2^*\ \kappa_2 \text{ where } \kappa_2 = (y \ \mapsto \\ &\qquad\quad \kappa\ \mathsf{PAIR}(\mathit{Id}_x, \mathit{Id}_y))) \\ (\pi_1\ M)^*\ \kappa &= M^*\ \kappa' \text{ where } \kappa' = ((x, y) \mapsto \kappa\ \mathit{Id}_x) \\ (\pi_2\ M)^*\ \kappa &= M^*\ \kappa' \text{ where } \kappa' = ((x, y) \mapsto \kappa\ \mathit{Id}_y) \end{aligned}$$

On injection/case-analysis as follows:

$$\begin{aligned} (\iota_1\ M)^*\ \kappa &= M^*\ \kappa' \text{ where } \kappa' = (y \mapsto \kappa\ (\mathsf{INL}\,\mathit{Id}_y)) \\ (\iota_2\ M)^*\ \kappa &= M^*\ \kappa' \text{ where } \kappa' = (z \mapsto \kappa\ (\mathsf{INR}\,\mathit{Id}_z)) \\ (\mathrm{case}(M, y.M_1, z.M_2))^*\ \kappa &= M^*\ \kappa' \text{ where } \kappa' = (\ \mathsf{inl}\,y \mapsto M_1^*\ \kappa \\ &\qquad\qquad\qquad\qquad\quad \mid \mathsf{inr}\,z \mapsto M_2^*\ \kappa) \end{aligned}$$

And finally on unit/abort as follows:

$$\begin{aligned} ()^*\ \kappa &= \kappa\ () \\ (\mathrm{abort}(M))^*\ \kappa &= M^*\ \kappa' \text{ where } \kappa' = \mathit{abort} \end{aligned}$$

where abort is the vacuous 0-continuation, with no pattern branches.

A couple observations about this translation:

- We have to verify that the translation does what it says it does, i.e., that the $\mathcal{L}^+$ expressions have the appropriate type. But this is an easy mechanical exercise—and really the translation was almost entirely determined by types in the first place.

Types $\tau ::= \sim\tau \mid \tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2 \mid \mathrm{T} \mid \mathrm{F}$

Terms:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash M : \#}{\Gamma \vdash \lambda x.M : \sim\tau} \qquad \frac{\Gamma \vdash M_1 : \sim\tau \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1\ M_2 : \#}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \wedge \tau_2} \qquad \frac{\Gamma \vdash M : \tau_1 \wedge \tau_2}{\Gamma \vdash \pi_1\ M : \tau_1} \qquad \frac{\Gamma \vdash M : \tau_1 \wedge \tau_2}{\Gamma \vdash \pi_2\ M : \tau_2}$$

$$\frac{\Gamma \vdash M : \tau_1}{\Gamma \vdash \iota_1\ M : \tau_1 \vee \tau_2} \qquad \frac{\Gamma \vdash M : \tau_2}{\Gamma \vdash \iota_2\ M : \tau_1 \vee \tau_2} \qquad \frac{\Gamma \vdash M : \tau_1 \vee \tau_2 \quad \Gamma, y : \tau_1 \vdash M_1 : \tau' \quad \Gamma, z : \tau_2 \vdash M_2 : \tau'}{\Gamma \vdash \mathrm{case}(M, y.M_1, z.M_2) : \tau'}$$

$$\frac{}{\Gamma \vdash () : \mathrm{T}} \qquad \frac{\Gamma \vdash M : \mathrm{F}}{\Gamma \vdash \mathrm{abort}(M) : \tau'}$$

Figure 4.1: CPS-restricted λ-calculus

- The choice of left-to-right evaluation order in the translations of application and pairing is *not* determined by types, only by convention.

□

Our next step will be to give a general translation from $\mathcal{L}^+$ terms to the fragment of λ-calculus in continuation-passing style. The λ-calculus typing rules specialized to this fragment are displayed in Figure 4.1, and are a direct annotation of natural deduction with conjunction, disjunction, and minimal negation (Figure 3.7).

Let $(-)^m$ be the translation from polarized logic into minimal logic defined in §3.5. We recall its action on positive propositions, and on assertions or refutations:

$$({\div} A)^m = \sim A^m \qquad (A \oplus B)^m = A^m \vee B^m \qquad (A \otimes B)^m = A^m \wedge B^m \qquad 1^m = \mathrm{T} \qquad 0^m = \mathrm{F}$$

$$(A^+)^m = A^m \qquad (\bullet A^+)^m = \sim A^m$$

The following statement simply gives the computational content of Theorem 3.5.1, specialized to the positive fragment (generalizing it back to cover all of total $\mathcal{L}$ is a straightforward exercise).

Theorem 4.4.2. *For any total $\mathcal{L}^+$ term $t :: (\Gamma \vdash J)$ there is a corresponding CPS term $M = t^m$, where $\Gamma^m \vdash M : J^m$.*

Proof. To define the translation, we need two auxiliary translations:

1. For any pattern $p :: (\Delta \Vdash A)$, there exists a CPS term $M = p^m$, where $\Delta^m \vdash M : A^m$.

2. Suppose that for every pattern $p :: (\Delta \Vdash A)$, there is a CPS term M_p, where $\Gamma^m, \Delta^m \vdash M_p : J^m$. Then there is a CPS term $\lambda x.M = \Lambda(p \mapsto M_p)$, where $\Gamma^m, x : A^m \vdash M : J^m$.

The translations (1) and (2) are defined by a straightforward induction on types as follows:

1.

Type	Translation of patterns (p^m)
$\overset{+}{\neg} A$	$\kappa^m = \kappa$
$A \otimes B$	$(p_1, p_2)^m = (p_1^m, p_2^m)$
$A \oplus B$	$(\mathsf{inl}\, p)^m = \iota_1\ (p^m)$ and $(\mathsf{inr}\, p)^m = \iota_2\ (p^m)$
1	$()^m = ()$
0	no patterns

2.

Type	Translation of pattern-indexed terms $(\Lambda(p \mapsto M_p))$
$\overset{+}{\neg} A$	$\Lambda(\kappa \mapsto M_\kappa) = \lambda\kappa.M_\kappa$
$A \otimes B$	$\Lambda((p_1, p_2) \mapsto M_{(p_1,p_2)}) = \lambda(y, z).M'$
	where $\lambda y.\lambda z.M' = \Lambda(p_1 \mapsto \Lambda(p_2 \mapsto M_{(p_1,p_2)}))$
$A \oplus B$	$\Lambda(\mathsf{inl}\, p_1 \mapsto M_{\mathsf{inl}\, p_1} \mid \mathsf{inr}\, p_2 \mapsto M_{\mathsf{inr}\, p_2}) = \lambda x.\mathrm{case}(x, y.M_1, z.M_2)$
	where $\lambda y.M_1 = \Lambda(p_1 \mapsto M_{\mathsf{inl}\, p_1})$, $\lambda z.M_2 = \Lambda(p_2 \mapsto M_{\mathsf{inr}\, p_2})$
1	$\Lambda(() \mapsto M_{()}) = \lambda x.M_{()}$
0	$\Lambda\emptyset = \lambda x.\mathrm{abort}(x)$

Note that in the $\otimes$-clause of translation (2), $\lambda(y, z).M'$ is syntactic sugar for $\lambda x.M'[\pi_1\ x/y][\pi_2\ x/z]$.

The translation of arbitrary $\mathcal{L}^+$ terms then proceeds as follows:

$$(p[\sigma])^m = p^m[\sigma^m] \qquad K^m = \Lambda(p \mapsto K(p)^m)$$

$$(\cdot)^m = () \qquad (K/\kappa)^m = K^m \qquad (\sigma_1, \sigma_2)^m = (\sigma_1^m, \sigma_2^m) \qquad (\kappa\ V)^m = \kappa\ (V^m)$$

where $p^m[\sigma^m]$ is a bit of syntactic high fructose corn syrup, standing for the result of substituting the components of the tuple σ^m for the appropriate variables in p^m. (We could avoid this notational malnutrition if natural deduction were better fortified with first-class substitutions.) □

We have a translation from λ-calculus into $\mathcal{L}^+$, and we have a translation from $\mathcal{L}^+$ back into CPS-restricted λ-calculus (or "CPS calculus" for short). So what happens when we compose them? In Figure 4.2, we show a well-known CPS translation, sending a λ-calculus term $M : \tau$ to another term $M^v : {\sim}{\sim}\tau^v$, where the translation on types is:

$$(A \to B)^v = {\sim}(A^v \wedge {\sim} B^v) \qquad (A \wedge B)^v = A^v \wedge B^v \quad (A \vee B)^v = A^v \vee B^v \qquad \mathrm{T}^v = \mathrm{T} \quad \mathrm{F}^v = \mathrm{F}$$

M^v is essentially the CPS transformation described by Reynolds [1972], Fischer [1972], and Plotkin [1975], generalized to arbitrary terms of λ-calculus with sums and products. And indeed, we can derive it as the composition $(-)^m \circ (-)^*$.

Theorem 4.4.3. *If* $\Gamma \vdash M : \tau$ *then* $\Gamma^v \vdash M^v : {\sim}{\sim}\tau^v$.

Proof. Immediate consequence of Theorems 4.4.1 and 4.4.2, after mechanical verification that $\Gamma^v = (\Gamma^*)^m$ and $M^v = \lambda\kappa.(M^*\ \kappa)^m$. □

4.4.2 Reconstructing call-by-value and call-by-name

It is natural to consider the purely negative polarization dual to the one in §4.4.1 (and close to the computational content of Theorem 3.5.6):

$$(\tau_1 \supset \tau_2)^* = \overset{-}{\neg}\tau_1^* \parr \tau_2^* \qquad (\tau_1 \wedge \tau_2)^* = \tau_1^* \& \tau_2^* \qquad (\tau_1 \vee \tau_2)^* = \tau_1^* \parr \tau_2^* \qquad \mathrm{T}^* = \top \quad \mathrm{F}^* = \bot$$

Action on types:

$$(A \to B)^v = {\sim}(A^v \wedge {\sim} B^v) \qquad (A \wedge B)^v = A^v \wedge B^v \quad (A \vee B)^v = A^v \vee B^v \qquad \mathrm{T}^v = \mathrm{T} \quad \mathrm{F}^v = \mathrm{F}$$

Action on terms ($M : \tau \Longrightarrow M^v : {\sim}{\sim}\tau^v$):

$$\begin{aligned}
x^v &= \lambda\kappa.\kappa\ x \\
(\lambda x.M)^v &= \lambda\kappa.\kappa\ (\lambda(x,\kappa').M^v\ \kappa') \\
(M_1\ M_2)^v &= \lambda\kappa.M_1^v\ (\lambda f.M_2^v\ (\lambda x.f\ (x,k))) \\
(M_1, M_2)^v &= \lambda\kappa.M_1^v\ (\lambda y.M_2^v\ (\lambda z.k\ (y,z))) \\
(\pi_1\ M)^v &= \lambda\kappa.M^v\ (\lambda x.k\ (\pi_1\ x)) \\
(\pi_2\ M)^v &= \lambda\kappa.M^v\ (\lambda x.k\ (\pi_2\ x)) \\
(\iota_1\ M)^v &= \lambda\kappa.M^v\ (\lambda x.k\ (\iota_1\ x)) \\
(\iota_2\ M)^v &= \lambda\kappa.M^v\ (\lambda x.k\ (\iota_2\ x)) \\
(\mathrm{case}(M, y.M_1, z.M_2))^v &= \lambda\kappa.M^v\ (\lambda x.\mathrm{case}(x, y.M_1^v\ \kappa, z.M_2^v\ \kappa)) \\
()^v &= \lambda\kappa.\kappa\ () \\
(\mathrm{abort}(M))^v &= \lambda\kappa.M^v\ (\lambda x.\mathrm{abort}(x))
\end{aligned}$$

Figure 4.2: A CPS translation of call-by-value functions, strict products and sums

This is natural, albeit somewhat artificial. Such a global dualization dramatically changes the computational interpretations of *all* of the λ-calculus type constructors, and not all of these interpretations occur in practice. This interpretation of λ-calculus sums in particular is very strange (e.g., because of type isomorphisms like $A \parr \top \approx \top$), and although such interpretations do appear sometimes in the literature on call-by-value/call-by-name duality, I would claim only as artifacts of misdirected attempts to treat evaluation order as a global, uniform policy decision independent of the type structure of a language, rather than as a local policy determined by types.[8] So instead of reconsidering the entire type structure of the λ-calculus, let us narrow our attention to the *call* in call-by-value/call-by-name, and consider specifically the different possible polarizations of the function space, and how they come about.

As a logical connective in §2.3.1, and as a type constructor in §4.3.3, we introduced a very natural negative polarity implication/function space. The type $A \to B$, with A positive and B negative, is defined by its continuation patterns $p@d$, where p is an A-value pattern, and d a B-continuation pattern. Let us call this negative connective $- \to -$ the *primordial function space.* Given the properties of the primordial function space, we can understand the choice to use call-by-value or call-by-name semantics uniformly for all functions in a program as corresponding to

[8]In one of the seminal works on computational duality, for example, Filinski [1989] introduced the *symmetric lambda calculus*, and defined what he called its "pure call-by-name semantics", corresponding very closely to this negative polarization. He made the observation that "the pure CBN coproduct is quite different from the one found in typical lazy programming languages" [*ibid.*, §2.5.1], but argued that this was not a defect, because the typical notion could be encoded by delaying the two branches of the sum. Essentially, in our terminology, Filinski was arguing that lazy sums could be encoded by the type $\neg\neg A \parr \neg\neg B$, which is indeed isomorphic (by Prop. 4.3.8) to the encoding $\Uparrow(\downarrow A \oplus \downarrow B)$ mentioned in §4.3.3. As I argued above, though, the fact that two types are isomorphic does not mean it is particularly natural to replace one by the other.

a choice to give all types the same polarity, positive or negative. Because the primordial function space is fundamentally mixed polarity, to get the polarities to work out for call-by-value/call-by-name function spaces we have to insert shifts in different places. Suppose, for instance, that we decide to interpret all types positively, as we did in §4.4.1. How do we make use of the primordial function space? The primordial function space will gladly *accept* positive types, but it *returns* negative types. So if we want to return a positive type B^+, it must be shifted, i.e., we must use $A \to \uparrow B$. But $A \to \uparrow B$ is itself negative, so we must also shift the entire formula (or else we could not have higher-order functions). In this way, we derive the decomposition of the call-by-value function space $A \xrightarrow{v} B = \downarrow(A \to \uparrow B)$, which is indeed isomorphic to the polarization $\overset{+}{\neg}(A \otimes \overset{+}{\neg} B)$ we discussed in §4.4.1. Conversely, if we choose to interpret all types negatively, then the return type of the primordial function space has the right polarity, as well as the function space itself, but we are forced to shift the argument type. In this way, we derive the decomposition of the call-by-name function space $A \xrightarrow{n} B = \downarrow A \to B$, which is isomorphic to the above polarization $\overset{-}{\neg} A \parr B$.

Thus, the choice to use uniformly positive or negative polarity results in two different *minimal* annotations of the primordial function space:

$$A \xrightarrow{v} B = \downarrow(A \to \uparrow B) \qquad\qquad A \xrightarrow{n} B = \downarrow A \to B$$

In both cases, we could of course insert extra shifts without breaking the polarity invariants—e.g., we could replace $\downarrow A \to B$ by $\uparrow\downarrow(\downarrow A \to B)$—but these are the fewest we can get away with.

For both of these polarizations, the embedding $(-)^*$ of λ-calculus into $\mathcal{L}$ is almost entirely determined by types, which then determines a unique CPS translation by the embedding $(-)^m$ from $\mathcal{L}$ into CPS calculus. For the call-by-value polarization, we define $(-)^*$ as follows on variables, function abstraction, and application:[9]

$$\boxed{(\tau_1 \supset \tau_2)^* = \downarrow(\tau_1^* \to \uparrow\tau_2^*)}$$

$$\begin{aligned}
x^*\ \kappa &= \kappa\ \mathit{Id}_x \\
(\lambda x.M)^*\ \kappa &= \kappa\ (x@\kappa' \mapsto M^*\ \kappa') \\
(M_1\ M_2)^*\ \kappa &= M_1^*\ \kappa_1 \text{ where } \kappa_1 = (y \mapsto \\
&\qquad M_2^*\ \kappa_2 \text{ where } \kappa_2 = (x \ \mapsto \\
&\qquad\quad y\ (\mathit{Id}_x@\mathit{Id}_\kappa)))
\end{aligned}$$

Note this is almost identical to the definition of $(-)^*$ we gave in the previous section for the isomorphic polarization of the function space $\overset{+}{\neg}(A \otimes \overset{+}{\neg} B)$, with only a few trivial syntactic differences. Indeed, after the embedding $(-)^m$, it yields the same CPS translation, Reynolds' original call-by-value one:

$$\begin{aligned}
x^v &= \lambda\kappa.\kappa\ x \\
(\lambda x.M)^v &= \lambda\kappa.\kappa\ (\lambda(x,\kappa').M^v\ \kappa') \\
(M_1\ M_2)^v &= \lambda\kappa.M_1^v\ (\lambda f.M_2^v\ (\lambda x.f\ (x,k)))
\end{aligned}$$

[9] Again, here we cannot yet define it for products and sums, because we have not fixed their polarization, only their polarity.

Plotkin [1975]: polarization $(\tau_1 \supset \tau_2)^* = \uparrow\downarrow(\downarrow\tau_1^* \to \tau_2^*)$

$$\begin{aligned} x^P &= x \\ (\lambda x.M)^P &= \lambda\kappa.\kappa\ (\lambda x.M^P) \\ (M_1\ M_2)^P &= \lambda\kappa.M_1^P\ (\lambda f.f\ M_2^P\ \kappa) \end{aligned}$$

Streicher and Reus [1998]: polarization $(\tau_1 \supset \tau_2)^* = \downarrow\tau_1^* \to \tau_2^*$

$$\begin{aligned} x^S &= x \\ (\lambda x.M)^S &= \lambda(x,\kappa).M^S\ \kappa \\ (M_1\ M_2)^S &= \lambda\kappa.M_1^S\ (M_2^S,\kappa) \end{aligned}$$

Figure 4.3: The Plotkin and Streicher call-by-name CPS transformations

Now consider the call-by-name polarization. Because τ^* is negative, we can translate $M : \tau$ directly to a negative τ^*-value M^*, without having to abstract in a continuation variable (cf. Corollary 3.4.11 of the focusing completeness theorem):[10]

$$\boxed{(\tau_1 \supset \tau_2)^* = \downarrow\tau_1^* \to \tau_2^*}$$

$$\begin{aligned} x^* &= \mathit{Id}_x \\ (\lambda x.M)^* &= x@d \mapsto M^*(d) \\ (M_1\ M_2)^* &= \kappa \mapsto M_1^* \bullet (M_2^*@\mathit{Id}_\kappa) \end{aligned}$$

Under the embedding $(-)^m$, this directly yields the following CPS translation:

$$\begin{aligned} x^S &= x \\ (\lambda x.M)^S &= \lambda(x,\kappa).M^S\ \kappa \\ (M_1\ M_2)^S &= \lambda\kappa.M_1^S\ (M_2^S,\kappa) \end{aligned}$$

Note, this is *not* Plotkin's original call-by-name translation: this is what is sometimes called the Streicher translation (introduced by Lafont, Reus, and Streicher [1993], and later studied in depth by Streicher and Reus [1998] and Hofmann and Streicher [1997]). Recall the Plotkin translation:

$$\begin{aligned} x^P &= x \\ (\lambda x.M)^P &= \lambda\kappa.\kappa\ (\lambda x.M^P) \\ (M_1\ M_2)^P &= \lambda\kappa.M_1^P\ (\lambda f.f\ M_2^P\ \kappa) \end{aligned}$$

As it happens, Plotkin's translation really does correspond to the "inefficient" negative polarization $\uparrow\downarrow(\downarrow A \to B)$. The reader may try reconstructing this, before looking at the next three

[10]And here the definition of $(-)^*$ is entirely forced by types, so the reader can try reconstructing it before reading on.

lines:

$$\boxed{(\tau_1 \supset \tau_2)^* = \uparrow\!\downarrow\!(\downarrow\tau_1^* \to \tau_2^*)}$$

$$\begin{aligned} x^* &= \mathit{Id}_x \\ (\lambda x.M)^* &= \kappa \mapsto \kappa\ (x@d \mapsto M^*(d)) \\ (M_1\ M_2)^* &= \kappa \mapsto M_1^* \bullet (y \mapsto y\ (M_2^*@\mathit{Id}_\kappa)) \end{aligned}$$

The syntactic correspondence between this polarization and the Plotkin translation is almost completely apparent—it becomes completely transparent if we fill in the missing η-expansion in the abstraction clause of the Plotkin translation:

$$(\lambda x.M)^P = \lambda\kappa.\kappa\ (\lambda x \lambda\kappa'.M^P\ \kappa')$$

4.4.3 Polarization for fun and profit

We have not exhausted the possibilities, of course. There are infinitely many polarizations of the simple types, and the focusing completeness theorem tells us that each of these results in a CPS translation. Whether or not these different polarizations are interesting is another story, but it's hard to rule them out. Before ending the chapter, let's consider one more polarization of the λ-calculus function space, very simple but unconventional:

$$\boxed{(\tau_1 \supset \tau_2)^* = \neg\tau_1^* \oplus \tau_2^*}$$

$$\begin{aligned} x^*\ \kappa &= \kappa\ \mathit{Id}_x \\ (\lambda x.M)^*\ \kappa\ &= \kappa\ \mathsf{INL}\,(x \mapsto M^*\ \kappa') \\ &\quad \text{where } \kappa' = (y \mapsto \kappa\ \mathsf{INR}\,(\mathit{Id}_y)) \\ (M_1\ M_2)^*\ \kappa\ &= M_1^*\ \kappa' \\ &\quad \text{where } \kappa' = (\mathsf{inl}\,\kappa' \mapsto M_2^*\ \kappa' \mid\ \mathsf{inr}\,x \mapsto \kappa\ \mathit{Id}_x) \end{aligned}$$

Here we have chosen to polarize the λ-calculus function space not using the negative function space $A^+ \to B^-$ of $\mathcal{L}$, but rather with the positive $\neg A \oplus B$. It may be surprising that this works: a functional value is either a continuation for its argument, or the value of its result, with no dependence between them. However, because the polarization is positive, λ-calculus terms are not translated directly into values, but rather to expressions with a free continuation variable. And we see in the translation $(\lambda x.M)^*$ above that the expression invokes the continuation variable κ *twice*. The first time, it calls κ with a τ_1^* continuation. Within that continuation, it has a value x of type τ_1, so now it evaluates M^*, which can then call κ with a value of type τ_2^* depending on x.

Naturally, if the body of the term $\lambda x.M$ does not actually depend on x, we could give a more efficient translation,

$$(\lambda x.M)^*\ \kappa\ = M^*\ \kappa' \text{ where } \kappa' = (y \mapsto \kappa\ \mathsf{INR}\,(\mathit{Id}_y))$$

By continuing down these lines, can we give a logical account of call-by-*need?*

4.5 Related Work

"Classical Curry-Howard". Since Griffin [1990] drew the connection between control operators and classical theorems (e.g., callcc and Peirce's Law), and Murthy [1992] and Parigot [1992] gave hope that classical logic could be provided a decent proof theory, there have been many different attempts at building a constructive interpretation of classical logic. These include different calculi by Barbanera and Berardi [1996], Ong and Stewart [1997] and Streicher and Reus [1998], among others. Stewart's dissertation [1999] represents a particularly sophisticated analysis, trying to judge the resulting natural deduction for classical logic by the sort of "internal justification" principles of proof-theoretic semantics described in Chapter 2. I explained the sense in which $\mathcal{L}$ is and is not such a Curry-Howard interpretation of classical logic:

- Any classical theorem can be *polarized* and inhabited by a term of $\mathcal{L}$, but...
- ...the meaning of positive types is essentially intuitionistic, and the meaning of negative types co-intuitionistic. Proof-by-contradiction is confined to negative types.

In other words, $\mathcal{L}$ gives a propositions-as-types interpretation of classical logic basically in the sense of being a syntactically elegant double-negation interpretation. This may not seem like a very satisfactory interpretation—have we made much progress on understanding the constructive content of classical logic since Kolmogorov [1925]?

I think polarization and focusing do provide some insights. On the one hand, they tell us that classical propositions can be given constructive readings which are not very different from the standard intuitionistic ones, or alternatively, dual to them. But then they also tell us that there is no *single* constructive content of a classical theorem, because there are many different ways to polarize a classical proposition. And finally, they provide a constructive interpretation of duality, which *is* an important and useful concept.

Computational duality. Beginning with Filinski's master's thesis [1989], a line of work has explored a duality between call-by-value and call-by-name evaluation in the presence of first-class continuations. Filinski was inspired by categorical duality, but did not make a connection to logic. The logical accounts came following attempts by Girard [1991a, 1993] and by Danos, Joinet, and Schellinx [1995, 1997] to understand cut-elimination for classical sequent calculus through the lens of linear logic and polarity. Ogata [2000] showed how to translate Danos et alia's LKT and LKQ into a CPS calculus (like the one we used in §4.4, also discussed by Thielecke [1997]), and Curien and Herbelin [2000] showed how LKT and LKQ could be directly annotated as call-by-name and call-by-value languages (respectively). Curien and Herbelin also defined a larger sequent calculus/language, of which CBN and CBV were fragments. However, this larger calculus was non-confluent, so they noted one can obtain confluence by imposing a global bias towards either call-by-value or call-by-name. The same approach has been more recently refined and exposited by Wadler [2003].

Again, in retrospect, it seems the main deficiency of these languages defined by Filinski [1989], Curien and Herbelin [2000], and Wadler [2003] was that call-by-value and call-by-name were interpreted as global policy decisions, rather than as local policies determined by types. This precluded the definition of a single language subsuming both call-by-value and call-by-name evaluation. However, such a type distinction *is* implicit (from a semantic perspective) in Selinger's *control categories* [2001], and explicit syntactically in Levy's *call-by-push-value* language [2001].

Call-By-Push-Value. In the Introduction I suggested that this work gives a proof-theoretic reconstruction of some of the ideas from CBPV. Let me try to explain the similarities between $\mathcal{L}$ and CBPV, the differences, and what the proof-theoretic approach has to offer beyond what is already present in Levy's account. The type structure of CBPV is very close to that of $\mathcal{L}$:

$$\begin{aligned} A &::= U\underline{B} \mid \sum_{i \in I} A_i \mid A \times A \\ \underline{B} &::= FA \mid \prod_{i \in I} \underline{B}_i \mid A \to \underline{B} \end{aligned}$$

Types A are *value types*, and types $\underline{B}$ are *computation types*. And clearly the analogy is,

$$\textit{value} \sim \textit{positive} \qquad \textit{computation} \sim \textit{negative}.$$

Like positive types, value types include possibly infinite sums (e.g., $\mathbf{N}$) and finite products ($\otimes$). Like negative types, computation types include possibly infinite products (e.g., $\mathcal{S}\uparrow\mathbf{B}$) and polarity mixing functions ($\to$). The coercions U and F play the role of $\downarrow$ and $\uparrow$, respectively.

The difference is that value types and computation types are not given symmetric treatment in CBPV, whereas positive and negative types are completely dual in $\mathcal{L}$. There is no $⅋$ connective in CBPV, nor the subtraction operator dual to implication. More to the point, there are no continuation variables in CBPV, and a term of type FA does not necessarily have access to its continuation (only in models of CBPV with control operators). Basically, we can summarize the difference between the type structure of $\mathcal{L}$ and the type structure of CBPV as the difference between polarizing classical logic and polarizing intuitionistic logic.

Depending on your perspective, this asymmetry in CBPV may be seen as an unnecessary restriction, or as an important gain in expressivity. But in either case, the point is that while this difference exists between CBPV and $\mathcal{L}$ as I have defined it here, it is not inherent to the ideas of polarity and focusing. It is possible to give an asymmetric, pattern-based formulation of focusing for intuitionistic logic (as I have done elsewhere [Zeilberger, 2008, Licata et al., 2008]), and then the type structure exactly mirrors that of CBPV. And conversely, Levy often analyzes CBPV in terms of a simpler language "jump-with-argument" (JWA), which is very close to $\mathcal{L}^+$.

So besides this subtle but ultimately inessential difference in symmetry, what does our proof-theoretic analysis add to the CBPV story, besides the simple fact that such a proof-theoretic analysis *exists?* Well, although the types of CBPV and $\mathcal{L}$ are very similar, their type systems are structured quite differently. As we have seen, the terms of $\mathcal{L}$ correspond to derivations in a sequent calculus, subject to the subformula property, and are defined generically by reference to patterns and variables. CBPV, in contrast, has a more standard, natural deduction-style definition, with introduction and elimination rules for each connective. The combination of the subformula property and the pattern-based definition is important, because as we saw, it often allows us to make generic arguments about programs of $\mathcal{L}$ and $\mathcal{L}^+$, without having to consider individual types. For example, we could give a standard syntactic type safety argument in the style of Wright and Felleisen [1994], but avoiding much of the usual bureaucracy of such arguments. (Our type safety theorem was a bit non-standard in being an *intrinsic* type safety theorem, but we will also give an extrinsic one in Chapter 6.) Likewise, the Separation Theorem was stated and proved generically. We will see more demonstrations of the power of patterns in Chapter 6 when we consider refinement types such as intersections and unions, which have not been studied in the CBPV setting.[11]

Ludics. Many of our definitions and results were inspired and informed by ludics [Girard, 2001]. There is a close correspondence between the terms of $\mathcal{L}$ and the designs of ludics. Roughly,

$$\begin{array}{cc} \textit{locus} \sim \textit{variable} & \textit{ramification} \sim \textit{frame} \\ \textit{positive design} \sim \textit{expression} & \textit{negative design} \sim \textit{substitution} \\ \multicolumn{2}{c}{\textit{cut-net} \sim \textit{program}} \end{array}$$

$\mathcal{L}$ is not just an alternate presentation of ludics, though. Our definitions were motivated by programming languages intuitions, and at times these diverge from ludics. Most notably in the notion of pattern itself, which is absent in ludics. Likewise, $\mathcal{L}$ has first-class notions of value and continuation, which are not used in ludics. The chronicle representation (§4.2.14) and the compound rules for the environment semantics (go^+ and go^-) show that values and continuations are strictly speaking unnecessary, because they can be factored out into expressions and substitutions. Nonetheless, because, they match the usual notions of value and continuation from operational semantics, they provide useful intuition for the meaning of types.

Ludics has been an inspiration for many others as well. Perhaps most closely related to our work, Terui [2008] has recently begun exploring a "computational ludics", as a first step towards bridging the gap between computability/complexity theory and logic/type theory.

Pattern-matching and realizability. Pattern-matching is a well-established and important feature of functional programming languages [Augustsson, 1987, Peyton Jones, 1987]. The higher-order formulation of pattern-matching given here as an iterated inductive definition is (as far as I know) new, although it bears some similarities to the approach of Coquand [1992]. The general idea of allowing computation in syntax is not really new, however. In a sense it is already present in the BHK realizability interpretation of intuitionistic logic, with the idea that a proof of $A \supset B$ is a procedure for converting proofs of A into proofs of B [Heyting, 1974, Kolmogorov, 1932]. Note, though, that our interpretation is only *second-order*, because patterns are first-order objects—whereas the realizability interpretation is arbitrarily higher-order. Realizability was put into practice in the NuPRL framework [Constable et al., 1986], and Howe [1991] explored how it leads to a notion of "computational open-endedness", i.e., the idea that even in constructive settings the meta-level functions are somehow arbitrary. We will have much more to say about this in the next chapter.

[11]It is worth noting that a notion of "ultimate pattern" similar to ours has shown up in games semantics interpretations of CBPV and JWA [Lassen and Levy, 2007]. But then why exclude it from syntax?

Chapter 5

Concrete notations for abstract derivations

It thus became apparent that the "finite standpunkt" is not the only alternative to classical ways of reasoning and is not necessarily implied by the idea of proof theory. An enlarging of the methods of proof theory was therefore suggested: instead of a restriction to finitist methods of reasoning, it was required only that the arguments be of a constructive character, allowing us to deal with more general forms of inference.

—Paul Bernays (quoted by Buchholz et al. [1981, p.55])

In the previous chapter we defined a new language, $\mathcal{L}$, but our aim was not to present this language for its own sake, but rather to use it as an example of a new, more abstract approach to language definition based on polarity and focusing. What made this approach unusual was that syntax was given a second-order definition: certain kinds of syntactic objects (positive continuations and negative values) were constructed as functions on more basic (first-order) syntactic objects, *patterns.* This allowed us to define a generic notion of computation while remaining in a typed setting, deriving the operational behavior of individual type constructors from their pattern-formation rules.

But it may seem a bit nebulous. What exactly are these functions on patterns? Because a type can have infinitely many patterns, different possibilities exist: primitive recursive functions, partial recursive functions, functional relations in ZFC set theory, etc. On the one hand, this computational open-endedness may be seen as a virtue, since it allows our language to be interpreted in many different settings. But on the other, it may be seen as a sign that the language definition is *too* open-ended. After all, syntax is something that ultimately has to be *written down,* and is supposed to communicate some meaning unambiguously. How do we write down these potentially infinitary functions on patterns, and how do we interpret them?

In this chapter, I will give two different, very concrete answers to these questions, by showing how to *embed* programs of $\mathcal{L}$ (or at least the $\mathcal{L}^+$ fragment) in two different meta-languages.

These meta-languages—Agda[1] and Twelf[2]—are real, working programming languages, and so we inherit a concrete syntax for writing down $\mathcal{L}$ programs in such a way that they can actually be type-checked and executed by machine. Both Agda's and Twelf's core type theories are intellectual descendents of Martin-Löf's dependent type theory [Nordström et al., 1990]—but down two very different paths: the type theory of Agda (iterated inductive-recursive definitions [Dybjer, 2000]) is descended from Martin-Löf's "theory of sets", while the type theory of Twelf (LF [Harper et al., 1993]) from Martin-Löf's "system of arities". As we will see, in each of these frameworks, certain aspects of $\mathcal{L}$ can be represented elegantly and concisely, while other aspects have to be encoded more indirectly. For our present purposes, though, this is an advantage: by manually going through the process of compiling some of the more abstract features of our language definition down to lower-level primitives, we get a better understanding of those features.

5.1 An embedding of $\mathcal{L}^+$ in Agda

In this section we will show how to represent the syntax and semantics of $\mathcal{L}^+$ in Agda. We restrict to $\mathcal{L}^+$ rather than $\mathcal{L}$ because it conveys most of the essential features. Some familiarity with dependently typed functional programming—such as in Coq or NuPRL, if not with Agda in particular—is probably a prerequisite for understanding this section. A classic introduction is provided by Nordström et al. [1990], and McBride [2004] provides a more modern one.

The first thing we do to encode the syntax of $\mathcal{L}^+$ is simply define a grammar of types. An open-ended definition would be desirable if we were using the language for real programming, but here we will simply copy the type constructors of §4.2.1:

```
data Pos : Set where
  _+_ :  Pos -> Pos -> Pos            -- binary sums
  void : Pos                          -- void
  _*_ :  Pos -> Pos -> Pos            -- binary products
  unit : Pos                          -- unit
  ¬ :     Pos -> Pos                  -- continuations
  bool : Pos                          -- booleans
  nat :  Pos                          -- naturals
  dom :  Pos                          -- recursive domain

infixr 15 _+_
infixr 16 _*_
```

Pos (the Agda type representing $\mathcal{L}^+$ types) is just an ordinary algebraic data type. (Agda has some neat parsing tricks, so we tell it the precedence of the right-associative, infix product and sum constructors.) Next we define a data type of frames, represented as join lists of continuation holes:

[1]The Agda code included in this chapter are actually written in "Agda 2" (v2.1.3), the redesign and reimplementation by Ulf Norell [2007] of an older language, also called Agda. At the moment, Agda 2 is in active development and evolving rapidly, so the code included here may or may not be syntactically well-formed a few years/months down the line. Hopefully, though, it will still make sense. A good source of information about Agda, http://wiki.portal.chalmers.se/agda/, also may or may not exist.

[2]The Twelf [Pfenning and Schürmann, 1999] code in this chapter is written in version 1.5. Hopefully when you are reading this the Twelf wiki still exists at http://twelf.plparty.org/.

```
data Frame : Set where
  ●_ : Pos -> Frame
  · : Frame
  _,_ : Frame -> Frame -> Frame
infixr 13 _,_
```

Now, we want to encode the containment relationship for frames. In Agda, we encode it as an inductive type family (note that the arguments of the ∀ quantifier within curly braces represent implicit arguments to the constructors, which Agda will attempt to infer):

```
infix 10 _∈_
data _∈_ : Frame -> Frame -> Set where
  here : ∀ {Δ} -> Δ ∈ Δ
  left  : ∀ {Δ Δ₁ Δ₂} -> Δ ∈ Δ₁ -> Δ ∈ (Δ₁ , Δ₂)
  right : ∀ {Δ Δ₁ Δ₂} -> Δ ∈ Δ₂ -> Δ ∈ (Δ₁ , Δ₂)
```

The constructors here match those we gave in §4.2.2 when we discussed what programming in $\mathcal{L}^+$ would look like without variable names. A proof of a containment $\Delta_1 \in \Delta_2$ is analogous to a de Bruijn index, representing the path we take to get from Δ_2 to its sublist Δ_1. For example, we could represent the variable $\kappa_2 : \bullet B \in (\kappa_1 : \bullet A, \kappa_2 : \bullet B, \kappa_3 : \bullet C)$ by the following index:

```
κ₂ : ∀ { A B C } -> ● B ∈ (● A , ● B , ● C)
κ₂ = right (left here)
```

So what we are starting to do here is give a nameless representation of $\mathcal{L}^+$. This is the easiest path towards encoding the language in Agda—otherwise we would have to explicitly build much of the machinery for dealing with variable names, and it would complicate the encoding. However, it will make the process of actually writing programs in our embedding more tedious, because we have to reason about indices.

Once we have defined frames, we can define patterns, also as an inductive family:

```
infix 10 _⊩_
data _⊩_ : Frame -> Pos -> Set where
   hole : ∀ {A} -> ● A ⊩ ¬ A
   #<> : · ⊩ unit
   #inl : ∀ {Δ A B}
     -> Δ ⊩ A
     -> Δ ⊩ A + B
   #inr : ∀ {Δ A B}
     -> Δ ⊩ B
     -> Δ ⊩ A + B
   #pair : ∀ {Δ₁ Δ₂ A B}
     -> Δ₁ ⊩ A -> Δ₂ ⊩ B
     -> Δ₁ , Δ₂ ⊩ A * B
   #tt : · ⊩ bool
   #ff : · ⊩ bool
   #z : · ⊩ nat
   #s_ : ∀ {Δ}
     -> Δ ⊩ nat
     -> Δ ⊩ nat
   #dn : ∀ {Δ}
     -> Δ ⊩ nat
```

```
      -> Δ ⊩ dom
    #dk : ∀ {Δ}
      -> Δ ⊩ ¬ dom
      -> Δ ⊩ dom
  infixr 15 #s_
```

These constructors let us build patterns much the same way as we did in Chapter 4, except that they can never name a continuation variable, only give a place for a hole.

Now that we have patterns, we are almost ready to define the terms of $\mathcal{L}^+$. We first need to give the the boring definition of contexts, and containment in a context:

```
  data Ctx : Set where
    ·· : Ctx
    _,,_ : Ctx -> Frame -> Ctx
  infixl 12 _,,_

  infixr 10 _∈∈_
  data _∈∈_ : Frame -> Ctx -> Set where
    x0 : ∀ {Δ Γ Δ'} -> Δ ∈ Δ' -> Δ ∈∈ Γ ,, Δ'
    xS_ : ∀ {Δ Γ Δ'} -> Δ ∈∈ Γ -> Δ ∈∈ Γ ,, Δ'
  infixr 15 xS_
```

We also introduce an Agda type J, which simply names the different kinds of judgments:

```
  data J : Set where
    True : Pos -> J
    False : Pos -> J
    All_ : Frame -> J
    # : J
  infix 10 All_
```

Finally, we can define a type family Γ ⊢ J, whose inhabitants will be the terms of our language:

```
  infix 8 _⊢_
  codata _⊢_ : Ctx -> J -> Set where
```

Notice we mark this definition as "codata" rather than "data". This tells Agda to treat the definition that follows as coinductive rather than inductive, and thus to be more relaxed when we build non-well-founded terms. Formally, though, we don't really mean to restrict to *productive* terms either, because we want to be able to define continuations by arbitrary partial recursive functions on patterns. Fortunately, Agda works just fine as a partial type theory—it will warn us when a term is not productive, but still accept it.

Let us move on to the definition of Γ ⊢ J. First, we define values as patterns under substitutions:

```
    _[_] : ∀ {Γ A Δ}
      -> Δ ⊩ A -> Γ ⊢ All Δ
      -> Γ ⊢ True A
```

Note that _[_] is a *mixfix* operator: the parser lets us write values as `p [ σ ]`. Now, we define continuations as maps from patterns to expressions:

```
    con_ : ∀ {Γ A}
      -> (∀ {Δ} -> Δ ⊩ A -> Γ ,, Δ ⊢ #)
      -> Γ ⊢ False A
```

This is really the key trick. We are giving a higher-order definition of the syntax of our language, by embedding meta-level (i.e., Agda-level) functions over patterns as object-level (i.e., $\mathcal{L}^+$-level) continuations. Moreover, we are making essential use of the fact that the meta-level function space is *dependent:* the type of the constructor `con` doesn't just tell us that continuations are maps from patterns to expressions, but that they map A-patterns with frame Δ to expressions in a context extended by Δ. We will soon explore what this means in greater depth, with examples. Now, we finish the definition of $\mathcal{L}^+$ by defining substitutions and expressions:

```
    -- combinators for substitutions
    sHole : ∀ {Γ A} -> Γ ⊢ False A -> Γ ⊢ All • A
    sNil : ∀ {Γ} -> Γ ⊢ All ·
    sJoin : ∀ {Γ Δ₁ Δ₂}
      -> Γ ⊢ All Δ₁ -> Γ ⊢ All Δ₂ -> Γ ⊢ All Δ₁ , Δ₂

    -- a value thrown to a continuation variable
    throw : ∀ {Γ A}
      -> • A ∈∈ Γ -> Γ ⊢ True A
      -> Γ ⊢ #

    -- immediate failure
    ℧ : ∀ {Γ} -> Γ ⊢ #
```

Before considering some specific examples, let us make some general observations, mirroring the ones we made in Chapter 4. First, we can formally derive the analogue of Observation 4.2.3, showing how to apply a substitution to a continuation variable in its frame to obtain a continuation, and conversely, how to construct a substitution given a map from continuation variables to continuations:

```
  appSub : ∀ {Γ Δ}
    -> Γ ⊢ All Δ
    -> (∀ {A} -> • A ∈ Δ -> Γ ⊢ False A)
  appSub (sHole K) here ~ K
  appSub sNil ()          -- the empty frame has no variables, so this case is refuted
  appSub (sJoin σ₁ σ₂) (left κ)  ~ appSub σ₁ κ
  appSub (sJoin σ₁ σ₂) (right κ) ~ appSub σ₂ κ

  sub_ : ∀ {Γ Δ}
    -> (∀ {A} -> • A ∈ Δ -> Γ ⊢ False A)
    -> Γ ⊢ All Δ
  sub_ {Δ = ·} f ~ sNil
  sub_ {Δ = Δ₁ , Δ₂} f ~ sJoin (sub \x -> f (left x)) (sub \x -> f (right x))
  sub_ {Δ = • _} f ~ sHole (f here)
```

(Here the clauses are specified with `~` rather than `=`, because the type family `Γ ⊢ J` is coinductive.) Similarly, we can show that given an A-continuation and an A-pattern with frame Δ, we can obtain an expression in a context extended by Δ. The proof is quite trivial, since it just reduces to Agda function application:

```
appCon : ∀ {Γ A}
  -> Γ ⊢ False A
  -> ∀ {Δ} -> Δ ⊩ A -> Γ ,, Δ ⊢ #
appCon (con K) p ~ K p
```

Now, let's try to build the $\mathcal{L}^+$ terms witnessing the identity principles. To do that, we need some easy lemmas about the transitivity of containment:

```
trans∈ : {Δ₁ Δ₂ Δ₃ : Frame}
        -> Δ₁ ∈ Δ₂ -> Δ₂ ∈ Δ₃ -> Δ₁ ∈ Δ₃
trans∈ Δ₁ here = Δ₁
trans∈ here Δ₁ = Δ₁
trans∈ Δ₁ (left Δ₂)  = left (trans∈ Δ₁ Δ₂)
trans∈ Δ₁ (right Δ₂) = right (trans∈ Δ₁ Δ₂)

trans∈∈ : ∀ {Δ₁ Δ₂ Γ} -> Δ₁ ∈ Δ₂ -> Δ₂ ∈∈ Γ -> Δ₁ ∈∈ Γ
trans∈∈ x (x0 y) = x0 (trans∈ x y)
trans∈∈ x (xS y) = xS (trans∈∈ x y)
```

We can then build the identity continuation and the identity substitution, with mutually recursive definitions:

```
mutual
  IdCon : ∀ {Γ A} -> • A ∈∈ Γ -> Γ ⊢ False A
  IdCon κ ~ con \p -> throw (xS κ) (p [ IdSub (x0 here) ])

  IdSub : ∀ {Γ Δ} -> Δ ∈∈ Γ -> Γ ⊢ All Δ
  IdSub {Δ = • A} κ ~ sHole (IdCon κ)
  IdSub {Δ = ·} σ ~ sNil
  IdSub {Δ = Δ₁ , Δ₂} σ ~ sJoin (IdSub (trans∈∈ (left here) σ))
                               (IdSub (trans∈∈ (right here) σ))
```

These definitions mirror the definitions in §4.2.7, except for the bureaucracy involving context indices. Agda verifies that the identity terms are productive.

We won't consider the binary composition principles here—the reader can consult Appendix **??**. But for both binary composition and for the environment semantics (which we define below), we need the weakening property on contexts. As I explained in Chapter 4, on the representation of $\mathcal{L}^+$ using scoped variable names, weakening comes "for free", since it has no effect on terms. With the de Bruijn-ish representation, though, we need to give an explicit proof of weakening. To even *state* weakening (and composition), though, we first need to define a notion of splitting of contexts:

```
data split : Ctx -> Ctx -> Frame -> Ctx -> Set where
  here : ∀ {Δ Γ}
       -> split (Γ ,, Δ) Γ Δ ··
  skip : ∀ {Γ Γ₁ Δ Γ₂ Δ'}
       -> split Γ Γ₁ Δ Γ₂
       -> split (Γ ,, Δ') Γ₁  Δ (Γ₂ ,, Δ')
  assoc : ∀ {Γ Δ₁ Δ₂ Γ₁ Δ Γ₂}
       -> split (Γ ,, Δ₁ ,, Δ₂) Γ₁ Δ Γ₂
       -> split (Γ ,, ( Δ₁ , Δ₂ )) Γ₁ Δ Γ₂
  nil : ∀ {Γ Γ₁ Δ Γ₂}
```

```
      -> split Γ Γ₁ Δ Γ₂
      -> split (Γ ,, ·) Γ₁ Δ Γ₂
```

The relation `split Γ Γ₁ Δ Γ₂` says that Γ can be split into an initial context Γ_1, a hole plugged in by Δ, and a trailing context Γ_2. The weakening property of variable indices says that if Γ can be split as $\Gamma = \Gamma_1(\Delta),\Gamma_2$, and Δ_0 is in the concatenation of Γ_1 and Γ_2, then Δ_0 is in Γ:

```
_++_ : Ctx -> Ctx -> Ctx
Γ₁ ++ ·· = Γ₁
Γ₁ ++ (Γ₂ ,, Δ) = (Γ₁ ++ Γ₂) ,, Δ
infixr 12 _++_

weakvar : ∀ {Γ Γ₁ Δ Γ₂ Δ₀}
     -> split Γ Γ₁ Δ Γ₂ -> Δ₀ ∈∈ Γ₁ ++ Γ₂
     -> Δ₀ ∈∈ Γ
```

The proof of `weakvar` is not so interesting, so we omit it here. Once we have it, we can define weakening of terms simply by crawling through and applying `weakvar` to any variables:

```
weaken : ∀ {Γ Γ₁ Δ Γ₂ J}
     -> split Γ Γ₁ Δ Γ₂ -> Γ₁ ++ Γ₂ ⊢ J
     -> Γ ⊢ J
weaken s (p [ σ ]) ~ p [ weaken s σ ]
weaken s (con φ) ~ con (\p -> weaken (skip s) (φ p))
weaken s (sHole K) ~ sHole (weaken s K)
weaken s (sNil) ~ sNil
weaken s (sJoin σ₁ σ₂) ~ sJoin (weaken s σ₁) (weaken s σ₂)
weaken s (throw κ V) ~ throw (weakvar s κ) (weaken s V)
weaken s ℧ ~ ℧
```

Finally, let us encode the $\mathcal{L}^+$ environment semantics. First we define environments and the lookup operation:

```
data Env : Ctx -> Set where
  emp : Env ··
  _bind_ : ∀ {Γ Δ} -> Env Γ -> Γ ⊢ All Δ -> Env (Γ ,, Δ)
infixl 8 _bind_

lookup : ∀ {Γ A}
      -> Env Γ -> • A ∈∈ Γ -> Γ ⊢ False A
lookup emp ()   -- impossible pointer into empty environment
lookup (γ bind σ) (x0 κ) = weaken here (appSub σ κ)
lookup (γ bind σ) (xS κ) = weaken here (lookup γ κ)
```

And then we define programs, results, and the evaluation relation:

```
data Prog : Set where
  prog : ∀ {Γ} -> Env Γ -> Γ ⊢ # -> Prog

codata Result : Set where
  abort : Result
  step : Result -> Result
```

```
eval : Prog -> Result
eval (prog γ (throw κ (p [ σ ]))) ~
     step (eval (prog (γ bind σ) (appCon (lookup γ κ) p)))
eval (prog γ ℧) ~ abort
```

Note that for concision, we are encoding the compound go$^+$ rule, rather than the separate lookup$^+$ and bind/call$^+$ rules. We are also making somewhat more precise here the approach to divergence: in the style of Leroy [2006], `Result`s are defined coinductively, and divergence is represented by an endless series of `step`s.

At last! We have our Agda embedding of $\mathcal{L}^+$, both its syntax and semantics. Can we start writing programs now? We will begin by writing a silly one:

```
selfapp : ∀ {Γ} -> Γ ⊢ False dom
selfapp ~ con K
  where
    K : ∀ {Δ} -> Δ ⊩ dom -> _ ,, Δ ⊢ #
    K (#dk hole) ~ throw (x0 here) (#dk hole [ sHole selfapp ])
    K (#dn _) ~ ℧
```

`selfapp` is the self-applying **D**-continuation defined in §4.2.8. Here the syntax looks more or less like it did there, except that we have to reference the continuation variable with a de Bruijn index. Observe that we only give two clauses when defining `selfapp`, relying on Agda's coverage checker to verify that the meta-level function is exhaustive over `dom`-patterns. (The `#dn` case can't be ruled out, although it is senseless in terms of the intended semantics of `selfapp`, so we simply return ℧ as a sort of error. We will have more to say about this in Chapter 6.) Also observe that the definition of `selfapp` is *circular:* `selfapp` is used as an embedded substitution. The definition is nonetheless productive.

By applying `selfapp` to itself, we can write the diverging program:

```
ωω : Prog
ωω = prog (emp bind sHole selfapp)
         (throw (x0 here) (#dk hole [ sHole selfapp ]))
```

Let's try to verify that $\omega\omega$ really diverges—or at least that it runs for awhile. . . We start by defining a procedure that decides whether a program is safe (i.e., does not abort) for n steps:

```
data Bool : Set where
  True : Bool
  False : Bool

data Nat : Set where
  Z : Nat
  S_ : Nat -> Nat
infixr 12 S_

safeN : Result -> Nat -> Bool
safeN R Z = True
safeN abort (S _) = False
safeN (step R) (S n) = safeN R n
```

We could just run `safeN` on $\omega\omega$ and look at the result, but since we are working in dependent type theory, it's more fun to have the typechecker do it. We use a well-known trick:

```
data Void : Set where
data Unit : Set where
  u : Unit

isTrue : Bool -> Set
isTrue True = Unit
isTrue False = Void

isFalse : Bool -> Set
isFalse True = Void
isFalse False = Unit

safe10ωω : isTrue (safeN (eval ωω) (S S S S S S S S S S Z))
safe10ωω = u
```

The type of `safe10ωω` verifies that $\omega\omega$ runs for at least ten steps.

Now consider another, slightly less silly $\mathcal{L}^{+}$ term:

```
isEven : ∀ {Γ} -> • bool ∈∈ Γ -> Γ ⊢ False nat
isEven κ ~ con K
  where
    K : ∀ {Δ} -> Δ ⊩ nat -> _ ,, Δ ⊢ #
    K #z ~ throw (xS κ) (#tt [ sNil ])
    K (#s #z) ~ throw (xS κ) (#ff [ sNil ])
    K (#s #s n) ~ K n
```

This is the function of type $\mathbf{N} \to \mathbf{B}$ that decides whether a number is even, expressed as a continuation transformer. We can also define a generic combinator for building $\mathbf{B}$ continuations, given a pair of expressions:

```
branch : ∀ {Γ} -> Γ ⊢ # -> Γ ⊢ # -> Γ ⊢ False bool
branch E₁ E₂ ~ con K
  where
    K : ∀ {Δ} -> Δ ⊩ bool -> _ ,, Δ ⊢ #
    K #tt ~ weaken here E₁
    K #ff ~ weaken here E₂
```

Now, consider the following program:

```
even9 : Prog
even9 = prog (emp
      bind sHole selfapp
      bind sHole (branch (throw (x0 here) (#dk hole [ sHole selfapp ])) ℧)
      bind sHole (isEven (x0 here)))
        (throw (x0 here) (#s #s #s #s #s #s #s #s #s #z [ sNil ]))
```

This might be a bit difficult to parse. Ultimately, `even9` it is supposed to test whether the number nine is even. It begins by binding `selfapp` in the environment, and then defines a `bool`-continuation which diverges on true and aborts on false, and binds that. Finally, it binds `isEven` with the aforementioned `bool`-continuation as its return address, and calls `isEven` on nine. So, how does this actually behave? Well, it's safe for at least one step:

```
safe1even9 : isTrue (safeN (eval even9) (S Z))
safe1even9 = u
```

Indeed, it's safe for two steps:

```
safe2even9 : isTrue (safeN (eval even9) (S S Z))
safe2even9 = u
```

But after three steps, it aborts:

```
unsafe3even9 : isFalse (safeN (eval even9) (S S S Z))
unsafe3even9 = u
```

This might be a bit counterintuitive: how can the program tell that nine is not even in just three steps? The point is that the second step is really a large step, because we defined `isEven` using computation at the meta-level. The continuation immediately decides to call `branch` with `#ff`, which then decides to run $\mho$.

With this we conclude our discussion of the Agda embedding. The reader is invited to try writing some programs of their own.

5.2 An embedding of $\mathcal{L}^+$ in Twelf

Essentially, the Agda encoding is a direct transcription of the logical rules of Chapter 2. That is, it follows the definition of $\mathcal{L}^+$ almost literally, except where in Chapter 4 we introduced some additional conventions for referring to continuation variables by labels. We could have given an explicit treatment of variable names (see, e.g., Aydemir et al. [2008] for a survey of different possible techniques), but this would have necessitated introducing additional scaffolding which was not present in our original, albeit informal definition of $\mathcal{L}^+$. We did not do anything *new* in the above code, except by giving more explicit proofs of some lemmas we already used in Chapter 2.

In other words, the dependent type theory of Agda lets us take seriously our informal ideas about computation in syntax, but refuses to take seriously our informal ideas about variable binding.

In Twelf, the situation is precisely reversed. As mentioned above, the type theory of Agda is inspired by Martin-Löf's theory of sets, which includes as one essential feature the dependent function space $\Pi x : A.B$. The type $\Pi x : A.B$ represents *computations* from terms t of type A to terms of type $B(t)$. In contrast, the type theory of Twelf is inspired by Martin-Löf's system of arities, which includes a different dependent function space, written $(x : A).B$.[3] The type $(x : A).B$ represents a *substitution function* fom terms t of type A to terms of type $B(t)$. A substitution function is something much more limited than a computation: it cannot inspect its argument, only use it as a black box.

This weakness of substitution functions in comparison to computational functions is also a strength, because it means they exactly encode the notion of variable binding: the type $(x : A).B$ represents a $B(x)$ with a *hole* for an $x : A$. This enables a technique commonly known as *higher-order abstract syntax* [Pfenning and Elliott, 1988], whereby different variable binding constructs in the syntax of a language are directly represented by the LF function space. But it also means that we fundamentally cannot use $(x : A).B$ to represent pattern-matching, because it allows

[3]In the notation of Nordström et al. [1990]. The original LF paper [Harper et al., 1993], as well as subsequent papers on LF and Twelf, use the same notation $\Pi x : A.B$ for this function space.

no inspection of x, and the whole point of pattern-matching is to decompose/case-analyze the argument.[4]

So can we give an embedding of $\mathcal{L}^+$ in Twelf, without introducing a lot of extra scaffolding? Yes we can, using an old trick due to Reynolds [1972]. *Defunctionalization* is a technique for taking a higher-order functional program and turning it into a program with only first-order functions. The rough idea is to: 1. Give each function in a program a unique tag (i.e., some element of a first-order data type), 2. Pass tags around instead of actual functions, and 3. Define a separate "apply" function, which describes how to apply tags (denoting functions) to arguments. A slight complication is that function bodies may reference escaping variables, so defunctionalization has to be preceded by closure conversion. In the Twelf code below, I won't assume prior background with defunctionalization, but the reader may consult Danvy and Nielsen [2001] for a good introduction. I will assume some familiarity with Twelf. Again, the Twelf Wiki is a good source of information, as is the article by Harper and Licata [2007].

As in the Agda embedding, we begin by describing the syntax of types. However, in Twelf, it is much easier to define the syntax of general recursive types because we don't need to invent a notion of type substitution, deriving it from LF. So, we give a somewhat more general signature than we gave in the Agda embedding:

```
pos : type.
int :  pos.                               % primitive integers
+ :    pos -> pos -> pos.                 % binary sums
void : pos.                               % void
* :    pos -> pos -> pos.                 % binary products
unit : pos.                               % unit
¬ :    pos -> pos.                        % continuations
rec : (pos -> pos) -> pos.                % recursive types

%infix right 13 +.
%infix right 14 *.
```

Now we can define various useful derived types and type constructors:

```
bool : pos                           % booleans
    = unit + unit.
nat : pos                            % unary nats
    = rec [X] unit + X.
list : pos -> pos                    % cons lists
    = [A] rec [X] unit + A * X.
→ : pos -> pos -> pos                % CBV-CPS functions
    = [A] [B] ¬ (A * ¬ B).
    %infix right 12 →.
D : pos                              % domain D = D -> D
    = rec [X] X → X.
```

Observe that we are defining the recursive type nat in addition to the type int named above.

[4]Ironically, the very different technique I have described of representing object-level pattern-matching by functions in the meta-lanaguage could also reasonably be called "higher-order abstract syntax". This was done by Zeilberger [2008], probably a bit confusingly. Both forms of encoding are higher-order—but one uses the powerful function space $\Pi x : A.B$ to encode pattern-matching, and the other uses the weak function space $(x : A).B$ to encode variable binding/substitution. I am not sure of the best terminology to unite these concepts.

The $\mathcal{L}^+$ type `int` will stand for "machine integers", so to speak, represented by an LF type `i` with some basic arithmetic operations (here we suffice with addition):

```
i : type.
z : i.
s : i -> i.   %prefix 10 s.

add : i -> i -> i -> type.
%mode add +M +N -P.
add/z : add z N N.
add/s : add (s M) N (s P) <- add M N P.
%worlds () (add M _ _).
%total (M) (add M _ _).
%unique add +M +N -P.
```

`add` is just the usual logic programming definition of addition, which Twelf verifies is a functional relation (i.e., for every `M` and `N`, there is a unique `P` such that `add M N P`). In the code below, we will appeal to `add` when defining $\mathcal{L}^+$ continuations over `ints`. In contrast, we won't assume any operations for the $\mathcal{L}^+$ type `nat`, which stands for the usual recursive datatype definition.

Now that we have some types, we can define frames, just as we did in Agda:

```
frame : type.
· : frame.
, : frame -> frame -> frame.
● : pos -> frame.
%infix right 11 ,.
```

And now that we have frames, we can define patterns:

```
⊩ : frame -> pos -> type.
%infix none 9 ⊩.

n   : i -> · ⊩ int.
inl : DΔ ⊩ A -> DΔ ⊩ A + B.
inr : DΔ ⊩ B -> DΔ ⊩ A + B.
u   : · ⊩ unit.
pair : DΔ₁ ⊩ A -> DΔ₂ ⊩ B -> DΔ₁ , DΔ₂ ⊩ A * B.
fold : DΔ ⊩ A (rec A) -> DΔ ⊩ rec A.
hole : ● A ⊩ ¬ A.
```

We can also define some abbreviations for patterns for derived types:

```
tt : · ⊩ bool = inl u.
ff : · ⊩ bool = inr u.
zz : · ⊩ nat = fold (inl u).
ss : DΔ ⊩ nat -> DΔ ⊩ nat = [p] fold (inr p).  %prefix 9 ss.
nil : · ⊩ list A = fold (inl u).
cons : DΔ₁ ⊩ A -> DΔ₂ ⊩ list A -> DΔ₁ , DΔ₂ ⊩ list A
 = [p1] [p2] fold (inr (pair p1 p2)).
```

Now comes the interesting part—the representation of $\mathcal{L}^+$ terms—where the Twelf embedding will really diverge from the Agda embedding. Instead of representing variables as de Bruijn-ish indices into an explicit context, we will represent them as actual variables in the implicit, LF context. So, we declare a type family `tm` J, representing derivations of J, without mentioning any explicit context:

```
j : type.
true : pos -> j.    %prefix 10 true.
false : pos -> j.   %prefix 10 false.
all : frame -> j.   %prefix 10 all.
# : j.

tm : j -> type.
```

However, because we want the ability to bind a list of multiple variables at a time (during pattern-matching), which is not directly representable by LF types, we define an auxiliary judgment $\Delta \vdash J$, which simply transfers variables into the implicit context one-by-one (this is the so-called "explicit contexts" technique of Crary [2009]):

```
⊢ : frame -> j -> j.
%infix right 9 ⊢.

λ_ : tm J -> tm (DΔ ⊢ J).
λ, : tm (DΔ1 ⊢ DΔ2 ⊢ J) -> tm (DΔ1 , DΔ2 ⊢ J).
λcon : (tm (false A) -> tm J) -> tm (• A ⊢ J).
λsub : (tm (all DΔ) -> tm J) -> tm (DΔ ⊢ J).
%prefix 9 λ_. %prefix 9 λ,. %prefix 9 λcon. %prefix 9 λsub.
```

The constructors `λ_` and `λ,` just express weakening and associativity of explicit contexts. `λcon` and `λsub` do the interesting work: the former binds a continuation variable in the implicit context, while the latter binds an entire frame as a *substitution variable.* Note that this is a slightly simplified view from Chapter 4: we are not distinguishing continuations and continuation variables as syntactic classes, instead viewing continuation variables literally as LF variables for continuations. This means that there will be more terms in the embedding than only *canonical* $\mathcal{L}^+$ terms—basically we are directly stipulating the identity principles, rather than deriving them. We do not have to do things this way, but it lets us take greater advantage of the logical framework. In particular, substitution for variables is free.

Now we begin to define terms in the implicit context. As usual, a value is a pattern and a substitution:

```
val : DΔ ⊩ A -> tm (all DΔ) -> tm (true A).
```

But observe the rule does not mention any explicit context. Let us skip over continuations for now—assuming we know how to build them, we can also build substitutions:

```
shole : tm (false A) -> tm (all • A).
snil : tm (all ·).
sjoin : tm (all DΔ1) -> tm (all DΔ2) -> tm (all DΔ1 , DΔ2).
```

Finally, we define different kinds of expressions:

```
throw : tm (false A) -> tm (true A) -> tm #.
let : tm (all DΔ) -> tm (DΔ ⊢ #) -> tm #.
℧ : i -> tm #.
```

Again, here there is a subtle difference with the definition in Chapter 4: we are directly encoding the composition principles, rather than only allowing expressions of the form $\kappa\ V$. (We are also indexing $\mho$ by an integer, but that is merely to have some more interesting observable results in the examples below.)

At last, the moment of truth (or rather, the moment of falsehood): how do we define continuations? As we said, the trick boils down to defunctionalization. What that means here is that instead of giving a generic constructor for inhabiting the LF type `tm (false A)` by maps from patterns to expressions (analogous to the `con` constructor in the Agda embedding), we will instead treat constants of type `tm (false A)` as *tags*, and define an "apply function" that does the actual work of associating—for each tag—patterns with expressions. We call this function `body`, since it defines the body of the continuation denoted by each tag. Because the operational semantics of Twelf are based on logic programming, we actually declare `body` as a relation:[5]

```
body : tm (false A) -> DΔ ⊩ A -> tm (DΔ ⊢ #) -> type.
%mode body +K +P -E.
```

Of course, since `body` is only declared as a relation, it isn't forced to behave like a function—but with the appropriate keywords, we can have Twelf verify that it is in fact a total functional relation:

```
%worlds () (body K P _).
%total P (body K P _).
%unique body +K +P -E.
```

The check is actually trivial at this point, because we haven't inhabited the LF type `tm (false A)` with any continuation tags. Twelf allows us to spread out the definition of `body` throughout the file, which is convenient because we can place the `body` clauses relevant to a particular tag immediately after its declaration. Then, adding `%total` and `%unique` declarations will induce Twelf to check that our pattern-matching definitions really are exhaustive and non-redundant.

Before we define any particular continuations, though, we can already define the generic operational semantics by appealing to `body`. First, we define results:

```
result : type.
halt : i -> result.
```

Twelf does not (yet) support coinductively defined relations, so for simplicity we assume that the only possible result is termination with some integer. The operational semantics itself is a slight variant of the environment semantics: rather than maintaining an explicit environment, we use LF substitution. First, we define the (total functional) relation `load`, which takes a substitution σ for Δ, a term t in a context extended by Δ, and returns the term $t[\sigma]$:

```
load : tm (all DΔ) -> tm (DΔ ⊢ J) -> tm J -> type.
%mode load +Sσ +T -T'.

ld/tm : load Sσ (λ_ T) T.
ld/join : load (sjoin Sσ₁ Sσ₂) (λ, T) T''
          <- load Sσ₁ T T'
```

[5] The Twelf embedding could be easily adapted to functional languages based on LF, such as Delphin [Poswolsky and Schürmann, 2008] or Beluga [Pientka and Dunfield, 2008]. In that case, `body` would actually be a (computational) function.

```
          <- load Sσ2 T' T''.
ld/con : load (shole K) (λcon T*) (T* K).
ld/sub : load Sσ (λsub T) (T Sσ).
%worlds () (load _ _ _).
%total (Sσ) (load Sσ _ _).
%unique load +Sσ +T -T.
```

And then we define eval, which evaluates an expression to a result:

```
eval : tm # -> result -> type.
%mode eval +E -R.

ev/load : eval (let Sσ E) R
       <- load Sσ E E'
       <- eval E' R.
ev/throw : eval (throw K (val P Sσ)) R
       <- body K P E
       <- eval (let Sσ E) R.
ev/℧ : eval (℧ N) (halt N).

%worlds () (eval _ _).
%covers eval +E -R.
```

The rules ev/load and ev/throw are analogous to the $\mathcal{L}^+$ environment semantics rules lookup^+ and $\mathrm{bind/call}^+$, except that they don't use an explicit environment. Case ev/throw is where defunctionalization comes in: instead of directly computing $K(p) = E$ by evaluating K on p, we appeal to body. The eval relation is not in general going to be total, because we can write non-terminating programs. However, we can ask that eval covers all the different kinds of expressions.[6]

With that we are done encoding the generic type system and operational semantics of $\mathcal{L}^+$ in Twelf, and can now consider some examples. To begin, we define a pair of simple but useful defunctionalized continuations:

```
ignore : tm (false A).
ignore/p : body ignore P (λ_ ℧ z).

exit : tm (false int).
exit/n : body exit (n N) (λ_ ℧ N).
```

ignore is the A-continuation that ignores its argument and just aborts with zero, while exit is the int-continuation that reads an integer and aborts with that integer. Observe the overall form of the definition of continuations in defunctionalized style: we declare a constant K : tm (false A), and then give clauses inhabiting body K P1 E1, body K P2 E2, etc. Here, we only used a single body clause for each continuation, and we can verify that that is enough:

```
%total P (body K P _).
%unique body +K +P -E.
```

Nothing forces us to only define continuation *constants,* and it is quite useful to define constructors that build defunctionalized continuations out of other objects. For example, given two expressions E_z and E_{nz}, we can build an int-continuation iszero E_z E_{nz} that executes E_z if its argument is zero, and E_{nz} if its argument is non-zero:

```
iszero : tm # -> tm # -> tm (false int).
iszero/z  : body (iszero Ez Enz) (n z) (λ_ Ez).
iszero/nz : body (iszero Ez Enz) (n (s _)) (λ_ Enz).
```

Observe that now we are using two `body` clauses to define the continuation by pattern-matching. We can also appeal to side-conditions when giving the `body` clauses themselves. For example, we can define `plus`, the continuation transformer implementing addition of `int`s, by appealing directly to "native arithmetic", i.e., the relation `add` defined above:

```
plus : tm (false int) -> tm (false (int * int)).
plus/mn : body (plus K) (pair (n M) (n N)) (λ_ throw K (val (n P) snil))
             <- add M N P.
```

Compare this with `plus'`, the corresponding continuation transformer for `nat`s, defined in terms of an auxiliary continuation transformer `succ`:

```
succ : tm (false nat) -> tm (false nat).
succ/n : body (succ K) N (λsub [σ] throw K (val (ss N) σ)).

plus' : tm (false nat) -> tm (false (nat * nat)).
plus'/zn : body (plus' K) (pair zz N) (λ, λ_ λsub [σ]
                throw K (val N σ)).
plus'/sn : body (plus' K) (pair (ss M) N) (λsub [σ]
                throw (plus' (succ K)) (val (pair M N) σ)).
```

Except for some uninteresting contortions to bind substitution variables and carry them through, the body of `plus'` corresponds to the standard recursive definition of addition, in continuation-passing style. Finally, we define a conversion function from `nat` to `int`, using an auxiliary continuation transformer `add1`:

```
add1 : tm (false int) -> tm (false int).
add1/n : body (add1 K) (n N) (λ_ throw K (val (n (s N)) snil)).

n2i : tm (false int) -> tm (false nat).
n2i/zz : body (n2i K) zz (λ_ throw K (val (n z) snil)).
n2i/ss : body (n2i K) (ss N) (λsub [σ] throw (n2i (add1 K)) (val N σ)).
```

At this point, it is worth checking again that all of our definitions are exhaustive and non-redundant:

```
%total P (body K P _).
%unique body +K +P -E.
```

Thankfully, this passes. If we had forgotten a case (e.g., if `n2i/zz` were commented out), the Twelf coverage checker would have flagged an exception.

Now, let's try running some programs. First, we try evaluating "2 + 2" using machine integers (note that `plus exit` denotes the `int*int`-continuation that adds its two arguments and then terminates with the result):

```
%query 1 *
  eval (throw (plus exit)
         (val (pair (n (s s z)) (n (s s z))) (sjoin snil snil)))
       R.
```

And indeed, the Twelf server answers `R = halt (s s s s z)`. Then we try evaluating "2 + 3" using the `nat` datatype (precomposing `exit` with the coercion `n2i`, to convert the result of `plus'` into an `int`):

```
%query 1 *
  eval (throw
         (plus' (n2i exit)) (val (pair (ss ss zz) (ss ss ss zz)) (sjoin snil snil)))
        R.
```

And indeed, `R = halt (s s s s s z)`.

To end this chapter, we will work up to a more significant example, along the lines of §4.4, building macros that let us view direct-style programs in call-by-value lambda-calculus (with strict products and sums) as syntactic sugar for $\mathcal{L}^+$ programs. (Or if you prefer, we are building a continuations semantics of CBV lambda-calculus via $\mathcal{L}^+$.) To begin, we define combinators for defunctionalized continuations, corresponding to the standard sequent-calculus left rules for products, sums, and negation:

```
fst : tm (false A) -> tm (false (A * B)).
fst/xy : body (fst K) (pair P1 P2) (λ, λsub [σ1] λsub [σ2] let σ1 E)
         <- body K P1 E.
snd : tm (false B) -> tm (false (A * B)).
snd/xy : body (snd K) (pair P1 P2) (λ, λsub [σ1] λsub [σ2] let σ2 E)
         <- body K P2 E.

case : tm (false A) -> tm (false B) -> tm (false (A + B)).
case/inl : body (case K1 K2) (inl P) E1
        <- body K1 P E1.
case/inr : body (case K1 K2) (inr P) E2
        <- body K2 P E2.

not : (tm (false A) -> tm #) -> tm (false ¬ A).
not/k : body (not K) hole (λcon [k] K k).
```

We also define some combinators that allow us to directly convert LF functions into continuations:

```
con : ({Δ} Δ ⊩ A -> tm (all Δ) -> tm #) -> tm (false A).
con/x : body (con K) P (λsub [σ] K _ P σ).

conV : (tm (true A) -> tm #) -> tm (false A).
conV/x : body (conV K) P (λsub [σ] K (val P σ)).

uncurry : (tm (false B) -> tm (false A)) -> tm (false (A * ¬ B)).
uncurry/pk : body (uncurry K) (pair P hole) (λ, λsub [σ1] λcon [k] throw (K k) (val P σ1)).
```

Note we will use `con` and `conV` to define continuations precisely when we *don't* need to do any pattern-matching on the argument. As usual, we can verify that these are proper definitions:

```
%total P (body K P _).
%unique body +K +P -E.
```

Now, as in §4.4, terms of CBV lambda-calculus will be represented by expressions with a free continuation variable. We define an LF type abbreviation `cmp A`, representing lambda terms computing type A:

```
%abbrev cmp : pos -> type = [A] tm (false A) -> tm #.
```

Then we define the canonical injection from values to computations:

```
%abbrev Lift : tm (true A) -> cmp A
   = [x] [k] throw k x.
```

The left-to-right pairing operation:

```
%abbrev
 Pair : cmp A -> cmp B -> cmp (A * B)
   = [e1] [e2] [k] e1 (con [_] [p1] [σ1] e2 (con [_] [p2] [σ2]
                   throw k (val (pair p1 p2) (sjoin σ1 σ2)))).
```

The first and second projections:

```
%abbrev
 Fst : cmp (A * B) -> cmp A
   = [e] [k] e (fst k).
%abbrev
 Snd : cmp (A * B) -> cmp B
   = [e] [k] e (snd k).
```

The left and right injections:

```
%abbrev
 Inl : cmp A -> cmp (A + B)
   = [e] [k] e (con [_] [p] [σ1] throw k (val (inl p) σ1)).
%abbrev
 Inr : cmp B -> cmp (A + B)
   = [e] [k] e (con [_] [p] [σ2] throw k (val (inr p) σ2)).
```

(Strict) case-analysis:

```
%abbrev
 Case : cmp (A + B) -> (tm (true A) -> cmp C) -> (tm (true B) -> cmp C) -> cmp C
   = [e] [f] [g] [k] e (case (conV [x] (f x) k) (conV [y] (g y) k)).
```

(Call-by-value) abstraction and application:

```
%abbrev
 Fn : (tm (true A) -> cmp B) -> cmp (A → B)
   = [f] Lift (val hole (shole (uncurry [k'] conV [x] (f x) k'))).

%abbrev
 App : cmp (A → B) -> cmp A -> cmp B
   = [f] [e] [k]
       f (not [kf]
       e (con [_] [p] [σ1] throw kf (val (pair p hole) (sjoin σ1 (shole k))))).
```

The natural numbers, represented internally by `int`:

```
%abbrev
 Z : cmp int
   = Lift (val (n z) snil).

%abbrev
 S : cmp int -> cmp int
   = [e] [k] e (add1 k).
%prefix 9 S.
```

Addition, with left-to-right evaluation order:

```
%abbrev
 Plus : cmp int -> cmp int -> cmp int
   = [e1] [e2] [k] e1 (conV [n1] e2 (conV [n2]
      (Pair (Lift n1) (Lift n2)) (plus k))).
```

(Note we could also give a slightly more efficient definition, with fewer administrative redexes:)

```
%abbrev
 Plus* : cmp int -> cmp int -> cmp int
   = [e1] [e2] [k] e1 (con [_] [n1] [σ1] e2 (con [_] [n2] [σ2]
      throw (plus k) (val (pair n1 n2) (sjoin σ1 σ2)))).
```

And finally, two effectful computations, which simply abort (with different results):

```
%abbrev
 Abort0 : cmp A
   = [k] ℧ z.
%abbrev
 Abort1 : cmp A
   = [k] ℧ (s z).
```

For convenience, we define a type abbreviation `run t n`, representing the fact that an `int`-computation terminates with result `n`:

```
%abbrev
  run  : cmp int -> i -> type = [t] [n] eval (t exit) (halt n).
```

We can now run direct-style programs directly using the operational semantics of $\mathcal{L}^+$. For example, Twelf answers the following query with `N = s s s s s z`:

```
%query 1 *
  run (Plus (S S Z) (S S S Z)) N.
```

It answers the following with `N = s z`:

```
%query 1 *
  run (Plus (S S Z) Abort1) N.
```

And the following with `N = z` (demonstrating left-to-right evaluation):

```
%query 1 *
  run (Plus Abort0 Abort1) N.
```

For the big finale, we evaluate the term representing "$(\lambda x.x + 1)\ 1$":

```
%query 1 *
  run (App (Fn [x] Plus (Lift x) (S Z)) (S Z)) N.
```

And Twelf answers `N = s s z`.

5.3 Related work

The idea of building finitary notation systems for infinite proofs goes back to the days of Hilbert's project and the ω-rule [Hilbert, 1931], particularly as developed by Schütte [1950] and Shoenfield [1959]. Infinite terms were also considered early in type theory by Tait [1965] (with a term constructor analogous to the ω-rule) and further explored by Martin-Löf [1972]. Since then, more systematic connections between infinitary and finitary proof/type theory have drawn on the work of Mints [1978], who explained how to view cut-elimination on infinite objects as a continuous operation that could be reflected back to finite objects [Mints, 2000, Buchholz, 1991, 1997, Schwichtenberg, 1998]. Note though that these approaches mainly take advantage of the idea of proofs/terms as infinitely *wide* objects (what I would call derivations for types with infinitely many patterns). The idea of proofs/terms as infinitely *deep* objects (what I would call derivations for types with a non-well-founded definition ordering) has also been taken seriously, drawing on the work of Aczel [1988] on non-well-founded sets. Coquand [1994], for example, has considered infinite objects in type theory defined through guarded recursive clauses. More recently, Brotherston and Simpson [2007] have introduced a sequent calculus of *cyclic proofs*, and explored its practical application in inductive theorem proving.

Chapter 6

Refinement types and completeness of subtyping

But why accept the counterexample? We proved our conjecture—now it is a theorem. I admit that it clashes with this so-called "counterexample". One of them has to give way. But why should the theorem give way, when it has been proved? It is the "criticism" that should retreat. It is fake criticism.... It is a *monster,* a pathological case, not a counterexample.

—DELTA, speaking in Imre Lakatos' *Proofs and Refutations*

6.1 Introduction

In Chapter 4 we built a language, $\mathcal{L}$, by placing our faith stubbornly in the proofs-as-programs correspondence—programs of $\mathcal{L}$ are literally derivations in polarized logic, and their semantics are literally defined by composition of derivations, a.k.a. cut-elimination. The real work in making this dogma practical was developing a type-free notation for $\mathcal{L}$ amenable to functional programmers. The notation we gave was hopefully largely familiar to ML or Haskell programmers—who program happily with pattern-matching and arbitrary mutual recursion—except for the novel twist of continuation patterns.

What does it mean that the notation is type-free? Because $\mathcal{L}$ programs are intrinsically typed, the type of a program is a promise about how the program will be structured. For example, the way we define a (negative) value of type $\mathbf{N} \to \uparrow\mathbf{N}$ is by presenting a map from $\mathbf{N}$-patterns to $\uparrow\mathbf{N}$-values. The type tells us we need not clutter our syntax by, say, considering $\mathbf{B}$-patterns. On the other hand, this is not telling us very much. There are potentially many other things we would like to know about a function on the natural numbers. For example, whether it is bounded, or linear, or monotonic. At the very least, we might like to know that it always terminates, or that it never aborts.

This is the motivation behind *refinement types.* Originally introduced by Freeman and Pfenning [1991] in the context of ML, the idea was to develop a second, more refined type system on top of standard Hindley-Milner inference, capable of expressing and automatically verifying more precise invariants of programs—but without changing the language itself. A succession of Pfenning's students pursued this line of research in their dissertations, including Tim Freeman

[1994], Hongwei Xi [1998], Rowan Davies [2005], and Joshua Dunfield [2007]. Along the way, an observation was made that became the original motivation for *this* thesis: as refinement type systems become more and more precise, they become sensitive to the evaluation order of the original language.

This was in some ways surprising, and in some ways unsurprising. Surprising because of examples like Haskell and ML, which use different evaluation order but share, at least to a first approximation, the same underlying type system. On the other hand, not completely surprising because of artifacts like the *value restriction*, specific to ML, which emerged after painful years of trying to understand the interaction of polymorphism with effects. In early compilers, programs like the following clearly senseless one passed the typechecker, with the variable `x` given polymorphic type `('a list) ref`:

```
let val x = ref []
in (x := ["hello world!\n"]; hd (!x) + 2)
end
```

This was thought to be an issue peculiar to mutable references, and various patches to typechecking were proposed. A non-exhaustive list of people who worked specifically on the problem of combining polymorphism with references includes MacQueen [1988], Tofte [1988], Leroy and Weis [1991], Wright [1992], Hoang, Mitchell, and Viswanathan [1993], Greiner [1993], and Talpin and Jouvelot [1994]. However the problem with polymorphism was more pervasive than originally realized—not confined to its interaction with references but with effects more generally—as Harper and Lillibridge [1991] exhibited by giving another unsound but (at the time) well-typed program, using SML/NJ's `callcc` feature.[1]

The value restriction, proposed by Wright [1995] and eventually adopted in the definition of Standard ML [Milner et al., 1997], ruled out all of these programs by limiting the situations in which polymorphic generalization could be applied: for `x` to be generalized in `let val x = e1 in e2 end`, `e1` must be a polymorphic *value.* This rules out the senseless program above, for example, because `ref []` is not a value (at run-time it evaluates to a fresh location storing the empty list). It is important to see, though, that the reason this restriction on typing is necessary is *not only* the presence of effects: it is also the eager semantics of the `let` construct. If instead, `e1` were substituted wholesale for `x`, without first evaluating it down to a value, then no value restriction would be necessary. For instance, under such semantics, the expression `hd (!x) + 2` in the above program would attempt to take the head of an empty list—which would be strange and raise an exception, but would not be unsound.

Given this background, it should then not have been too surprising when Davies and Pfenning [2000] discovered that the standard typing rules for intersection types—invented in the late '70s by Coppo and Dezani-Ciancaglini [1978] and independently by Sallé [1978]—were unsound for ML. After all, intersection types are a form of finitary polymorphism. Thus Davies and Pfenning found that a value restriction on intersection introduction was necessary. Somewhat unexpectedly, they also found that a standard rule of subtyping,

$$(A \to B) \cap (A \to C) \leq A \to (B \cap C)$$

was unsound for call-by-value functions in the presence of effects. The reason for this turns out to be precisely the same as for the value restriction, but might be more psychologically disturbing, since it seems to be a statement about what intersection types and function types *mean*, independently of the syntax of the language.

In an even more bizarre turn, Dunfield and Pfenning [2004] discovered an entirely new *evaluation context restriction* when investigating the theory of (untagged) union types in ML. Earlier, in an effect-free setting, Barbanera et al. [1995] had studied union types with this elimination rule:

$$\frac{\Gamma \vdash e : A \cup B \quad \Gamma, x : A \vdash e' : C \quad \Gamma, x : B \vdash e' : C}{\Gamma \vdash e'[e/x] : C}$$

The rule allows eliminating arbitrarily many occurrences of an expression e of union type, by discriminating the union. But Dunfield and Pfenning [2004] found this was unsound in the presence of effects, because the different occurrences of e could evaluate to a value of type A or B nondeterministically. Therefore they proposed the unusual "tridirectional" rule, schematized by evaluation contexts $E[]$:

$$\frac{\Gamma \vdash e : A \cup B \quad \Gamma, x : A \vdash E[x] : C \quad \Gamma, x : B \vdash E[x] : C}{\Gamma \vdash E[e] : C}$$

This rule only allows eliminating a single occurrence of e, in evaluation position. Although Dunfield and Pfenning did not mention it explicitly, the reasons for the evaluation context restriction also imply that standard laws $[(A \to C) \cap (B \to C) \leq (A \cup B) \to C$ and $\top \leq \perp\!\!\!\perp \to C]$ are unsound for call-by-*name* functions in the presence of effects (the latter even when the only effect is non-termination).

All of these restrictions were discovered by "disillusionment", so to speak, in the sense that the messy world of side-effects provided counterexamples to simple but naive rules. Yet, bitter experience is only a poor substitute for explanation, and understandably we may get the feeling that policies such as the value and evaluation context restrictions amount to ad hoc *monster-barring*. The central aim of this chapter is to explain why phenomena such as the value and evaluation context restrictions, as well as evaluation-order-dependent subtyping laws, need not be policies imposed after-the-fact, but instead can arise synthetically from a logical view of refinement typing. Our goal is not only to better understand existing choices, but to develop a theoretical framework that narrows the design space for future, more expressive type systems for effectful programming languages.

We will pay special attention to the question of subtyping, particularly through a view of subtyping I call the *identity coercion interpretation*. The idea of this interpretation is to equate subtyping relationships with typings of an identity coercion, i.e., whenever we assert $S \leq T$, we actually have a typing derivation showing that under the assumption that x has type S, the reconstructed value Id_x has type T. This idea is not entirely new—a more traditional way of expressing the identity coercion interpretation is that subtyping is witnessed by η-expansion, as was discussed for example by Brandt and Henglein [1998]—but I think it is underappreciated. It has several advantages over more typical definitions of subtyping (such as those based on an axiomatization, or on a value inclusion interpretation), including:

- The question of deciding subtyping relationships is reduced to the question of type checking. There is no need for a separate axiomatization, or a separate decision procedure.
- The subtyping relationship is sound by construction: it is no stronger than typing.

- Conversely, the identity coercion interpretation imposes a useful methodological constraint when designing a type system: it must be strong enough to derive all expected subtyping laws!

The last point is an important one in my opinion. For example, to respect the identity coercion interpretation, I believe (although I won't attempt to prove formally) that any type system for refining algebraic datatypes with intersections and unions must deal directly with pattern-matching in some form or another. Essentially, pattern-matching provides the witness to the expected distributivity properties of intersections and unions through sums and products (laws such as $(A \oplus B) \cap (A \oplus C) \leq A \oplus (B \cap C)$, etc.), which seems difficult or impossible to capture with natural deduction-style typing rules.

On the other hand, when a subtyping relationship fails by this interpretation, all we know is that the identity coercion fails to type check. In contrast, the counterexamples found by Davies and Pfenning [2000] and by Dunfield and Pfenning [2004] demonstrated that certain subtyping relationships are *unsound.* To better understand those counterexamples, we consider another notion of subtyping called the *no-counterexamples interpretation.* The idea here is that $S \leqslant T$ holds if any value of type S can be safely combined with a continuation accepting type T. The relationship $\leqslant$ is effectively the largest possible subtyping relationship we could hope for, while $\leq$ is the smallest. The question, naturally, is how close they come to each other. In §6.2.12, I give a "conditional" completeness theorem, which shows that the identity coercion interpretation in fact captures all safe subtyping relationships, assuming the language is equipped with enough effects (different forms of nondeterminism). We can then take this for what it is—we can either read it unconditionally by extending the language (although in general the effects are fairly high-powered), or we can learn to live with incompleteness.

6.2 Refining $\mathcal{L}^+_{\mho}$

As we did in Chapter 4, we will try to illustrate most of the general concepts with a simpler special case, the positive fragment $\mathcal{L}^+$ of $\mathcal{L}$. We include the impure expression $\mho$ from the beginning, because as we will see, it has a very interesting interpretation from the point of view of refinement typing.

6.2.1 A refinement "restriction"?

Freeman and Pfenning originally introduced refinements as a way of verifying additional properties of already well-typed programs, rather than for allowing new programs to be typed. This is the so-called *refinement restriction,* which is meant to be contrasted with the historical approach to intersection types. Indeed, one of the best-known results about intersection types is that if they are added to simply-typed λ-calculus, suddenly many more programs become typable: all strongly normalizing ones [Pottinger, 1980]. But there is another way of comparing these two approaches, in light of our discussion of intrinsic vs. extrinsic definitions (§4.1), and of typed vs. untyped languages (§4.3.4). The historical approach to intersections should not be thought of merely as an extension to simply-typed λ-calculus. Intersection type systems are always defined *extrinsically,* with Curry-style rules assigning types to terms of the untyped λ-calculus. So, we can view general intersection type systems as a special kind of refinement type system, subject to the "refinement restriction", where the underlying language is *very* coarsely typed. Another way of putting this is that "refinement" is really just a synonym for "extrinsic".

This is more than a play on words, I think. We saw in Chapter 4 that many properties of $\mathcal{L}^+$ and $\mathcal{L}$ (their operational semantics, the separation theorem, etc.) could be explained in an entirely generic way by reference to patterns, independently of types. The same will be true in this chapter. We define extrinsic refinements of intrinsic $\mathcal{L}$-types through a notion of *pattern-inversion*, and then provide generic, pattern-based rules for establishing when a program is well-refined. Thus, if the reader wishes, they can adapt all of our development to the historical tradition simply by starting with a different language of patterns: all value patterns introducing the universal positive type, and all continuation patterns introducing the universal negative type (cf. §4.3.4). A legitimate question is whether there is any point in having a fine-grained layer of intrinsic types, if we are only going to add an even finer layer of extrinsic types. I believe there is, and that having a fine-grained intrinsic layer makes it easier to reason about the extrinsic layer—but this is a somewhat subjective question that needs both mathematical and empirical evidence, outside the scope of this thesis.

6.2.2 Refining types

To help avoid confusion between intrinsic $\mathcal{L}$-types and extrinsic refinements of those types, we will follow Pfenning [2008] and call the latter *sorts*. To avoid inventing too many neologisms, though, we will still refer to the process of verifying sorts of terms as refinement typing, and the partial order between sorts as a subtyping relationship.

Notation (Sorts). *We write $S \sqsubseteq A$ to indicate that S is a sort of (the intrinsic $\mathcal{L}$-type) A.*

A sort $S \sqsubseteq A$ is *not* a subtype of A in the traditional sense. For example, if $S \sqsubseteq A$, then it will also be the case that $\neg S \sqsubseteq \neg A$, violating contravariance. Likewise, if $S \sqsubseteq A$, then the *assertion* S refines the assertion A, but so too does the *denial* $\bullet S$ refine the denial $\bullet A$.

In order to explain how to define individual sorts through patterns, we first need an analogous notion of refinement for frames:

Definition 6.2.1 (Frame refinements). *A* **frame refinement** $\Psi \sqsubseteq \Delta$ *is built inductively out of refutation refinements $\bullet S \sqsubseteq \bullet A$ (or annotated refinements $\kappa : \bullet S \sqsubseteq \kappa : \bullet A$) using concatenation and conjunction, as described by the following rules:*

$$\frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Psi_1 \sqsubseteq \Delta_1 \quad \Psi_2 \sqsubseteq \Delta_2}{\Psi_1, \Psi_2 \sqsubseteq \Delta_1, \Delta_2} \qquad \frac{}{\top \sqsubseteq \Delta} \qquad \frac{\Psi_1 \sqsubseteq \Delta \quad \Psi_2 \sqsubseteq \Delta}{\Psi_1 \wedge \Psi_2 \sqsubseteq \Delta}$$

For $\Delta_1 \in \Delta_2$, $\Psi_1 \sqsubseteq \Delta_1$ and $\Psi_2 \sqsubseteq \Delta_2$, we define the containment relation $\Psi_1 \in^{\Delta_2}_{\Delta_1} \Psi_2$ as follows:

$$\frac{}{\Psi \in^{\Delta}_{\Delta} \Psi} \qquad \frac{\Psi \in^{\Delta_1}_{\Delta} \Psi_1}{\Psi \in^{\Delta_1,\Delta_2}_{\Delta} \Psi_1, \Psi_2} \qquad \frac{\Psi \in^{\Delta_2}_{\Delta} \Psi_2}{\Psi \in^{\Delta_1,\Delta_2}_{\Delta} \Psi_1, \Psi_2} \qquad \frac{\Psi \in^{\Delta}_{\Delta} \Psi_1}{\Psi \in^{\Delta}_{\Delta} \Psi_1 \wedge \Psi_2} \qquad \frac{\Psi \in^{\Delta}_{\Delta} \Psi_2}{\Psi \in^{\Delta}_{\Delta} \Psi_1 \wedge \Psi_2}$$

We usually leave the intrinsic frame annotations implicit, writing simply $\Psi_1 \in \Psi_2$. In this relaxed notation, the containment relation rules read like so:

$$\frac{}{\Psi \in \Psi} \qquad \frac{\Psi \in \Psi_1}{\Psi \in \Psi_1, \Psi_2} \qquad \frac{\Psi \in \Psi_2}{\Psi \in \Psi_1, \Psi_2} \qquad \frac{\Psi \in \Psi_1}{\Psi \in \Psi_1 \wedge \Psi_2} \qquad \frac{\Psi \in \Psi_2}{\Psi \in \Psi_1 \wedge \Psi_2}$$

The frame refinement constructors give us many different ways of writing effectively the same thing. For example, if $S_1, T_1 \sqsubseteq A$ and $S_2, T_2 \sqsubseteq B$, we can refine the frame $\kappa_1 : \bullet A, \kappa_2 : \bullet B$ by both $\Psi_1 = (\kappa_1 : \bullet S_1, \kappa_2 : \bullet S_2) \wedge (\kappa_1 : \bullet T_1, \kappa_2 : \bullet T_2)$ and $\Psi_2 = (\kappa_1 : \bullet S_1 \wedge \kappa_1 : \bullet T_1), (\kappa_2 : \bullet S_2 \wedge \kappa_2 : \bullet T_2)$, but these really represent the same frame refinement. Formally, we mean this in the following sense:

Observation 6.2.2. *Any frame refinement $\Psi \sqsubseteq \Delta$ defines a map from continuation variables $\kappa : \bullet A \in \Delta$ to finite sets of annotations $\Psi(\kappa) = \{\bullet S \mid \kappa : \bullet S \in \Psi\}$, where all the $\bullet S \in \Psi(\kappa)$ refine A. Explicitly, $\Psi(\kappa)$ is defined as follows:*

$$
\begin{aligned}
(\kappa : \bullet S)(\kappa) &= \{\bullet S\} \\
(\Psi_1, \Psi_2)(\kappa) &= \begin{cases} \Psi_1(\kappa) & \textit{if } \kappa : \bullet A \in \Delta_1 \sqsupseteq \Psi_1 \\ \Psi_2(\kappa) & \textit{if } \kappa : \bullet A \in \Delta_2 \sqsupseteq \Psi_2 \end{cases} \\
\top(\kappa) &= \emptyset \\
(\Psi_1 \wedge \Psi_2)(\kappa) &= \Psi_1(\kappa) \cup \Psi_2(\kappa)
\end{aligned}
$$

So Ψ_1 and Ψ_2 are really the same frame refinement, in the sense that $\Psi_1(\kappa_1) = \Psi_2(\kappa_1) = \{\bullet S_1, \bullet T_1\}$ and $\Psi_1(\kappa_2) = \Psi_2(\kappa_2) = \{\bullet S_2, \bullet T_2\}$. The sets $\Psi(\kappa) = \{\bullet S_1, \ldots, \bullet S_n\}$ should be thought of conjunctively, as asserting that κ accepts each of the S_i. Beware, though, that because of the contravariant reading of $\bullet S$, this is the same as saying that κ accepts the union sort $S_1 \cup \cdots \cup S_n$.

Just as intrinsic types were defined by their patterns, extrinsic sorts are defined by *pattern-inversion.* To define the sort $S \sqsubseteq A$ one specifies, for every pattern $p :: (\Delta \Vdash A)$, a finite set $[\![p : S]\!]$ of frame refinements of Δ. Intuitively, the set $[\![p : S]\!]$ is interpreted *disjunctively,* as the different possible refinements of the continuation variables bound by the pattern, assuming it has the given sort. A few examples should give an intuition for the idea:

$$
\begin{aligned}
[\![(\kappa_1, \kappa_2) : \neg S \otimes \neg T]\!] &= \{(\kappa_1 : \bullet S, \kappa_2 : \bullet T)\} \\
[\![\kappa : \neg S \cap \neg T]\!] &= \{(\kappa : \bullet S \wedge \kappa : \bullet T)\} \\
[\![\kappa : \neg S \cup \neg T]\!] &= \{(\kappa : \bullet S), (\kappa : \bullet T)\} \\
[\![\mathsf{inl}\,\kappa_1 : \neg S \oplus \neg T]\!] &= \{(\kappa_1 : \bullet S)\} \\
[\![(\kappa_1, \kappa_2) : \neg S \otimes \bot]\!] &= \emptyset \\
[\![\kappa : \top]\!] &= \{\top\}
\end{aligned}
$$

These examples are calculated from the following general definitions, as the reader can verify:

Intersections and unions

$$
\begin{aligned}
[\![p : S \cap T]\!] &= \{(\Psi_1 \wedge \Psi_2) \mid \Psi_1 \in [\![p : S]\!], \Psi_2 \in [\![p : T]\!]\} \\
[\![p : S \cup T]\!] &= [\![p : S]\!] \cup [\![p : T]\!] \\
[\![p : \top]\!] &= \{\top\} \\
[\![p : \bot]\!] &= \emptyset
\end{aligned}
$$

Products, sums, negations

$$\frac{}{\top \sqsubseteq A} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq A}{S \cap T \sqsubseteq A}$$

$$\frac{}{\bot \sqsubseteq A} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq A}{S \cup T \sqsubseteq A}$$

$$\frac{S \sqsubseteq A}{\overset{+}{\neg} S \sqsubseteq \overset{+}{\neg} A} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq B}{S \otimes T \sqsubseteq A \otimes B} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq B}{S \oplus T \sqsubseteq A \oplus B}$$

Figure 6.1: Some sort constructors

$$\begin{aligned}
[\![\kappa : \overset{+}{\neg} S]\!] &= \{(\kappa : \bullet S)\} \\
[\![(p_1, p_2) : S \otimes T]\!] &= \{(\Psi_1, \Psi_2) \mid \Psi_1 \in [\![p_1 : S]\!], \Psi_2 \in [\![p_2 : T]\!]\} \\
[\![\mathsf{inl}\, p : S \oplus T]\!] &= [\![p : S]\!] \\
[\![\mathsf{inr}\, p : S \oplus T]\!] &= [\![p : T]\!]
\end{aligned}$$

In order for these general definitions of pattern-inversion to make sense, we have to respect some implicit conventions on the formation of sorts. For example, the binary union and intersection can only be formed when both S and T refine the same type (i.e., the "refinement restriction"), or else we might not be able to invert p at either S or T. These implicit conventions are described in Figure 6.1. Note, though, that the refinement restriction is trivially satisfied if we view all (positive) sorts as refining the same (positive) universal type. In addition to these generic refinement constructors, we can define some more interesting refinements of datatypes. Recall we defined booleans ($\mathbf{B}$), natural numbers ($\mathbf{N}$), and a paradoxical domain ($\mathbf{D} = \mathbf{N} \oplus \neg\mathbf{D}$) in §4.2.1. We can refine these types to isolate more interesting properties:

Sorts of booleans ($S \sqsubseteq \mathbf{B}$)

$$[\![\mathsf{tt} : \mathbf{T}]\!] = \{\cdot\} \qquad [\![\mathsf{tt} : \mathbf{F}]\!] = \emptyset$$

$$[\![\mathsf{ff} : \mathbf{T}]\!] = \emptyset \qquad [\![\mathsf{ff} : \mathbf{F}]\!] = \{\cdot\}$$

Sorts of natural numbers ($S \sqsubseteq \mathbf{N}$)

$$[\![\mathsf{z} : \mathbf{N}_e]\!] = \{\cdot\} \qquad [\![\mathsf{z} : \mathbf{N}_o]\!] = \emptyset \qquad [\![\mathsf{z} : \mathbf{N}_{nz}]\!] = \emptyset$$

$$[\![\mathsf{s}\, p : \mathbf{N}_e]\!] = [\![p : \mathbf{N}_o]\!] \qquad [\![\mathsf{s}\, p : \mathbf{N}_o]\!] = [\![p : \mathbf{N}_e]\!] \qquad [\![\mathsf{s}\, p : \mathbf{N}_{nz}]\!] = [\![p : \top]\!]$$

Sorts of the paradoxical domain ($S \sqsubseteq \mathbf{D}$)

$$[\![\mathsf{dn}\, p : \mathbf{D}_e]\!] = [\![p : \mathbf{N}_e]\!] \qquad [\![\mathsf{dn}\, p : \mathbf{D}_*]\!] = \emptyset$$

$$[\![\mathsf{dk}\, p : \mathbf{D}_e]\!] = [\![p : \overset{+}{\neg}\mathbf{D}_e]\!] \qquad [\![\mathsf{dk}\, p : \mathbf{D}_*]\!] = [\![p : \overset{+}{\neg}\top]\!]$$

These different sorts have intuitive descriptions:

- $\mathbf{T}$: the one-point subset of the booleans containing only "true"

- **F**: the one-point subset of the booleans containing only "false"
- $\mathbf{N}_e$, $\mathbf{N}_o$, $\mathbf{N}_{nz}$: respectively, the even, odd, and positive natural numbers
- $\mathbf{D}_e$: the subset of $\mathbf{D}$ obtained by restricting natural numbers to be even, and propagating the restriction through to continuations
- $\mathbf{D}_*$: the continuation subset of $\mathbf{D}$

It will be useful to isolate a degenerate case of pattern-inversion.

Definition 6.2.3 (Absurd patterns)**.** *We say that an A-pattern p is* **absurd** *at sort $S \sqsubseteq A$ if $[\![p : S]\!] = \emptyset$. For example* $\mathsf{s\,z}$ *is absurd at sort $\mathbf{N}_e$, and* $\mathsf{dn\,z}$ *is absurd at $\mathbf{D}_*$. If p is absurd at S, we say that S* **refutes** *p.*

6.2.3 Refining terms

Just as the terms of $\mathcal{L}^+$ were defined generically by reference to value patterns—without mentioning any particular types—refinement typing of $\mathcal{L}^+$ is defined generically by reference to value pattern-inversion. The refinement type system consists of four judgments:

$$\Xi \vdash V : S \qquad \Xi \vdash K : \bullet S \qquad \Xi \vdash \sigma : \Psi \qquad \Xi \vdash E : \checkmark$$

In general, the refinement typing judgment takes the form $\Xi \vdash t : \mathcal{J}$, where $t :: (\Gamma \vdash J)$ is an $\mathcal{L}^+$-term, $\mathcal{J}$ refines J, and Ξ refines the context Γ. Refinement of contexts is defined as you would expect:

$$\cdot \sqsubseteq \cdot \qquad\qquad \frac{\Xi \sqsubseteq \Gamma \quad \Psi \sqsubseteq \Delta}{\Xi, \Psi \sqsubseteq \Gamma, \Delta}$$

We write $\Psi_1 \in \Xi, \Psi_2$ if either $\Psi_1 \in \Xi$ or $\Psi_1 \in \Psi_2$. Again, any context refinement $\Xi \sqsubseteq \Gamma$ uniquely defines a map from continuation variables $\kappa : \bullet A \in \Gamma$ to sets of annotations $\Xi(\kappa) = \{\bullet S \mid \kappa : \bullet S \in \Xi\}$.

We now explain how to establish each of the refinement typing judgments.

- ($V : S$): A value $p[\sigma]$ has sort S if there is some Ψ in $[\![p : S]\!]$ such that σ satisfies all of Ψ:

  $$\frac{\Psi \in [\![p : S]\!] \quad \Xi \vdash \sigma : \Psi}{\Xi \vdash p[\sigma] : S}$$

  Note that implicit in this rule are the conditions:

  $$p :: (\Delta \Vdash A) \qquad \sigma :: (\Gamma \vdash \Delta)$$

  $$\Xi \sqsubseteq \Gamma \qquad \Psi \sqsubseteq \Delta \qquad S \sqsubseteq A$$

  But it is safe to leave these conditions implicit, since they are the only sensical way to interpret the rule (by our definition of patterns and pattern-inversion).

- ($K : \bullet S$): A continuation K accepts sort $S \sqsubseteq A$ if for every A-pattern p, in every possible context $\Psi \in [\![p : S]\!]$, the expression $K(p)$ is well-sorted:

$$\frac{\Psi \in [\![p : S]\!] \quad \longrightarrow \quad \Xi, \Psi \vdash K(p) : \checkmark}{\Xi \vdash K : \bullet S}$$

Again, this rule leaves implicit various conditions that are forced for the rule to make sense. Notice that the set $[\![p : S]\!]$ need not be a singleton (e.g., if S is a union), and so checking the continuation could involve checking the same branch $K(p)$ multiple times.

- ($\sigma : \Psi$): A substitution σ satisfies all of $\Psi \sqsubseteq \Delta$ if for every continuation variable $\kappa : \bullet A \in \Delta$, for every hypothesis $\bullet S \in \Psi(\kappa)$, the continuation $\sigma(\kappa)$ accepts sort S. We establish this with rules that follow the structure of Ψ:

$$\frac{}{\Xi \vdash \sigma : \top} \qquad \frac{\Xi \vdash \sigma : \Psi_1 \quad \Xi \vdash \sigma : \Psi_2}{\Xi \vdash \sigma : \Psi_1 \wedge \Psi_2} \qquad \frac{}{\Xi \vdash \cdot : \cdot} \qquad \frac{\Xi \vdash \sigma_1 : \Psi_1 \quad \Xi \vdash \sigma_2 : \Psi_2}{\Xi \vdash (\sigma_1, \sigma_2) : (\Psi_1, \Psi_2)}$$

Notice that when Ψ is a meet of two frame refinements $\Psi_1 \wedge \Psi_2$, we must check the same substitution—and ultimately the same set of continuations—against multiple sorts.

Proposition 6.2.4. *The following rule for type checking substitutions is sound and complete:*

$$\frac{\kappa : \bullet S \in \Psi \quad \longrightarrow \quad \Xi \vdash \sigma(\kappa) : \bullet S}{\Xi \vdash \sigma : \Psi}$$

- ($E : \checkmark$): An expression $\kappa\ V$ is well-sorted in context Ξ if there is at least some hypothesis $\bullet S \in \Xi(\kappa)$ such that V has sort S:

$$\frac{\kappa : \bullet S \in \Xi \quad \Xi \vdash V : S}{\Xi \vdash \kappa\ V : \checkmark}$$

For reference, we include the complete set of refinement typing rules for $\mathcal{L}^+$ in Figure 6.2. One note about interpreting these rules: Recall that in general we allow $\mathcal{L}^+$ continuations to be defined by partial recursive functions on patterns, and $\mathcal{L}^+$ terms to be built up using arbitrary mutual reference. A typing derivation for such a term mirrors its structure, i.e., can be non-well-founded. By this convention, the non-terminating pseudo-expression Ω is necessarily well-sorted. On the other hand, we have a choice as to how to treat the aborting expression $\mho$. We will find it far more interesting to declare that $\mho$ is *not* well-sorted. Thus, we can see refinement checking as a way of guaranteeing that a term does not make essential use of $\mho$ (even though it may mention it syntactically). Formally, we will eventually establish a variant of the famous slogan,

well-sorted programs don't go $\mho$

We consider some example $\mathcal{L}^+$ programs, old and new, to illustrate refinement typing.

Example 6.2.5. Recall the $(\mathbf{B} \otimes \mathbf{B}) \otimes \neg\mathbf{B}$-continuation *and** from Example 4.2.9:

$$\begin{aligned}
\textit{and}^* \ ((\mathsf{tt}, \mathsf{tt}), \kappa) &= \kappa\ \mathsf{TT} \\
\textit{and}^* \ ((\mathsf{tt}, \mathsf{ff}), \kappa) &= \kappa\ \mathsf{FF} \\
\textit{and}^* \ ((\mathsf{ff}, \mathsf{tt}), \kappa) &= \kappa\ \mathsf{FF} \\
\textit{and}^* \ ((\mathsf{ff}, \mathsf{ff}), \kappa) &= \kappa\ \mathsf{FF}
\end{aligned}$$

(See body of §6.2.2 and §6.2.3 for implicit refinement restrictions.)

Frame refinements

$\Psi \sqsubseteq \Delta \ ::= \kappa : \bullet S \mid \cdot \mid \Psi_1, \Psi_2 \mid \mathbb{T} \mid \Psi_1 \wedge \Psi_2$

Context refinements

$\Xi \sqsubseteq \Gamma \quad ::= \quad \cdot \mid \Xi, \Psi$

Refinement typing judgments

$\Xi \vdash V : S$	V has sort S
$\Xi \vdash K : \bullet S$	K accepts sort S
$\Xi \vdash \sigma : \Psi$	σ satisfies Ψ
$\Xi \vdash E : \checkmark$	E is well-sorted

$$\frac{\Psi \in [\![p : S]\!] \quad \Xi \vdash \sigma : \Psi}{\Xi \vdash p[\sigma] : S} \qquad \frac{\Psi \in [\![p : S]\!] \quad \longrightarrow \quad \Xi, \Psi \vdash K(p) : \checkmark}{\Xi \vdash K : \bullet S}$$

$$\frac{}{\Xi \vdash \sigma : \mathbb{T}} \qquad \frac{\Xi \vdash \sigma : \Psi_1 \quad \Xi \vdash \sigma : \Psi_2}{\Xi \vdash \sigma : \Psi_1 \wedge \Psi_2} \qquad \frac{}{\Xi \vdash \cdot : \cdot} \qquad \frac{\Xi \vdash \sigma_1 : \Psi_1 \quad \Xi \vdash \sigma_2 : \Psi_2}{\Xi \vdash (\sigma_1, \sigma_2) : (\Psi_1, \Psi_2)}$$

$$\frac{\bullet S \in \Xi(\kappa) \quad \Xi \vdash V : S}{\Xi \vdash \kappa\ V : \checkmark}$$

$$\Xi \vdash \Omega : \checkmark \qquad \Xi \nvdash \mho : \checkmark$$

Figure 6.2: Refinement typing of $\mathcal{L}_{\mho}^{+}$

We can assign *and** many precise refinement types. For example, *and** accepts sorts $(\mathbf{T}\otimes\mathbf{T})\otimes\neg\mathbf{T}$, $(\mathbf{F}\otimes\mathbf{F})\otimes\neg\mathbf{F}$, etc., but not $(\mathbf{T}\otimes\mathbf{T})\otimes\neg\mathbf{F}$, $(\mathbf{F}\otimes\mathbf{F})\otimes\neg\mathbf{T}$, etc. Indeed, we can fully capture the truth table by noting that *and** accepts the union sort

$$\begin{array}{rl} & (\mathbf{T}\otimes\mathbf{T})\otimes\neg\mathbf{T} \\ \cup & (\mathbf{T}\otimes\mathbf{F})\otimes\neg\mathbf{F} \\ \cup & (\mathbf{F}\otimes\mathbf{T})\otimes\neg\mathbf{F} \\ \cup & (\mathbf{F}\otimes\mathbf{F})\otimes\neg\mathbf{F} \end{array}$$

which can also be expressed slightly more concisely as:

$$(\mathbf{T}\otimes\mathbf{T})\otimes\neg\mathbf{T} \quad\cup\quad (\mathbf{F}\otimes\top)\otimes\neg\mathbf{F} \quad\cup\quad (\top\otimes\mathbf{F})\otimes\neg\mathbf{F}$$

(Warning: $\top \sqsubseteq \mathbf{B}$ is something very different from $\mathbf{T}$!) The reader can try working through the refinement typing derivations that establish these respective facts.

Saying that a continuation accepts a union is perhaps a bit unnatural. In the refinement type system for full $\mathcal{L}$, we can talk about the isomorphic definition of *and** as a negative value (of type $\mathbf{B}\otimes\mathbf{B}\to\uparrow\mathbf{B}$), and say that it has an intersection sort:

$$(\mathbf{T}\otimes\mathbf{T})\to\uparrow\mathbf{T} \quad\cap\quad (\mathbf{F}\otimes\top)\to\uparrow\mathbf{F} \quad\cap\quad (\top\otimes\mathbf{F})\to\uparrow\mathbf{F}$$

For now, the reader can take this as an intuition for understanding the sort of *and**. ■

Example 6.2.6. We can similarly refine the type of the $(\mathbf{N}\otimes\mathbf{N})\otimes\neg\mathbf{N}$-continuations *plus** and *times** from Example 4.2.11. For example, *plus** accepts the union sort

$$(\mathbf{N}_e\otimes\mathbf{N}_e)\otimes\neg\mathbf{N}_e \quad\cup\quad (\mathbf{N}_e\otimes\mathbf{N}_o)\otimes\neg\mathbf{N}_o \quad\cup\quad (\mathbf{N}_o\otimes\mathbf{N}_e)\otimes\neg\mathbf{N}_o \quad\cup\quad (\mathbf{N}_o\otimes\mathbf{N}_o)\otimes\neg\mathbf{N}_e$$

while *times** accepts

$$(\mathbf{N}_e\otimes\top)\otimes\neg\mathbf{N}_e \quad\cup\quad (\top\otimes\mathbf{N}_e)\otimes\neg\mathbf{N}_e \quad\cup\quad (\mathbf{N}_o\otimes\mathbf{N}_o)\otimes\neg\mathbf{N}_o$$

In contrast to the example above, these refinements of course do not come close to characterizing the behavior of *plus** and *times**. They are only a partial specification. ■

Example 6.2.7. Consider the following definition of a $\mathbf{D}$-continuation *app0*, which attempts to apply its argument to zero:

$$\begin{array}{rcl} \mathit{app0}\ (\mathsf{dn}\,\overline{n}) & = & \mho \\ \mathit{app0}\ (\mathsf{dk}\,\kappa) & = & \kappa\ (\mathsf{DN}\,\mathsf{Z}) \end{array}$$

A $\mathbf{D}$-value can be either a natural number or a continuation, and in the former case there is no sensible action for *app0* to take—and so its most sensible action is to simply abort. This style of definition is typical in ordinary programming: when the type of a function is too coarse to capture its intended use, an extra case is added raising an exception. This helps avoid the dreaded SML "Warning: match nonexhaustive" compiler message, but unfortunately, it gives no static guarantees that the function really *is* being used as intended, i.e., that the exception will never be raised. As such, this situation provides a classic application for refinement types.

In our case, we can indeed verify that *app0* accepts the refinement $\mathbf{D}_* \sqsubseteq \mathbf{D}$, because $\mathbf{D}_*$ rules out the $\mathsf{dn}\,\overline{n}$ branch. Observe that it will *not* accept $\mathbf{D}_e \sqsubseteq \mathbf{D}$, despite the fact that Z is even, because that refinement does not rule out the $\mathsf{dn}\,\overline{n}$ branch where the ill-sorted expression $\mho$ is executed. ■

6.2.4 Refining equality, identity and composition

In Chapter 4 we explained how to define syntactic equality for $\mathcal{L}^+$, how to compose two terms, and how to build terms representing the identities for these operations. We want to know that these notions behave reasonably with respect to refinement typing.

Proposition 6.2.8 (Refinement respects equality)**.** *If* $\Xi \vdash t : \mathcal{J}$ *and* $t = t'$ *then* $\Xi \vdash t' : \mathcal{J}$*.*

Proof. Immediate from the definition of the refinement typing rules. □

Proposition 6.2.9 (Identity preserves refinement)**.**

1. *If* $\kappa : \bullet S \in \Xi$ *then* $\Xi \vdash Id_\kappa : \bullet S$
2. *If* $\Psi \in \Xi$ *then* $\Xi \vdash Id_{[\Delta]} : \Psi$ *(where* $\Psi \sqsubseteq \Delta$*)*

Proposition 6.2.10 (Composition preserves refinement)**.**

1. *If* $\Xi \vdash K : \bullet S$ *and* $\Xi \vdash V : S$ *then* $\Xi \vdash K \bullet V : \checkmark$
2. *If* $\Xi \vdash \sigma : \Psi$ *and* $\Xi(\Psi) \vdash t : \mathcal{J}$ *then* $\Xi \vdash t[\sigma] : \mathcal{J}$

Proof. The proofs of these statements precisely mirror the construction of the identity and composition terms, respectively. Again, it is important that we allow typing derivations to be as non-well-founded as terms. □

Operationally, we will be more interested in the type preservation theorem for the environment semantics (i.e., n-ary composition), rather than the binary version above. Proposition 6.2.9 is likewise just a special case—reflexivity—of the identity coercion interpretation of subtyping. Nonetheless, these simple properties give us some confidence that the rules of refinement typing make sense.

6.2.5 Refining complex hypotheses

We can refine complex value hypotheses $x : A \in \Delta$ with hypotheses $x : S \in \Psi$. As with $\Psi(\kappa)$, the set $\Psi(x) = \{S_1, \ldots, S_n\}$ is interpreted conjunctively. Because of the *covariant* reading of the assertion S, this is the same as asserting an intersection $x : S_1 \cap \cdots \cap S_n$.

We type check a case-analysis on a value variable by applying pattern-inversion:

$$\frac{\Psi_1 \in [\![p : S_1]\!] \quad \cdots \quad \Psi_n \in [\![p : S_n]\!] \quad \longrightarrow \quad \Xi(\Psi_1 \wedge \cdots \wedge \Psi_n) \vdash t_p : \mathcal{J}}{\Xi(x : S_1 \wedge \cdots \wedge x : S_n) \vdash \mathsf{case}\ x\ \mathsf{of}\ p \mapsto t_p : \mathcal{J}}$$

It is important that we combine all the information from the different refinement assumptions about x, in order to have the strongest possible reasoning principle. For example, we can verify that $\Xi(x : \mathbf{N}_e \wedge x : \mathbf{N}_o) \vdash \text{case } x \text{ of } p \mapsto \mho : \checkmark$, because every $\mathbf{N}$-pattern $\overline{n}$ is either absurd at $\mathbf{N}_e$ or absurd at $\mathbf{N}_o$.

Again, we should check that the various operations on complex hypotheses respect refinement typing.

Proposition 6.2.11 (Pattern substitution preserves refinement). *If $\Xi(x : S_1 \wedge \cdots \wedge x : S_n) \vdash t : \mathcal{J}$ and $\Psi_1 \in [\![p : S_1]\!], \ldots, \Psi_n \in [\![p : S_n]\!]$ then $\Xi(\Psi_1 \wedge \cdots \wedge \Psi_n) \vdash t[p/x] : \mathcal{J}$.*

Proposition 6.2.12 (Value identity preserves refinement). *If $x : S \in \Xi$ then $\Xi \vdash Id_x : S$.*

Proof. Immediate. □

6.2.6 Subtyping: the identity coercion interpretation

In this section we introduce the identity coercion interpretation of subtyping and study some of its "internal" properties.

Definition 6.2.13 (Subtyping). *For $S, T \sqsubseteq A$, we say that S is a **subtype** of T ($S \leq_A T$) if there is a derivation of $\kappa : \bullet T \vdash Id_\kappa : \bullet S$. For $\Psi, \Psi' \sqsubseteq \Delta$, we say that Ψ is a **subframe** of Ψ' (written $\Psi \leq_\Delta \Psi'$) if there is a derivation of $\Psi \vdash Id_{[\Delta]} : \Psi'$.*

We often omit the subscript and write $S \leq T$, when the intrinsic type A can be inferred from context or is unimportant. We write $S \equiv T$ if both $S \leq T$ and $T \leq S$, and $S \not\leq T$ if there is no derivation of $S \leq T$. We adopt the analogous conventions for the subframe relationship.

Intuitively, $S \leq T$ says that we can uniformly convert any T-continuation into an S-continuation by precomposing it with the identity—hence the *identity coercion interpretation.* We could equivalently define subtyping in terms of a number of different kinds of identity coercions:

Proposition 6.2.14. *The following are equivalent:*

1. $S \leq T$
2. $x : S \vdash Id_x : T$
3. $x : S, \kappa : \bullet T \vdash \kappa\ Id_x : \checkmark$
4. *For all A-patterns p, for every $\Psi \in [\![p : S]\!]$ there exists $\Psi' \in [\![p : T]\!]$ such that $\Psi \leq \Psi'$*

Proof. Immediate by expanding the definition of the identity terms and the typing rules. □

Proposition 6.2.15. *Both $\leq_A$ and $\leq_\Delta$ are reflexive and transitive.*

Proof. These properties are direct results of the fact that identity and composition preserve refinement, combined with the unit laws (Lemma 4.2.15). For example, we can compose any two derivations $\kappa : \bullet S_2 \vdash Id_\kappa : \bullet S_1$ and $\kappa : \bullet S_3 \vdash Id_\kappa : \bullet S_2$ to obtain $\kappa : \bullet S_3 \vdash Id_\kappa[Id_\kappa] : \bullet S_1$. But $Id_\kappa[Id_\kappa] = Id_\kappa$, and so $\kappa : \bullet S_3 \vdash Id_\kappa : \bullet S_1$ (i.e., $S_1 \leq S_3$) because refinement typing respects definitional equality. □

Lemma 6.2.16 (Term inclusion/reverse inclusion).

- *If* $\Xi \vdash V : S$ *and* $S \leq T$ *then* $\Xi \vdash V : T$
- *If* $\Xi \vdash K : \bullet T$ *and* $S \leq T$ *then* $\Xi \vdash K : \bullet S$
- *If* $\Xi \vdash \sigma : \Psi$ *and* $\Psi \leq \Psi'$ *then* $\Xi \vdash \sigma : \Psi'$
- *If* $\Xi(\Psi') \vdash E : \checkmark$ *and* $\Psi \leq \Psi'$ *then* $\Xi(\Psi) \vdash E : \checkmark$

Proof. All immediate consequences of composition preservation and the unit laws. □

Proposition 6.2.17. $\leq_A$ *is a distributive lattice with meet/join operations* $\cap$, $\cup$, $\top$, $\bot$.

Proposition 6.2.18. $\oplus$,$\otimes$, *and* $\overset{+}{\neg}$ *obey the usual covariant and contravariant laws:*

1. *If* $S_1 \leq T_1$ *and* $S_2 \leq T_2$ *then* $S_1 \otimes S_2 \leq T_1 \otimes T_2$
2. $S_1 \oplus S_2 \leq T_1 \oplus T_2$ *iff* $S_1 \leq T_1$ *and* $S_2 \leq T_2$
3. $\overset{+}{\neg} S \leq \overset{+}{\neg} T$ *iff* $T \leq S$

Proposition 6.2.19. *Intersections and unions distribute through sums and products, as follows:*

1. $S \otimes (T_1 \cap T_2) \equiv (S \otimes T_1) \cap (S \otimes T_2)$
2. $S \otimes (T_1 \cup T_2) \equiv (S \otimes T_1) \cup (S \otimes T_2)$
3. $S \otimes \bot \equiv \bot$
4. $S \oplus (T_1 \cap T_2) \equiv (S \oplus T_1) \cap (S \oplus T_2)$
5. $S \oplus (T_1 \cup T_2) \equiv (S \oplus T_1) \cup (S \oplus T_2)$

Proof (of Props. 6.2.17–6.2.19). These are all trivial consequences of the pattern-inversion rules. □

It is worth pointing out that although these properties are typical, intuitive, and easy for us to derive, the fact that we can derive them *within the language* is not so typical. In particular, the reason why it is easy to derive distributivity laws such as $(S \oplus T_1) \cap (S \oplus T_2) \leq S \oplus (T_1 \cap T_2)$ is because the identity coercions are defined by pattern-matching and checked by pattern-inversion. With the standard λ-calculus introduction and elimination rules for sums and intersections, for example, we cannot derive $x : (S \oplus T_1) \cap (S \oplus T_2) \vdash \eta(x) : S \oplus (T_1 \cap T_2)$ (unless we already assume a prior notion of subtyping to coerce x to type $S \oplus (T_1 \cap T_2)$, but that is cheating). We can derive $x : (S \otimes T_1) \cap (S \otimes T_2) \vdash \eta(x) : S \otimes (T_1 \cap T_2)$ in the standard presentation of λ-calculus with products and intersections, but not when the projection elimination rules for products are replaced by the less standard but not uncommon splitting construct "let $(x, y) = e_1$ in e_2". In other words, if we want to define subtyping syntactically as we have, it is important that we have a good notion of syntax—and conversely, the identity coercion interpretation provides a test that we really do have a good notion.

In addition to these familiar distributivity properties of intersections and unions through products and sums, we can consider the the distributivity and *non*-distributivity properties of intersections/unions through positive negation.

Proposition 6.2.20.

1. *(i)* $\top \equiv \dot{\neg}\bot$ *and (ii)* $\dot{\neg}S \cap \dot{\neg}T \equiv \dot{\neg}(S \cup T)$

2. *(i)* $\dot{\neg}\top \not\leq \bot$ *and (ii)* $\dot{\neg}(S \cap T) \not\leq \dot{\neg}S \cup \dot{\neg}T$ *unless* $S \leq T$ *or* $T \leq S$

Proof. 1(i) reduces to checking $\top \vdash Id_\kappa : \bullet\bot$, which holds vacuously since $\bot$ refutes every pattern. 1(ii) reduces to checking $\kappa : \bullet S \wedge \kappa : \bullet T \vdash Id_\kappa : \bullet S \cup T$. Since $[\![p : S \cup T]\!] = [\![p : S]\!] \cup [\![p : T]\!]$, this reduces to checking that Id_κ accepts both S and T, and that holds by identity preservation.

2(i) fails because $[\![p : \bot]\!]$ is empty but $[\![p : \dot{\neg}\top]\!]$ is not. 2(ii) requires that either $\kappa : \bullet S \cap T \vdash Id_\kappa : \bullet S$ or $\kappa : \bullet S \cap T \vdash Id_\kappa : \bullet T$, but by definition these hold if and only if $S \leq T$ or $T \leq S$. □

These are perhaps the most interesting laws and non-laws, and we should try to develop an intuition for them. As one easy consequence, we derive laws and non-laws for double-negated sorts:

Corollary 6.2.21 (Laws of double-negation)**.**

1. $\dot{\neg}\dot{\neg}S \leq \dot{\neg}\dot{\neg}T$ *iff* $S \leq T$

2. $\dot{\neg}\dot{\neg}S \cap \dot{\neg}\dot{\neg}T \not\leq \dot{\neg}\dot{\neg}(S \cap T)$ *unless* $S \leq T$ *or* $T \leq S$

3. $\dot{\neg}\dot{\neg}(S \cup T) \not\leq \dot{\neg}\dot{\neg}S \cup \dot{\neg}\dot{\neg}T$ *unless* $S \leq T$ *or* $T \leq S$

Proof. (1) is immediate by Prop. 6.2.18(3), while (2) and (3) reduce to Prop. 6.2.20 as follows:

$$\dot{\neg}\dot{\neg}S \cap \dot{\neg}\dot{\neg}T \equiv \dot{\neg}(\dot{\neg}S \cup \dot{\neg}T) \not\leq \dot{\neg}\dot{\neg}(S \cap T)$$
$$\dot{\neg}\dot{\neg}(S \cup T) \equiv \dot{\neg}(\dot{\neg}S \cap \dot{\neg}T) \not\leq \dot{\neg}\dot{\neg}S \cup \dot{\neg}\dot{\neg}T$$

□

6.2.7 Subtyping: axiomatization

Although the identity coercion interpretation itself gives a generic axiomatization of subtyping—by reducing it to the rules of refinement typing—it is instructive to give a more direct axiomatization by expanding the typing rules.

We begin by unrolling the meaning of $S \leq T$ by one step:

$$\frac{\Psi \in [\![p : S]\!] \quad \longrightarrow \quad \Psi, \kappa : \bullet T \vdash \kappa\ (p[Id_{[p]}]) : \checkmark}{\kappa : \bullet T \vdash Id_\kappa : \bullet S}$$

and likewise unrolling the meaning of $\Psi \leq \Psi'$, using Proposition 6.2.4:

$$\frac{\bullet T \in \Psi'(\kappa) \quad \longrightarrow \quad \Psi \vdash Id_\kappa : \bullet T}{\Psi \vdash Id_\Delta : \Psi'}$$

Can we go further? Well, to derive $\Psi, \kappa : \bullet T \vdash \kappa\ (p[Id_{[p]}]) : \checkmark$, we first have to find some $\Psi' \in [\![p : T]\!]$, and then show that $\Psi, \kappa : \bullet T \vdash Id_{[p]} : \Psi'$. The hypothesis about κ is irrelevant, so this reduces to showing that for all κ' and $\kappa' : \bullet T' \in \Psi'$, we can derive $\Psi' \vdash Id_{\kappa'} : \bullet T'$. That is, the following inference is sound and complete:

$$\frac{\exists \Psi' \in [\![p : T]\!] \quad \forall \bullet T' \in \Psi'(\kappa') \quad \Psi \vdash Id_{\kappa'} : \bullet T'}{\Psi, \kappa : \bullet T \vdash \kappa\ (p[Id_{[p]}]) : \checkmark}$$

And what about deriving $\Psi \vdash Id_\kappa : \bullet T$? We can unroll the definitions to see that the following inference is sound and complete:

$$\frac{\forall \Psi' \in [\![p : T]\!] \quad \exists \bullet S \in \Psi(\kappa) \quad \Psi, \kappa : \bullet S \vdash \kappa\ (p[Id_{[p]}]) : \checkmark}{\Psi \vdash Id_\kappa : \bullet T}$$

In other words, we have succeeded in reducing these two auxiliary typing judgments to each other!

Theorem 6.2.22 (Subtyping axiomatization)**.** *We define two relations $\Psi \leq_p T$ and $S \leq_\kappa \Psi$, where*

$$p :: (\Delta \Vdash A) \qquad \kappa : \bullet B \in \Delta$$
$$T \sqsubseteq A \qquad \Psi \sqsubseteq \Delta \qquad S \sqsubseteq B$$

by the following coinductive rules:

$$\frac{\exists \Psi' \in [\![p : T]\!] \quad \forall \bullet S \in \Psi'(\kappa) \quad S \leq_\kappa \Psi}{\Psi \leq_p T} \qquad \frac{\forall \Psi' \in [\![p : S]\!] \quad \exists \bullet T \in \Psi(\kappa) \quad \Psi' \leq_p T}{S \leq_\kappa \Psi}$$

Then the subtyping/subframing relationships are soundly and completely axiomatized as follows:

$$\frac{\forall \Psi \in [\![p : S]\!] \quad \Psi \leq_p T}{S \leq T} \qquad \frac{\forall \bullet S \in \Psi'(\kappa) \quad S \leq_\kappa \Psi}{\Psi \leq \Psi'}$$

Proof. A restatement of the previous paragraph. □

6.2.8 Reconstructing the value/evaluation context restrictions

The failed distributivity laws for double-negation are closely related to the value and evaluation context restrictions discovered by Davies and Pfenning [2000] and by Dunfield and Pfenning [2004]. We can see this pretty directly if we recall (§4.4) that one way of understanding arbitrary terms of call-by-value λ-calculus is as $\mathcal{L}^+$ expressions with a distinguished continuation variable $\kappa : \bullet A$. Such expressions are in one-to-one correspondence with $\mathcal{L}^+$ values of double-negated type $\neg\neg A$. Suppose then that we have a single value that can be assigned two double-negated sorts, $V : \neg\neg S$ and $V : \neg\neg T$, where $S, T \sqsubseteq A$. We can validly assign it an intersection, $V : \neg\neg S \cap \neg\neg T$. But we cannot go on to conclude $V : \neg\neg(S \cap T)$, because of the failure of the principle $\neg\neg S \cap \neg\neg T \leq \neg\neg(S \cap T)$. On the other hand, there is no value restriction on union introduction, because from either $V : \neg\neg S$ or $V : \neg\neg T$ we can correctly conclude $V : \neg\neg(S \cup T)$, applying the valid subtyping law $\neg\neg S \cup \neg\neg T \leq \neg\neg(S \cup T)$.

Dually, an arbitrary (not necessarily evaluation) context for an ML term can be seen as an $\mathcal{L}^+$ continuation *accepting* a double-negated type. Suppose we have a single continuation accepting two double-negated sorts, $K : \bullet\neg\neg S$ and $K : \bullet\neg\neg T$. This immediately implies that it accepts the union $K : \bullet\neg\neg S \cup \neg\neg T$, but this does *not* imply $K : \bullet\neg\neg(S \cup T)$, because of the failure of $\neg\neg(S \cup T) \leq \neg\neg S \cup \neg\neg T$. On the other hand, from either $K : \bullet\neg\neg S$ or $K : \bullet\neg\neg T$ we can correctly conclude $K : \bullet\neg\neg(S \cap T)$, so there is no restriction on intersection elimination.

It is also possible to understand the value restriction on its own, without resorting to subtyping, by looking directly at potential typing rules for expressions.

Notation. *We write* $\Xi \vdash E \div S$ *as an abbreviation for* $\Xi, \kappa : \bullet S \vdash E : \checkmark$. *(Note that the typing judgment implicitly binds* κ*. If we want to be more explicit, we can write* $\Xi \vdash \kappa.E \div S$*.)*

Consider the following rule for giving an expression an intersection type:

$$\frac{\Xi \vdash E \div S \quad \Xi \vdash E \div T}{\Xi \vdash E \div S \cap T}\,*$$

We write "$*$" because the rule is *not* admissible. If it were, we would show this by examining every potential use of the continuation variable κ in E, and establishing that the use is well-sorted under assumption that $\kappa : \bullet S \cap T$. Well, suppose there is a use $\kappa\ V$. From the first premise of the rule, we know that $V : S$ in a context containing at least $\kappa : \bullet S$ (and Ξ, and possibly other assumptions introduced during the course of the refinement typing derivation). From the second premise, we know that $V : T$ in a context containing at least $\kappa : \bullet T$. But this does *not* justify concluding that $V : S \cap T$ in a context containing $\kappa : \bullet S \cap T$ (and whatever other relevant hypotheses). One potentially useful observation we can make here, though, is that this step *would* be justified if the expression were *linear* (or affine), because then κ could not occur in V.

We can give a similar direct explanation of the evaluation context restriction, but it requires the refinement type system for full $\mathcal{L}$.

6.2.9 The environment semantics and type safety

In this section we consider the traditional (extrinsic) type safety properties for $\mathcal{L}^+$, with respect to its operational semantics and refinement type system. The reader may want to review §4.2.12 to recall the definition of the environment semantics.

Definition 6.2.23 (Environment typing)**.** *Let* γ *be a* Γ*-environment. For any context refinement* $\Xi \sqsubseteq \Gamma$*, we check that* $\gamma : \Xi$ *as follows:*

$$\frac{}{\mathsf{emp} : \cdot} \qquad \frac{\gamma : \Xi \quad \Xi \vdash \sigma : \Psi}{\mathsf{bind}(\gamma, \sigma) : (\Xi, \Psi)}$$

Proposition 6.2.24. *If* $\gamma : \Xi$ *and* $\bullet S \in \Xi(\kappa)$ *then* $\Xi \vdash \mathsf{lookup}(\gamma, \kappa) : \bullet S$*.*

Proof. Immediate (the extrinsic analogue of Proposition 4.2.26). □

Definition 6.2.25 (Program typing)**.** *The program* $\langle \gamma \mid E \rangle$ *is well-sorted if there is some* Ξ *such that* $\gamma : \Xi$ *and* $\Xi \vdash E : \checkmark$*. The program* $\langle \gamma \mid K \mid V \rangle$ *is well-sorted if there are some* Ξ *and* S *such that* $\gamma : \Xi$ *and* $\Xi \vdash K : \bullet S$ *and and* $\Xi \vdash V : S$*.*

We can now state and prove the progress and preservation lemmas in their usual, extrinsic form.

Lemma 6.2.26 (Progress)**.** *If* P *is a well-sorted, there exists a* P' *such that* $P \leadsto P'$*.*

Proof. Immediate, as in the proof of Lemma 4.2.30, with well-sortedness taking the place of the purity assumption. □

Lemma 6.2.27 (Preservation)**.** *If* P *is well-sorted and* $P \leadsto P'$ *then* P' *is well-sorted.*

Proof. The transition is either by lookup^+ or $\mathrm{bind/call}^+$. If the former, we know that $P = \langle \gamma \mid \kappa\ V \rangle$ and $P' = \langle \gamma \mid \mathsf{lookup}(\gamma, \kappa) \mid V \rangle$. From the assumption that P is well-sorted, there exists a context refinement Ξ such that $\gamma : \Xi$ and $\Xi \vdash \kappa\ V : \checkmark$. Inverting the latter typing derivation, there must exist $\bullet S \in \Xi(\kappa)$ such that $\Xi \vdash V : S$. By Proposition 6.2.24, $\Xi \vdash \mathsf{lookup}(\gamma, \kappa) : \bullet S$. Hence P' is well-sorted.

If the transition is by $\mathrm{bind/call}^+$, we know that $P = \langle \gamma \mid K \mid p[\sigma] \rangle$ and $P' = \langle \mathsf{bind}(\gamma, \sigma) \mid K(p) \rangle$. From the assumption that P is well-sorted, there exists a context refinement Ξ such that $\gamma : \Xi$ and sort S such that $\Xi \vdash K : \bullet S$ and $\Xi \vdash p[\sigma] : S$. Inverting the value typing derivation, there must exist $\Psi \in [\![p : S]\!]$ such that $\Xi \vdash \sigma : \Psi$, which implies $\mathsf{bind}(\gamma, \sigma) : (\Xi, \Psi)$. Inverting the continuation typing derivation, we know that $\Xi, \Psi \vdash K(p) : \checkmark$. Hence P' is is well-sorted. $\square$

Corollary 6.2.28 (Type safety). *If P is well-sorted, then $P \Downarrow R$ for some well-sorted R.*

Because $\mho$ is ill-sorted by convention, we can now confidently declare,

well-sorted programs don't go $\mho$

6.2.10 Subtyping: the no-counterexamples interpretation

We saw that the identity coercion interpretation of subtyping gives a synthetic reconstruction of the value and evaluation context restrictions for intersections and unions. On the other hand, observing that the unrestricted laws are not derivable is not the same as saying they are *unsafe*, which was in fact what Davies and Pfenning [2000] and Dunfield and Pfenning [2004] noticed about unrestricted intersection introduction and union elimination. In this section, we consider another possible interpretation of subtyping, aiming to deal directly with these safety violations.

Definition 6.2.29 (Safety). *Let $E :: (\Gamma, \Delta \vdash \#)$ and $\sigma :: (\Gamma \vdash \Delta)$. For any Γ-environment γ, we say that $\gamma \models E \perp \sigma$ (E and σ are* **safe for** *each other in γ) if $\langle \mathsf{bind}(\gamma, \sigma) \mid E \rangle \not\Downarrow \mho$. We write $\gamma \models V \perp K$ if $\langle \gamma \mid K \mid V \rangle \not\Downarrow \mho$. We write $E \perp \sigma$ and $V \perp K$ if these relationships hold relative to* emp.

Safety is an *orthogonality* relation, in the sense of Melliès and Vouillon [2005]. It is interesting in its own right and we will describe some of its properties later, but for now we are content to use it to give another interpretation of subtyping.[2]

Definition 6.2.30 (Safe subtyping). *Let $\Psi_1, \Psi_2 \sqsubseteq \Delta$. We say that Ψ_1 is a* **safe subframe** *of Ψ_2 (written $\Psi_1 \leqslant \Psi_2$) if for all substitutions $\sigma :: (\cdot \vdash \Delta)$ and expressions $E :: (\Delta \vdash \#)$,*

$$\text{if } \cdot \vdash \sigma : \Psi_1 \text{ and } \Psi_2 \vdash E : \checkmark \text{ then } E \perp \sigma$$

For two sorts $S, T \sqsubseteq A$ of a positive type, we say that S is a **safe subtype** *of T (written $S \leqslant T$) if for all values $V :: (\cdot \vdash A)$ and continuations $K :: (\cdot \vdash \bullet A)$,*

$$\text{if } \cdot \vdash V : S \text{ and } \cdot \vdash K : \bullet T \text{ then } V \perp K$$

We call this the *no-counterexamples interpretation* of subtyping. Clearly, if we have an explicit witness to the safety of a subtyping relationship using the identity coercion, then there can be no counterexamples.

[2]Beware that our relation $E \perp \sigma$ is in fact the precise dual of that of Girard [2001], who defines orthogonality as normalization *towards* (rather than away from) $\mho$. The idea of defining an orthogonality relation by safety rather than termination comes from Melliès and Vouillon, inspired by Krivine-style realizability [Danos and Krivine, 2000, Krivine, 2001].

Theorem 6.2.31 (Soundness). *$\Psi_1 \leq \Psi_2$ implies $\Psi_1 \leqslant \Psi_2$, and $S \leq T$ implies $S \leqslant T$*

Proof. Let $\cdot \vdash \sigma : \Psi_1$. $\Psi_2 \vdash E : \checkmark$. By Prop. 6.2.16, we can use the witness $\Psi_1 \leq \Psi_2$ either to coerce the substitution to $\cdot \vdash \sigma : \Psi_2$, or the expression to $\Psi_1 \vdash E : \checkmark$. In either case, we have a well-sorted program $\langle \mathsf{bind}(\mathsf{emp}, \sigma) \mid E \rangle$, which cannot evaluate to $\mho$ by the type safety theorem. We reason similarly to show that $S \leq T$ implies $S \leqslant T$. $\square$

The really interesting question is completeness: if the identity coercion does not check, can we come up with an explicit safety violation, i.e.,

$$\textit{does } S \not\leq T \textit{ imply } S \not\leqslant T\textit{, and } \Psi \not\leq \Psi' \textit{ imply } \Psi \not\leqslant \Psi'\textit{?}$$

This question does not have a completely straightforward answer, however. The identity coercion interpretation is fixed by the pure, logical fragment of $\mathcal{L}^+$, but the no-counterexamples interpretation depends on the set of non-logical effects we have available. Indeed even just to define $\leqslant$, we needed the "unsafe" aborting expression $\mho$, as well as at least one other "safe" observable result (Ω in our case) for $\leqslant$ to be non-trivial.

In this respect, the relationship between $\leq$ and $\leqslant$ is reminiscent of the relationship between definitional equality and observational equivalence that we already explored in Chapter 4. We could hope to settle the completeness question similarly, by including an additional effect in $\mathcal{L}^+$ sufficient for building counterexamples to any invalid subtyping laws—just as we used ground input to observationally distinguish any two syntactically distinct $\mathcal{L}^+$ expressions (§4.2.15). Such an approach works technically, but morally, these effects are a little strange.

Essentially, we need two kinds of nondeterminism: demonic and angelic. Demonic nondeterminism, written $E_1 \curlywedge E_2$, is the more mundane: the expression $E_1 \curlywedge E_2$ can step to either E_1 or E_2, leaving the choice up to a maximally malicious environment—and so $E_1 \curlywedge E_2$ is safe in an environment only if both E_1 and E_2 are. This is how we often model nondeterminism in a programming language, because we believe in Murphy's Law, and want to know that a program will stay afloat even if everything that can possibly go wrong does go wrong. Angelic nondeterminism, written $E_1 \curlyvee E_2$, is the more mystical: an expression $E_1 \curlyvee E_2$ can again step to either E_1 or E_2, but now the choice is made by a *benevolent* environment, meaning that $E_1 \curlyvee E_2$ is safe if *either* E_1 or E_2 are. This kind of angelic nondeterminism for type safety is more difficult to accept in a programming language, although (in this binary version) it makes sense from a purely computational standpoint.

So, adding both $\curlywedge$ and $\curlyvee$ to $\mathcal{L}^+$ and considering the completeness question settled seems in poor taste. I *will* show that this is really true, i.e., that having these operations is sufficient for generating subtyping counterexamples. However, rather than taking them as generic primitives, we can consider them as *properties of particular types.* And here I really do mean intrinsic types, rather than their extrinsic refinements. A frame Δ defines a particular collection of expressions, and we can think of the operations $\curlywedge$ or $\curlyvee$ as making more or less sense when restricted to this particular collection. In some cases, we can define $\curlywedge$ or $\curlyvee$ for a frame in terms of the operations on smaller frames (in the sense of the definition ordering)—as a trivial example, in $\mathcal{L}^+$ we can always define both operations for the empty frame (which only has the expressions Ω and $\mho$) by:

$$E \curlywedge \mho = \mho \quad E \curlywedge \Omega = E \qquad E \curlyvee \mho = E \quad E \curlyvee \Omega = \Omega$$

Therefore we can consider a more refined but open-ended version of the completeness question:

under reasonable computational assumptions, for which types/frames,
does $S \not\leq_A T$ imply $S \not\leqslant_A T$, and $\Psi \not\leq_\Delta \Psi'$ imply $\Psi \not\leqslant_\Delta \Psi'$?

6.2.11 Some counterexample examples (and counterexamples)

Before considering the general version of the completeness question, I want to better illustrate the nature of the question with some specific examples of counterexamples, and some seeming counterexamples to the existence of counterexamples.

Often, when $S \leq T$ fails for "obvious" reasons, it is easy to come up with counterexamples to $S \leqslant T$. For instance, it is trivial that $\mathbf{T} \not\leq \mathbf{F}$, and we can easily come up with a counterexample:

$$V = \mathsf{TT} \qquad \begin{array}{l} K\ \mathsf{tt} = \mho \\ K\ \mathsf{ff} = \Omega \end{array}$$

We have $V : \mathbf{T}$ and $K : \bullet\mathbf{F}$, but $V \not\perp K$. Therefore:

Proposition 6.2.32. $\mathbf{T} \not\leqslant \mathbf{F}$

The really interesting completeness questions have to do with the non-distributivity of intersections through negation (Proposition 6.2.20). In the 0-ary case $\overset{\pm}{\neg}\Top \not\leq \Bot$, we can always come up with a counterexample by pairing the continuation which diverges on all inputs, treated as a value, together with the continuation that ignores its argument and aborts. That is, we take

$$V = (p \mapsto \Omega) \qquad K\ \kappa = \mho$$

(Really we should be writing $\_[(p \mapsto \Omega)]$ for the value, but the meaning of this shorthand is hopefully clear.) Note that whatever the type $\overset{\pm}{\neg}A$ the sorts refine, we have $V : \overset{\pm}{\neg}\Top$ and $K : \bullet\Bot$ and $V \not\perp K$. Hence,

Proposition 6.2.33. *For all positive types A, $\neg\Top \not\leqslant_{\neg A} \Bot$.*

But what about the binary case, $\overset{\pm}{\neg}(S \cap T) \not\leq \overset{\pm}{\neg}S \cup \overset{\pm}{\neg}T$? Let's consider it at almost the simplest possible type, $\neg\mathbf{B}$, where already there is a bit of subtlety. Take $S = \mathbf{T}$ and $T = \mathbf{F}$. Can we come up with a counterexample to show that $\neg(\mathbf{T} \cap \mathbf{F}) \not\leqslant \neg\mathbf{T} \cup \neg\mathbf{F}$?

It is easy to come up with a counterexample to the *value inclusion*. Take the continuation that aborts on any boolean, treated as a $\neg\mathbf{B}$-value:

$$V = (b \mapsto \mho)$$

We have that $V : \neg(\mathbf{T} \cap \mathbf{F})$ (because $\mathbf{T} \cap \mathbf{F} \equiv \Bot$ refutes every $\mathbf{B}$-pattern) but neither $V : \neg\mathbf{T}$ nor $V : \neg\mathbf{F}$. However, failure of value inclusion is not sufficient for producing a counterexample to *safety*—we also need to give a continuation K accepting sort $\neg\mathbf{T} \cup \neg\mathbf{F}$ that is unsafe for V. What would such a continuation look like?

Well, K takes a $\mathbf{B}$-continuation variable κ as an argument, and does something with it. But what can it do? Because K accepts sort $\neg\mathbf{T} \cup \neg\mathbf{F}$, the body of the continuation $K(\kappa)$ must be well-sorted whether we assume $\kappa : \bullet\mathbf{T}$ or $\kappa : \bullet\mathbf{F}$. But if we ever try to pass a boolean value to κ, it must be either TT or FF, and hence the application will be ill-sorted under one of these assumptions. Basically, we see that in our language—$\mathcal{L}^+$ with Ω and $\mho$, and no other additional effects—the *only* continuation accepting sort $\neg\mathbf{T} \cup \neg\mathbf{F}$ is the one that ignores its argument and diverges:

$$K \ \kappa = \Omega$$

Yet, this K also accepts $\neg(\mathbf{T} \cap \mathbf{F})$, and indeed is safe for V. Thus, we are led to our first failure of completeness:

Proposition 6.2.34. *In $\mathcal{L}^{+}_{\mho}$ (with no additional effects), $\neg(\mathbf{T} \cap \mathbf{F}) \leqslant \neg\mathbf{T} \cup \neg\mathbf{F}$.*

Is there any way to recover completeness, by introducing a new effect? Yes, this is a situation where the angelic choice operator $E_1 \curlyvee E_2$ can come to the rescue.

> I admire your perverted ingenuity in inventing one definition after another as barricades against the falsification of your pet ideas.
>
> —Alpha, speaking to Delta in Imre Lakatos' *Proofs and Refutations*

Definition 6.2.35 (Orthogonal). *The safety relation induces an operation $(-)^{\perp}$ on sets of terms $\overline{t}$, called the* **orthogonal***:*

$$\overline{\sigma}^{\perp} = \{E \mid \forall \sigma \in \overline{\sigma} \,.\, E \perp \sigma\} \qquad \overline{E}^{\perp} = \left\{\sigma \mid \forall E \in \overline{E} \,.\, E \perp \sigma\right\}$$
$$\overline{V}^{\perp} = \left\{K \mid \forall V \in \overline{V} \,.\, V \perp K\right\} \qquad \overline{K}^{\perp} = \left\{V \mid \forall K \in \overline{K} \,.\, V \perp K\right\}$$

We write $t^{\perp}$ for the corresponding operation on the singleton set.

Definition 6.2.36 (Choice operators). *A frame Δ is said to have (binary)* **demonic choice** *if for every pair of expressions $E_1, E_2 :: (\Delta \vdash \#)$, there is an expression $E_1 \curlywedge E_2 :: (\Delta \vdash \#)$, such that $(E_1 \curlywedge E_2)^{\perp} = E_1^{\perp} \cap E_2^{\perp}$, admitting the following typing rule for all $\Psi \sqsubseteq \Delta$:*

$$\frac{\Psi \vdash E_1 : \checkmark \quad \Psi \vdash E_2 : \checkmark}{\Psi \vdash E_1 \curlywedge E_2 : \checkmark}$$

It is said to have binary **angelic choice** *if for every pair E_1, E_2 there is an expression $E_1 \curlyvee E_2$ such that $(E_1 \curlyvee E_2)^{\perp} = E_1^{\perp} \cup E_2^{\perp}$, admitting the following typing rules for all $\Psi \sqsubseteq \Delta$:*

$$\frac{\Psi \vdash E_1 : \checkmark}{\Psi \vdash E_1 \curlyvee E_2 : \checkmark} \qquad \frac{\Psi \vdash E_2 : \checkmark}{\Psi \vdash E_1 \curlyvee E_2 : \checkmark}$$

We say that a type A has [angelic/demonic] choice if the singleton frame $(\kappa : \bullet A)$ does. Since these properties hold relative to a language (an extension of $\mathcal{L}^{+}_{\mho}$), we speak of a language having [angelic/demonic] Δ-choice or A-choice.

In other words, expanding the definition of the orthogonal, for any substitution σ, we have $E_1 \curlywedge E_2 \perp \sigma$ iff both $E_1 \perp \sigma$ and $E_2 \perp \sigma$, while $E_1 \curlyvee E_2 \perp \sigma$ iff either $E_1 \perp \sigma$ or $E_2 \perp \sigma$. Observe that binary choice implies *finite* choice, because Ω and $\mho$ provide the units of the operations:

Proposition 6.2.37. *We can build the demonic/angelic choice of finitely many expressions by*

$$\bigcurlywedge_{i=k}^{n} E_i = \begin{cases} \Omega & n < k \\ E_k \curlywedge \left(\bigcurlywedge_{i=k+1}^{n} E_i\right) & k \leq n \end{cases}$$
$$\bigcurlyvee_{i=k}^{n} E_i = \begin{cases} \mho & n < k \\ E_k \curlyvee \left(\bigcurlyvee_{i=k+1}^{n} E_i\right) & k \leq n \end{cases}$$

Note that the demonic choice of finitely many expressions is well-sorted if all of the expressions are, and the angelic choice is well-sorted if at least one of the expressions is.

In fact, binary *demonic* choice implies *countable* demonic choice, essentially because failure is finite—but we will have no need for this below.[3] Note also that $\curlywedge$ and $\curlyvee$ are, by definition, associative, commutative, and idempotent with respect to safety.[4]

Proposition 6.2.38. *In any extension of $\mathcal{L}^+_{\mho}$ with angelic* $\mathbf{B}$*-choice,* $\neg(\mathbf{T} \cap \mathbf{F}) \not\leq \neg\mathbf{T} \cup \neg\mathbf{F}$*.*

Proof. We pair the value $V = (b \mapsto \mho)$ with the continuation $K\ \kappa = \kappa\ \mathsf{TT} \curlyvee \kappa\ \mathsf{FF}$. Note that $V : \neg(\mathbf{T} \cap \mathbf{F})$ and $K : \bullet\neg\mathbf{T} \cup \neg\mathbf{F}$, but $V \not\perp K$. □

Does this make sense? Perhaps. We might think of the angelic choice operator $E_1 \curlyvee E_2$ as first executing E_1 in a "sandbox", and if anything goes wrong, throwing the computation away and executing E_2. Thus this program will try passing TT to $(b \mapsto \mho)$, fail, and then try passing FF to $(b \mapsto \mho)$—but that also fails. Or we might really interpret the angelic choice as being implemented by our guardian angel, who tries as hard as they can to find a safe expression (which is in this case impossible). Whether or not either of these interpretations correspond to a realistic operational semantics, in any case let us suspend disbelief for now.

We can build a more interesting counterexample by considering the invalid law at higher type. Take $S = \neg\mathbf{T}$ and $T = \neg\mathbf{F}$, noting (by the *valid* distributivity law for positive negation) that $\neg\mathbf{T} \cap \neg\mathbf{F} \equiv \neg(\mathbf{T} \cup \mathbf{F})$. How do we show $\neg\neg(\mathbf{T} \cup \mathbf{F}) \not\leq \neg\neg\mathbf{T} \cup \neg\neg\mathbf{F}$?

First, we have to come up with a value of sort $\neg\neg(\mathbf{T} \cup \mathbf{F})$. If we try to do this within the $\mathcal{L}^+_{\mho}$ fragment, we quickly realize that any such value will also have one of the sorts $\neg\neg\mathbf{T}$ or $\neg\neg\mathbf{F}$, and so cannot play a role in a safety violation. However, using *demonic* $\mathbf{B}$-choice, we have a nice candidate:

$$V = (\kappa \mapsto \kappa\ \mathsf{TT} \curlywedge \kappa\ \mathsf{FF})$$

We can think of V as a coin flip, i.e., as a suspended boolean computation that nondeterministically returns true or false. Note that $V : \neg\neg(\mathbf{T} \cup \mathbf{F})$, but neither $V : \neg\neg\mathbf{T}$ nor $V : \neg\neg\mathbf{F}$. What about the matching continuation? By analogy from the previous example, it is not difficult to come up with the following continuation:

$$K\ \kappa^* = \kappa^*\ K_{\mathsf{tt}} \curlyvee \kappa^*\ K_{\mathsf{ff}}$$

where K_{tt} and K_{ff} are defined by

$$\begin{array}{ll} K_{\mathsf{tt}}\ \mathsf{tt} = \Omega & \qquad K_{\mathsf{ff}}\ \mathsf{tt} = \mho \\ K_{\mathsf{tt}}\ \mathsf{ff} = \mho & \qquad K_{\mathsf{ff}}\ \mathsf{ff} = \Omega \end{array}$$

It is easy to verify that $K : \bullet\neg\neg\mathbf{T} \cup \neg\neg\mathbf{F}$ (but not $K : \bullet\neg\neg(\mathbf{T} \cup \mathbf{F})$), and that $V \not\perp K$. But what does this program actually *do?*

It turns out that we don't actually have to *postulate* the angelic $\neg\mathbf{B}$-choice operation, because we can already *implement* it within $\mathcal{L}^+_{\mho}$. Consider the following alternate definition of K:

$$K\ \kappa^* = \kappa^*\ (b \mapsto \kappa^*\ K_b)$$

[3] Because divergence is safe and aborting is unsafe, it is the demonic choice (rather than the angelic) that is analogous to the usual "parallel-or" construct: $E_1 \curlywedge E_2$ is unsafe (i.e., terminates with failure) if either E_1 or E_2 is unsafe.

[4] $\curlywedge$ and $\curlyvee$ can be seen as lattice operations for the *opposite* of the approximation ordering $E \leq E'$ defined in §4.2.14. The flip is because "less defined" coincides with "more safe".

which can be expanded out to

$$K\ \kappa^* = \kappa^* \begin{pmatrix} \mathsf{tt} \mapsto \kappa^* \begin{pmatrix} \mathsf{tt} \mapsto \Omega \\ \mathsf{ff} \mapsto \mho \end{pmatrix} \\ \mathsf{ff} \mapsto \kappa^* \begin{pmatrix} \mathsf{tt} \mapsto \mho \\ \mathsf{ff} \mapsto \Omega \end{pmatrix} \end{pmatrix}$$

Again, the reader can verify that $K : \bullet\neg\neg\mathbf{T} \cup \neg\neg\mathbf{F}$ (but not $K : \bullet\neg\neg(\mathbf{T} \cup \mathbf{F})$), and that $V \not\perp K$. And now it is clear what is going on: K evaluates its argument twice (i.e., passes it a $\mathbf{B}$-continuation), and checks that it returns the same value each time. Hence K can go wrong when paired with the value V, which sometimes returns TT and sometimes FF.

Proposition 6.2.39. *In any extension of $\mathcal{L}^+_\mho$ with demonic $\mathbf{B}$-choice, $\neg\neg(\mathbf{T} \cup \mathbf{F}) \not\leq \neg\neg\mathbf{T} \cup \neg\neg\mathbf{F}$.*

6.2.12 The (conditional) completeness of the identity coercion interpretation

Let us try to collect some of the previous observations into a more systematic answer to the subtyping completeness question.

Recall (from §2.1.2) that the relation $\prec$ is defined as the transitive closure of the following clauses:

$$\frac{\kappa : \bullet A \in \Delta}{A \prec \Delta} \qquad \frac{p :: (\Delta \Vdash B)}{\Delta \prec B}$$

And again, recall that the types/frames below a type/frame in the $\prec$ relation are called its *ancestors.*

Definition 6.2.40 (Ancestral properties)**.** *We speak of a property being* **ancestral** *for a type/frame, if it holds for all of its ancestors (though not necessarily for itself). In particular, we say that a type/frame has* **ancestral choice** *if all of its ancestor frames have both angelic and demonic choice.*

Notation. *If A has ancestral choice, we can define choice operators for A-continuations by the following maps:*

$$(K_1 \curlywedge K_2)(p) = K_1(p) \curlywedge K_2(p) \qquad (K_1 \curlyvee K_2)(p) = K_2(p) \curlyvee K_2(p)$$

Similarly, if Δ has ancestral choice, we can define choice operators for Δ-substitutions by the following maps:

$$(\sigma_1 \curlywedge \sigma_2)(\kappa) = \sigma_1(\kappa) \curlywedge \sigma_2(\kappa) \qquad (\sigma_1 \curlyvee \sigma_2)(\kappa) = \sigma_2(\kappa) \curlyvee \sigma_2(\kappa)$$

Proposition 6.2.41. *The derived choice operators for continuations satisfy the following orthogonality conditions:*

$$(K_1 \curlywedge K_2)^\perp = K_1^\perp \cap K_2^\perp \qquad (K_1 \curlyvee K_2)^\perp = K_1^\perp \cup K_2^\perp$$

and admit the following refinement typing rules:

$$\frac{\cdot \vdash K_1 : \bullet S \quad \cdot \vdash K_2 : \bullet S}{\cdot \vdash K_1 \curlywedge K_2 : \bullet S} \qquad \frac{\cdot \vdash K_1 : \bullet S}{\cdot \vdash K_1 \curlyvee K_2 : \bullet S} \quad \frac{\cdot \vdash K_2 : \bullet S}{\cdot \vdash K_1 \curlyvee K_2 : \bullet S}$$

Proof. Immediate from the definition of the derived choice operators and the refinement typing rules for continuations. □

Proposition 6.2.42. *The derived demonic choice operator for substitutions satisfies:*

$$(\sigma_1 \curlywedge \sigma_2)^\perp \subseteq \sigma_1^\perp \cap \sigma_2^\perp$$

and admits the following typing rule:

$$\frac{\cdot \vdash \sigma_1 : \Psi \quad \cdot \vdash \sigma_2 : \Psi}{\cdot \vdash \sigma_1 \curlywedge \sigma_2 : \Psi}$$

Proof. The orthogonality conditions express that if an expression is safe for a demonic choice of substitutions then it is safe for each substitution. This is clear because the demonic choice can always mimic the behavior of either substitution during an execution. Note however that the converse does not necessarily hold—by analogy, the empty demonic choice (the substitution $\kappa \mapsto p \mapsto \Omega$) is not safe for every expression (in particular it is unsafe for $\mho$).

The admissibility of the typing rule follows immediately from the definition of the demonic choice operator and the derived typing rule for substitutions (Prop. 6.2.4). □

In order to simplify the statement and proof of the completeness theorem, we make the following observation:

Observation 6.2.43 (Finitary polymorphism)**.** *For all sorts $S \sqsubseteq A$ and A-patterns p, the set $[\![p : S]\!]$ is finite. Likewise, for all frame refinements $\Psi \sqsubseteq \Delta$ and variables $\kappa : \bullet A \in \Delta$, the set $\Psi(\kappa)$ is finite.*

This is a property of our type system, although one could easily imagine an extension with infinitary polymorphism that breaks it—then we would have to deal explicitly with countable choice operators. We now state the main lemma:

Notation. *For any sort $S \sqsubseteq A$, we write $[\![S]\!]_V$ for the set of values (of type A) $[\![S]\!]_V = \{V \mid \cdot \vdash V : S\}$ and $[\![S]\!]_K$ for the set of continuations (accepting A) $[\![S]\!]_K = \{K \mid \cdot \vdash K : \bullet S\}$. For any frame refinement $\Psi \sqsubseteq \Delta$, we write $[\![\Psi]\!]_E$ for the set of expressions (in Δ) $[\![\Psi]\!]_E = \{E \mid \Psi \vdash E : \checkmark\}$ and $[\![\Psi]\!]_\sigma$ for the set of substitutions (for Δ) $[\![\Psi]\!]_\sigma = \{\sigma \mid \cdot \vdash \sigma : \Psi\}$.*

Lemma 6.2.44. *Let $S \sqsubseteq A$, $\Psi \sqsubseteq \Delta$, and $T \sqsubseteq B$. Suppose Δ has ancestral choice. Then for any $p :: (\Delta \Vdash B)$ and $\kappa : \bullet A \in \Delta$:*

1. *If $\Psi \not\leq_p T$ then there is some $K \in [\![T]\!]_K$ and $\sigma \in [\![\Psi]\!]_\sigma$ such that $p[\sigma] \not\perp K$.*
2. *If $S \not\leq_\kappa \Psi$ then there is some value $V \in [\![S]\!]_V$ and substitution $\sigma \in [\![\Psi]\!]_\sigma$, such that $V \not\perp \sigma(\kappa)$*

Proof. Recall the axiomatization of $\Psi \leq_p T$ and $S \leq_\kappa \Psi$ (Theorem 6.2.22):

$$\frac{\exists \Psi' \in [\![p : T]\!] \quad \forall \bullet S \in \Psi'(\kappa) \quad S \leq_\kappa \Psi}{\Psi \leq_p T} \qquad \frac{\forall \Psi' \in [\![p : S]\!] \quad \exists \bullet T \in \Psi(\kappa) \quad \Psi' \leq_p T}{S \leq_\kappa \Psi}$$

Inverting these rules, we construct the counterexamples by recursion on the definition ordering:

1. From the assumption $\Psi \not\leq_p T$, we have that for all $\Psi'_i \in [\![p : T]\!]$, there is some continuation variable κ_i and $\bullet S_i \in \Psi'_i(\kappa_i)$ such that $S_i \not\leq_{\kappa_i} \Psi$. By applying (2) recursively (on $S_i \prec \Psi'_i \prec T$), we obtain a (finite) set of values $V_i \in [\![S_i]\!]_V$ and substitutions $\sigma_i \in [\![\Psi]\!]_\sigma$ such that $V_i \not\perp \sigma_i(\kappa_i)$. Now define $K \in [\![T]\!]_K$ by setting $K\ p = \curlyvee_i (\kappa_i\ V_i)$, and $\sigma \in [\![\Psi]\!]_\sigma$ by setting $\sigma\ \kappa_i = \curlywedge_j \sigma_j(\kappa_i)$ (and setting $K\ p'$ and $\sigma\ \kappa'$ to arbitrary type safe expressions/continuations everywhere else).

First, we can verify that K and σ really have the indicated types. In particular, for K we must check that for all $\Psi'_i \in [\![p : T]\!]$, the expression $\curlyvee_i(\kappa_i\ V_i)$ is well-typed in Ψ'_i, and this holds because in particular $\kappa_i\ V_i$ is well-typed. Likewise, for σ we must check that for all $\bullet S \in \Psi(\kappa_i)$, $\curlywedge_j \sigma_j(\kappa_i) \in [\![S]\!]_K$, and this holds because $\sigma_j(\kappa_i) \in [\![S]\!]_K$ for all j.

Second, we can verify these provide a safety violation:

$$\begin{array}{rll} p[\sigma] \perp K & \text{iff} & \curlyvee_i(\kappa_i\ V_i) \perp \sigma \\ & \text{iff} & \exists i.\kappa_i\ V_i \perp \sigma \\ & \text{iff} & \exists i.V_i \perp \curlywedge_j \sigma_j(\kappa_i) \\ & \text{iff} & \exists i.\forall j.V_i \perp \sigma_j(\kappa_i) \end{array}$$

but $\kappa_i\ V_i \not\perp \sigma_i$ for all i, so $p[\sigma] \not\perp K$.

2. From the assumption $S \not\leq_\kappa \Psi$, we know there is some value pattern p and $\Psi' \in [\![p : S]\!]$, such that for all $\bullet T_i \in \Psi(\kappa)$, we have $\Psi' \not\leq_p T_i$. By applying (1) recursively (on $\Psi' \prec S$) we obtain a (finite) set of continuations $K_i \in [\![T_i]\!]_K$ and substitutions $\sigma_i \in [\![\Psi']\!]_\sigma$ such that $p[\sigma_i] \not\perp K_i$. Now define $V \in [\![S]\!]_V$ by $V = p[\curlywedge_j \sigma_j]$ and $\sigma \in [\![\Psi]\!]_\sigma$ by $\sigma\ \kappa = \curlyvee_i K_i$.

   First we verify the types. To show $V \in [\![S]\!]_V$, we need that $\curlywedge_j \sigma_j \in [\![\Psi']\!]_\sigma$, and this holds because $\sigma_j \in [\![\Psi']\!]_\sigma$ for all j. To show $\sigma \in [\![\Psi]\!]_\sigma$, we need that for all $\bullet T_j \in \Psi(\kappa)$, $\curlyvee_i K_i \in [\![T_j]\!]_K$, and this holds because in particular $K_j \in [\![T_j]\!]_K$.

   Second, we verify the safety violation:

$$\begin{array}{rll} V \perp \sigma(\kappa) & \text{iff} & p[\curlywedge_j \sigma_j] \perp \curlyvee_i K_i \\ & \text{iff} & \exists i.p[\curlywedge_j \sigma_j] \perp K_i \\ & \text{implies} & \exists i.\forall j\ .p[\sigma_j] \perp K_i \end{array}$$

   but $p[\sigma_i] \not\perp K_i$ for all i, so $V \not\perp \sigma(\kappa)$.

□

Theorem 6.2.45 (Conditional completeness of subtyping)**.**

1. *If A has ancestral choice, then $S \not\leq_A T$ implies $S \not\lessapprox_A T$.*

2. *If Δ has ancestral choice, then $\Psi \not\leq_\Delta \Psi'$ implies $\Psi \not\lessapprox_\Delta \Psi'$.*

Proof. Recall that the following rules are complete for the subtyping/subframing relationships:

$$\frac{\forall \Psi \in [\![p : S]\!] \quad \Psi \leq_p T}{S \leq T} \qquad \frac{\forall \bullet S \in \Psi'(\kappa) \quad S \leq_\kappa \Psi}{\Psi \leq \Psi'}$$

Inverting these rules, we construct counterexamples to subtyping by applying the previous lemma.

1. There exists a pattern $p :: (\Delta \Vdash A)$, $\Psi \sqsubseteq \Delta \in [\![p : S]\!]$, some continuation $K \in [\![T]\!]_K$ and substitution $\sigma \in [\![\Psi]\!]_\sigma$, such that $p[\sigma] \not\perp K$. Since $p[\sigma] \in [\![S]\!]_V$, this provides the desired counterexample.

2. There exists a variable $\kappa : \bullet A \in \Delta$, $\bullet S \sqsubseteq A \in \Psi'(\kappa)$, some value $V \in [\![S]\!]_V$ and substitution $\sigma \in [\![\Psi]\!]_\sigma$, such that $V \not\perp \sigma(\kappa)$. Then $\kappa\ V \not\perp \sigma$ provides the desired counterexample.

□

After we have a conditional completeness theorem, naturally the next question is whether we can make it less conditional. Of course, the simplest answer is "Yes", if we are willing to take the choice operators as primitives in our language.

Theorem 6.2.46. *In $\mathcal{L}^+_\mho$ extended with angelic and demonic choice operators at generic frames, the identity coercion interpretation is sound and complete for the no-counterexamples interpretation.*

Proof. Immediate corollary of the soundness theorem (6.2.31) and the conditional completeness theorem. □

On the other hand, we saw in the previous section that there are situations where we can already build a choice operation at a particular frame out of choice operations at smaller frames. Can we retain completeness while sufficing with fewer primitives? And what happens when we relax the assumption of finitary polymorphism? The general answer to these questions is beyond our scope here. In a sense, the choice operations are "topological properties", and so fully answering these questions about subtyping requires developing a topological interpretation of $\mathcal{L}$. I think that is a very worthwhile project, but will not pursue it here.

6.3 Refining full $\mathcal{L}$

By now, we have been through enough iterations of dualizing that it is probably a better exercise for the reader to work out the rules of refinement typing for negative types on their own, rather than reading through more definitions. For reference, however, we include the rules of refinement typing for full $\mathcal{L}$ in Figure 6.3, as well as the definition of negative intersections/unions and some mixed polarity refinement constructors in Figure 6.4.

6.4 Related Work

I gave a brief history of ML typing and of refinement types at the start of the chapter. Intersection and union types are also considered in ludics, and in fact Girard [2001] sent out "an invitation to revisit the extant approaches to subtyping in the light of ludics". In some ways this work can be seen as answering that invitation, although it originated from the practical motivation of better understanding the strange sort of operationally-sensitive typing phenomena uncovered by Davies and Pfenning [2000] and Dunfield and Pfenning [2004]. The question of subtyping completeness investigated in §6.2.10–§6.2.12 is an instance of the more general question of completeness in proof theory, i.e., the relationship between proofs and countermodels. That question of course has been studied by many people. It seems very likely that these questions can be fruitfully stated in more topological terms, particularly along the lines of Paul Taylor's Abstract Stone Duality and Martín Escardó's synthetic topology [Taylor, 2002, Escardó, 2004].

Frame refinements
$\Psi \sqsubseteq \Delta \;::=\; \kappa : \bullet S \mid x : S \mid \cdot \mid \Psi_1, \Psi_2 \mid \top \mid \Psi_1 \wedge \Psi_2$
Context refinements
$\Xi \sqsubseteq \Gamma \;::=\; \cdot \mid \Xi, \Psi$

Refinement typing judgments

$\Xi \vdash V : S$	V has sort S
$\Xi \vdash K : \bullet S$	K accepts sort S
$\Xi \vdash \sigma : \Psi$	σ satisfies Ψ
$\Xi \vdash E : \checkmark$	E is well-sorted

$\boxed{\text{Value and continuation typing}}$

$$\frac{\Psi \in [\![p : S^+]\!] \quad \Xi \vdash \sigma : \Psi}{\Xi \vdash p[\sigma] : S^+} \qquad \frac{\Psi \in [\![p : S^+]\!] \quad \longrightarrow \quad \Xi, \Psi \vdash K(p) : \checkmark}{\Xi \vdash K : \bullet S^+}$$

$$\frac{\Psi \in [\![d : \bullet S^-]\!] \quad \longrightarrow \quad \Xi, \Psi \vdash V(d) : \checkmark}{\Xi \vdash V : S^-} \qquad \frac{\Psi \in [\![d : \bullet S^-]\!] \quad \Xi \vdash \sigma : \Psi}{\Xi \vdash d[\sigma] : \bullet S^-}$$

$\boxed{\text{Complex hypotheses}}$

$$\frac{\Psi_1 \in [\![p : S_1^+]\!] \quad \cdots \quad \Psi_n \in [\![p : S_n^+]\!] \quad \longrightarrow \quad \Xi(\Psi_1 \wedge \cdots \wedge \Psi_n) \vdash t_p : \mathcal{J}}{\Xi(x : S_1^+ \wedge \cdots \wedge x : S_n^+) \vdash \text{case } x \text{ of } p \mapsto t_p : \mathcal{J}}$$

$$\frac{\Psi_1 \in [\![d : \bullet S_1^-]\!] \quad \cdots \quad \Psi_n \in [\![d : \bullet S_n^-]\!] \quad \longrightarrow \quad \Xi(\Psi_1 \wedge \cdots \wedge \Psi_n) \vdash t_d : \mathcal{J}}{\Xi(\kappa : \bullet S_1^- \wedge \cdots \wedge \kappa : \bullet S_n^-) \vdash \text{case } \kappa \text{ of } d \mapsto t_d : \mathcal{J}}$$

$\boxed{\text{Substitution typing}}$

$$\frac{}{\Xi \vdash \sigma : \top} \qquad \frac{\Xi \vdash \sigma : \Psi_1 \quad \Xi \vdash \sigma : \Psi_2}{\Xi \vdash \sigma : \Psi_1 \wedge \Psi_2} \qquad \frac{}{\Xi \vdash \cdot : \cdot} \qquad \frac{\Xi \vdash \sigma_1 : \Psi_1 \quad \Xi \vdash \sigma_2 : \Psi_2}{\Xi \vdash (\sigma_1, \sigma_2) : (\Psi_1, \Psi_2)}$$

$\boxed{\text{Expression typing}}$

$$\frac{\bullet S^+ \in \Xi(\kappa) \quad \Xi \vdash V : S^+}{\Xi \vdash \kappa\, V : \checkmark} \qquad \frac{S^- \in \Xi(x) \quad \Xi \vdash K : \bullet S^-}{\Xi \vdash x\, K : \checkmark}$$

$$\Xi \vdash \Omega : \checkmark \qquad \Xi \nvdash \mho : \checkmark$$

Figure 6.3: Refinement typing of $\mathcal{L}$

$$
\begin{array}{rcl}
[\![d : \bullet S \cup T]\!] & = & \{(\Psi_1 \wedge \Psi_2) \mid \Psi_1 \in [\![d : \bullet S]\!], \Psi_2 \in [\![d : \bullet T]\!]\} \\
[\![d : \bullet S \cap T]\!] & = & [\![d : \bullet S]\!] \cup [\![d : \bullet T]\!] \\
[\![d : \bullet \bot\!\!\!\bot]\!] & = & \{\top\!\!\!\top\} \\
[\![d : \bullet \top\!\!\!\top]\!] & = & \emptyset \\
\\
[\![x : \downarrow S^-]\!] & = & \{(x : S^-)\} \\
[\![\kappa : \bullet \uparrow S^+]\!] & = & \{(\kappa : \bullet S^+)\} \\
[\![p@d : \bullet S^+ \to T^-]\!] & = & \{(\Psi_1, \Psi_2) \mid \Psi_1 \in [\![p : S^+]\!], \Psi_2 \in [\![d : \bullet T^-]\!]\}
\end{array}
$$

Figure 6.4: Definition of negative intersections/unions and some mixed polarity refinement constructors

Chapter 7

Conclusion

We have given a new take on the proofs-as-programs analogy, that, I hope, gives it new force. By analyzing the structure of proofs and refutations in terms of patterns, we have designed a propositions-as-types interpretation that:

- Accounts for features in modern functional programming languages, such as evaluation order and pattern-matching.
- Includes the ability to mix evaluation strategies, and to define types by either their constructors or destructors.
- Elegantly accounts for untyped computation, intrinsic types, and extrinsic refinement types.
- Is in many ways *easier* to reason about meta-theoretically than standard typed λ-calculus, by using techniques from infinitary proof theory.

Here we have only scratched the surface, though. Important theoretical work that needs to be done is to extend our approach to the rest of type theory, including but not limited to parametric polymorphism, modalities, module systems, and dependent types. The latter, in particular, we used in Chapter 5 to give embeddings of our language into two existing languages—it seems only fair that we should be able to account for dependent types on our own, and give a meta-circular interpreter for $\mathcal{L}$ in $\mathcal{L}$.[1] And polarization and focusing would hopefully shed light on some of the open problems related to these features, in particular the treatment of equality, and how to account for effects.

Another important project that remains to be carried out is to apply our theoretical framework towards the construction of a real working compiler for a functional language, with a rich type theory including intersections and unions. Ideally, we could rely on proof-theoretic principles as guides at each stage of the compiler pipeline. We took a first step towards this goal with the embeddings of Chapter 5, particularly the Twelf embedding, which made the syntax of the language first-order via defunctionalization. It seems, moreover, that there is a link to be drawn with the recent work initiated by Danvy [Danvy, 2003, Ager et al., 2003, Danvy and Millikin, 2006, Danvy, 2008], which attempts to build systematic connections between definitional interpretors

[1]A restricted formulation of dependent types in polarized type theory has already been developed by Licata and Harper [2009], and should be sufficient for defining this meta-circular interpreter.

and abstract machines, by way of systematic program transformations such as CPS translation, closure-conversion, and defunctionalization.

At a more basic level, though, we have to evaluate whether the type theory is sufficiently general for reasoning about effects at a fine enough level of granularity. The language $\mathcal{L}$ intrinsically enforces continuation-passing-style, and that means that effects are always sequentialized, and the type system always sound. But it could also mean that sometimes programs are *over*-sequentialized and the type system *too* conservative, when no effects are present. Modalities for enforcing purity/linearity would no doubt be helpful. But it may also be necessary to move to a more general abstraction such as *delimited* continuations [Danvy and Filinski, 1990]. Hopefully, even in a more general framework, the basic concepts of polarity and focalization should remain illuminating.

Bibliography

Peter Aczel. *Non-well founded sets*, volume 14 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, CA, USA, 1988.

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *PPDP'03: the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 9–18. ACM Press, 2003.

Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

Lennart Augustsson. *Compiling pattern-matching*. PhD thesis, Chalmers University, 1987.

Jeremy Avigad. A variant of the double negation translation. Technical Report CMU-PHIL 179, Department of Philosophy, Carnegie Mellon University, 2006.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15, 2008.

Franco Barbanera and Stefano Berardi. A symmetric λ-calculus for "classical" program extraction. *Information and Computation*, 125:103–117, 1996.

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

Gianluigi Bellin and Corrado Biasi. Towards a logic for pragmatics: Assertions and conjectures. *Journal of Logic and Computation*, 14(4):473–506, 2004.

N. D. Belnap. Display logic. *Journal of Philosophical Logic*, 11(4):375–417, 1982.

Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:183–220, 1992. Special Volume dedicated to the memory of John Myhill.

Corrado Böhm. Alcune proprieta delle forme $\beta\eta$-normali nel λK-calculus. Pubblicazioni 696, Instituto per le Applicazioni del Calcolo, Roma, 1968.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 20:1–24, 1998.

James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *IEEE Symposium on Logic in Computer Science*, pages 51–60, July 2007.

W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*. Springer-Verlag, 1981.

Wilfried Buchholz. Notation systems for infinitary derivations. *Archive for Mathematical Logic*, 30: 277–296, 1991.

Wilfried Buchholz. Explaining Gentzen's consistency proof within infinitary proof theory. In *KGC: Kurt Godel Colloquium on Computational Logic and Proof Theory*, volume 1289. LNCS, Springer-Verlag, 1997.

Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, December 2006. Technical report CMU-CS-06-162.

Kaustuv Chaudhuri and Frank Pfenning. Focusing the inverse method for linear logic. In C.-H. Luke Ong, editor, *Computer Science Logic*, volume 3634 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2005.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58, 1936.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1): 56–68, 1940.

Agata Ciabattoni and Kazushige Terui. Towards a semantic characterization of cut-elimination. *Studia Logica*, 82(1):95–119, 2006.

Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archive für Mathematischer Logik und Grundlagenforschung*, 19:139–156, 1978.

Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.

Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Selected Papers 1st Int. Workshop on Types for Proofs and Programs, TYPES'93, Nijmegen, The Netherlands, 24–28 May 1993*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, Berlin, 1994.

Karl Crary. Explicit contexts in LF (extended abstract). *Electr. Notes Theor. Comput. Sci*, 228:53–68, 2009.

Tristan Crolard. Subtractive logic. *Theoretical Computer Science*, 254(1–2):151–185, 2001.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ACM SIGPLAN International Conference on Functional Programming*, volume 35(9) of *SIGPLAN Notices*, pages 233–243. ACM Press, 2000.

Haskell Brookes Curry and Robert Feys. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958. Second printing 1968.

Vincent Danos and Jean-Louis Krivine. Disjunctive tautologies as synchronisation schemes. In Peter Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *Lecture Notes in Computer Science*, pages 292–301. Springer, 2000.

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. LKQ and LKT: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In Girard, Lafont, and Regnier, editors, *Workshop on linear logic*, volume 222 of *London Mathematical Society Lecture Notes*, pages 211–224. Cambridge University Press, 1995.

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. A new deconstructive logic: Linear logic. *The Journal of Symbolic Logic*, 62(3):755–807, 1997.

Olivier Danvy. A rational deconstruction of Landin's SECD machine. Report RS-03-33, University of Aarhus, 2003.

Olivier Danvy. Defunctionalized interpreters for higher-order languages, September 23, 2008. Invited talk at ICFP '08.

Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the ACM conference on LISP and functional programming*, pages 151–160, New York, NY, USA, 1990. ACM Press.

Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, December 2006. Preliminary version, accepted for publication.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 162–174. Association for Computing Machinery, 2001.

Rowan Davies. *Practical refinement-type checking.* PhD thesis, Carnegie Mellon University, 2005. Computer Science Department.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ACM SIGPLAN International Conference on Functional Programming*, pages 198–208, 2000.

Jeremy E. Dawson and Rajeev Goré. Formalised cut admissibility for display logic. In Victor Carreño, César Muñoz, and Sofiène Tashar, editors, *TPHOLs*, volume 2410 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2002.

Nicolaas G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

Michael Dummett. *The Logical Basis of Metaphysics*. The William James Lectures, 1976. Harvard University Press, Cambridge, Massachusetts, 1991.

Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007. CMU-CS-07-129.

Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 281–292, January 2004.

J. M. Dunn. Gaggle theory: An abstraction of Galois connections and residuation with applications to negation and various logical operations. In *JELIA 1990: Proc. Europeaan Workshop on Logics in Artificial Intelligence*, volume 478 of *Lecture Notes in Computer Science*. Springer, 1991.

Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000.

Martín Escardó. *Synthetic toplogy of data types and classical spaces*. PhD thesis, School of Computer Science, University of Birmingham, Birhmingham, England, 2004.

Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, 1989. Computer Science Department.

Michael J. Fischer. Lambda calculus schemata. In *Proceedings of an ACM Conference on Proving Assertions about Programs*, pages 104–109, 1972.

Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1994.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277. ACM press, 1991.

Carsten Führmann and Hayo Thielecke. On the call-by-value cps transform and its semantics. *Information and Computation*, 188(2):241–283, 2004.

Gerhard Gentzen. Untersuchungen über das Logische Schließen. *Mathematische Zeitschrift*, 39: 176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Gerhard Gentzen. Die Widerspruchfreiheit der reinen Zahlentheorie. *Mathematische Annalen*, 112:495–565, 1936. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, North-Holland, 1969.

G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Cambridge University Press, Cambridge, UK, 2003.

Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

Jean-Yves Girard. *Le Point Aveugle*. Hermann, 2006. English translation, *The Blind Spot*, available online at `http://iml.univ-mrs.fr/~girard/coursang/coursang.html`.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1:255–296, 1991a.

Jean-Yves Girard. On the sex of angels, 1991b. Post to LINEAR mailing list, December 6, 1991, archived at http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html.

Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201–217, 1993.

Jean-Yves Girard. On the meaning of logical rules I: syntax vs. semantics. Unpublished manuscript, 1998.

V. I. Glivenko. Sur quelques points de la logique de M. Brouwer. *Bulletins de la classe des sciences*, 15:183–188, 1929.

Kurt Gödel. Zur intuitionistischen Arithmetik und Zahlentheorie. *Ergebnisse eines mathematisches Kolloquiums*, 4:34–38, 1932.

John Greiner. Standard ML weak polymorphism can be sound. Technical Report CMU-CS-93-160, Carnegie Mellon University, May 1993.

Timothy G. Griffin. The formulae-as-types notion of control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57, January 1990.

Peter Hancock and Per Martin-Löf. Syntax and semantics of the language of primitive recursive functions. Technical Report 3, University of Stockholm, Stockholm, Sweden, 1975.

Bob Harper and Mark Lillibridge. ML with callcc is unsound, 1991. Post to TYPES mailing list, July 8, 1991, archived at http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html.

Robert Harper. Practical foundations for programming languages. Draft of February 3, 2009. Available online at http://www.cs.cmu.edu/~rwh/plbook/book.pdf.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, July 2007.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

Leon Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods*, pages 167–183, Oxford, 1961. Pergamon. Proceedings of the Symposium on Foundations of Mathematics (Warszawa 1959).

Jacques Herbrand. *Investigations in proof theory*. PhD thesis, University of Paris, 1930. English translation in *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931,* Jean van Heijenoort (ed.).

Arend Heyting. *Mathematische Grundlagenforschung, Intuitionismus, Beweistheorie*. Springer, Berlin, 1974.

David Hilbert. Die Grundlegung der elementaren Zahlenlehre. *Mathematische Annalen*, 104: 485–494, 1931.

Jaakko Hintikka. *Logic, Language-Games and Information: Kantian Themes in the Philosophy of Logic*. Clarendon Press, 1973.

My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *IEEE Symposium on Logic in Computer Science*, pages 15–25, June 1993.

Martin Hofmann and Thomas Streicher. Continuation models are universal for $\lambda\mu$-calculus. In *IEEE Symposium on Logic in Computer Science*, pages 387–395, Warsaw, Poland, 1997. IEEE Press.

William A. Howard. The formulae-as-type notion of construction, 1969. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *IEEE Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.

Jacob M. Howe. *Proof search issues in some non-classical logics*. PhD thesis, University of St Andrews, December 1998. Available as University of St Andrews Research Report CS/99/1.

I. Johansson. Der Minimalkalkül: Ein reduzieerter intuitionistischer Formalismus. *Compositio Mathematica*, 4:119–136, 1937.

Reinhard Kahle and Peter Schroeder-Heister, editors. *Proof-Theoretic Semantics*. Special issue of *Synthese*, 2006.

Steven C. Kleene. *Mathematical Logic*. John Wiley and Sons, New York, 1967.

Andrei Nikolaevich Kolmogorov. Zur Deutung der intuitionistischen Logik. *Mathematischen Zeitschrift*, 35:58–65, 1932.

Andrei Nikolaevich Kolmogorov. On the principle of the excluded middle. *Matematicheskii Sbornik*, 32:646–667, 1925. English translation in *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, Jean van Heijenoort (ed.).

Jean-Louis Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Archive for Mathematical Logic*, 40(3):189–205, 2001.

Jean-Louis Krivine. Opérateurs de mise en mémoire et traduction de Godël. *Archive for Mathematical Logic*, 30:241–267, 1990.

Yves Lafont, Bernhard Reus, and Thomas Streicher. Continuation semantics or expressing implication by negation. Technical Report 93-21, University of Munich, 1993.

Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, New York, NY, 1976.

Søren B. Lassen and Paul Blain Levy. Typed normal form bisimulation. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2007.

Olivier Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, March 2002.

Olivier Laurent. A proof of the focalization property of linear logic. Unpublished note, May 2004a.

Olivier Laurent. Polarized games. *Annals of Pure and Applied Logic*, 130(1–3):79–123, December 2004b.

Olivier Laurent. Classical isomorphisms of types. *Mathematical Structures in Computer Science*, 15 (5):969–1004, October 2005.

Xavier Leroy. Coinductive big-step operational semantics. In Peter Sestoft, editor, *European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2006.

Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 291–302, 1991.

Paul Blain Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary, University of London, 2001.

Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

Daniel R. Licata and Robert Harper. Positively dependent types. In *ACM SIGPLAN-SIGACT Workshop on Programming Languages Meets Program Verification*, January 2009.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *IEEE Symposium on Logic in Computer Science*, 2008.

Paul Lorenzen. Algebraische und logistische untersuchungen über freie Verbände. *Journal of Symbolic Logic*, 16:81–106, 1951.

Paul Lorenzen. Logik und Agon. In *Atti del Congresso Internazionale di Filosofia*, pages 187–194, Firenze, 1960. Sansoni.

Paul Lorenzen. Ein dialogisches Konstruktivitätskriterium. In *Infinitistic Methods*, pages 193–200, Oxford, 1961. Pergamon. Proceedings of the Symposium on Foundations of Mathematics (Warszawa 1959).

David B. MacQueen. References and weak polymorphism. Standard ML of New Jersey compiler release notes, 1988.

Per Martin-Löf, 1976. Letter to Michael Dummett, dated 5 March, 1976, including lecture notes transcribed by Peter Hancock. Copy received from Peter Hancock.

Per Martin-Löf. *Hauptsatz* for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, Amsterdam, 1971.

Per Martin-Löf. Infinite terms and a system of natural deduction. *Compositio Mathematica*, 24(1): 93–103, 1972.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004. Revised lecture notes from the International Summer School in Tartu, Estonia.

John McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *Information Processing 1962*, pages 21–28, 1963.

Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. 2008.

Paul-André Melliès and Nicolas Tabareau. Resource modalities in game semantics. In *IEEE Symposium on Logic in Computer Science*, pages 389–398, 2007.

Paul-André Melliès and Nicolas Tabareau. Resource modalities in game semantics, 2008. Long version of [Melliès and Tabareau, 2007], submitted to *Annals of Pure and Applied Logic*.

Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *IEEE Symposium on Logic in Computer Science*, pages 82–91, 2005.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

Grigori E. Mints. Reduction of finite and infinite derivations. *Annals of Pure and Applied Logic*, 104(1–3):167–188, 2000.

Grigori E. Mints. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics*, 10: 548–596, 1978. Appears in Grigori Mints, *Selected papers in Proof Theory.* Bibliopolis, 1992.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Chetan R. Murthy. A computational analysis of girard's translation and lc. In *IEEE Symposium on Logic in Computer Science*, pages 90–101. IEEE, June 1992.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford, England, 1990.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

P. Novikov. On the consistency of a certain logical calculus. *Matématicésky sbovnik*, 12(3):353–369, 1943.

Ichiro Ogata. Constructive classical logic as CPS-calculus. *International Journal of Foundations of Computer Science*, 11(1):89–112, 2000.

C. H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 215–227. ACM Press, 1997.

Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 190–201, 1992.

Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, London, 1987.

Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. *Studies in Logic and the Foundations of Mathematics*, 2008. Festschrift in Honor of Peter B. Andrews on His 70th Birthday.

Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Programming Language Design and Implementation*, pages 199–208, 1988.

Frank Pfenning and Carsten Schürmann. System Description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In Sergio Antoy and Elvira Albert, editors, *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 163–173. Association for Computing Machinery, 2008.

Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, 2008.

Garrell Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, New York, 1980.

Dag Prawitz. On the idea of a general proof theory. *Synthese*, 27:63–77, 1974.

Greg Restall. Multiple conclusions. In *Logic, Methodology and Philosophy of Science: Proceedings of the Twelfth International Congress*, pages 189–205, 2005.

Greg Restall. Display logic and gaggle theory. *Reports in Mathematical Logic*, 29:133–146, 1995.

Greg Restall. Displaying and deciding substructural logics I: Logics with contraposition. *Journal of Philosophical Logic*, 27(2):179–216, 1998.

John C. Reynolds. The meaning of types: From intrinsic to extrinsic semantics. Report RS-00-32, University of Aarhus, December 2000.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

Patrick Sallé. Une extension de la théorie des types. In G. Ausiello and C. Böhm, editors, *Automata, languages and programming. Fifth Colloquium*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410. Springer Verlag, 1978.

Kurt Schütte. Beweistheoretische Erfassung der unendliche Induction in der Zahlentheorie. *Mathematische Annalen*, 122:369–389, 1950.

Kurt Schütte. *Proof Theory*. Grundlehren der mathematischen Wissenschaften 225. Springer-Verlag, 1977. Translated by J. N. Crossley.

Helmut Schwichtenberg. Finite notations for infinite terms. *Annals of Pure and Applied Logic*, 94 (1-3):201–222, 1998.

Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

J. R. Shoenfield. On a restricted ω-rule. *Bulletin de l'academie Polonaise des sciences*, VII(7), 1959.

Timothy Smiley. Rejection. *Analysis*, 56(1):1–9, 1996.

Charles A. Stewart. *On the Formulae-as-Types Correspondence for Classical Logic*. PhD thesis, Computing Laboratory Programming Research Group, Oxford University, 1999.

Thomas Streicher and Ulrich Kohlenbach. Shoenfield is Gödel after Krivine. *ZEITSCHR: Mathematical Logic Quarterly (formerly Zeitschrift fuer Mathematische Logik und Grundlagen der Mathematik)*, 53, 2007.

Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, November 1998.

W. W. Tait. Infinitely long terms of transfinite type. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, Studies in Logic and the Foundations of Mathematics, pages 176–185. North-Holland, 1965.

Gaisi Takeuti. *Proof Theory*, volume 81 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1975.

Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *INFCTRL: Information and Computation (formerly Information and Control)*, 111, 1994.

Paul Taylor. Sober spaces and continuations. *Theory and Applications of Categories*, 10:248–299, July 2002.

Kazushige Terui. Computational ludics, 2008. To appear in Theoretical Computer Science.

Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. Phd thesis (report ecs-lfcs-97-376), Laboratory for Foundations of Computer Science, University of Edinburgh, 1997.

Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Computer Science Department, University of Edinburgh, Edinburgh, Scotland, 1988.

Dimiter Vakarelov. Nelson's negation on the base of weaker versions of intuitionistic negation. *Studia Logica*, 80(2-3):393–430, 2005.

Philip Wadler. Call-by-value is dual to call-by-name. In *ACM SIGPLAN International Conference on Functional Programming*, pages 189–201, 2003.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Ludwig Wittgenstein. *Philosophical Grammar*. Blackwell, Oxford, 1974.

Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Brückner, editor, *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, 1992.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1998.

Noam Zeilberger. Focusing and higher-order abstract syntax. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–369, January 2008.

Appendix A

Agda embedding of $\mathcal{L}^+$

```
module LPos where
  ------------------------------------
  --  TYPES
  ------------------------------------
  data Pos : Set where
    _+_ :  Pos -> Pos -> Pos            -- binary sums
    void : Pos                          -- void
    _*_ :  Pos -> Pos -> Pos            -- binary products
    unit : Pos                          -- unit
    ¬ :    Pos -> Pos                   -- continuations
    bool : Pos                          -- booleans
    nat :  Pos                          -- naturals
    dom :  Pos                          -- recursive domain
  infixr 15 _+_
  infixr 16 _*_

  ------------------------------------
  --  FRAMES & INDICES
  ------------------------------------
  data Frame : Set where
    •_ : Pos -> Frame
    · : Frame
    _,_ : Frame -> Frame -> Frame
  infixr 13 _,_

  infix 10 _∈_
  data _∈_ : Frame -> Frame -> Set where
    here : ∀ {Δ} -> Δ ∈ Δ
    left  : ∀ {Δ Δ₁ Δ₂} -> Δ ∈ Δ₁ -> Δ ∈ (Δ₁ , Δ₂)
    right : ∀ {Δ Δ₁ Δ₂} -> Δ ∈ Δ₂ -> Δ ∈ (Δ₁ , Δ₂)

  ------------------------------------
  --  PATTERNS
  ------------------------------------
  infix 10 _⊩_
  data _⊩_ : Frame -> Pos -> Set where
     hole : ∀ {A} -> • A ⊩ ¬ A
    #<> : · ⊩ unit
    #inl : ∀ {Δ A B}
      -> Δ ⊩ A
      -> Δ ⊩ A + B
    #inr : ∀ {Δ A B}
      -> Δ ⊩ B
      -> Δ ⊩ A + B
    #pair : ∀ {Δ₁ Δ₂ A B}
      -> Δ₁ ⊩ A -> Δ₂ ⊩ B
      -> Δ₁ , Δ₂ ⊩ A * B
    #tt : · ⊩ bool
    #ff : · ⊩ bool
    #z : · ⊩ nat
    #s_ : ∀ {Δ}
      -> Δ ⊩ nat
      -> Δ ⊩ nat
    #dn : ∀ {Δ}
      -> Δ ⊩ nat
      -> Δ ⊩ dom
    #dk : ∀ {Δ}
      -> Δ ⊩ ¬ dom
      -> Δ ⊩ dom
infixr 15 #s_

------------------------------------
--  CONTEXTS & JUDGMENTS
------------------------------------
data Ctx : Set where
  ·· : Ctx
  _,,_ : Ctx -> Frame -> Ctx
infixl 12 _,,_

infixr 10 _∈∈_
data _∈∈_ : Frame -> Ctx -> Set where
  x0 : ∀ {Δ Γ Δ'} -> Δ ∈ Δ' -> Δ ∈∈ Γ ,, Δ'
  xS_ : ∀ {Δ Γ Δ'} -> Δ ∈∈ Γ -> Δ ∈∈ Γ ,, Δ'
infixr 15 xS_

data J : Set where
  True : Pos -> J
  False : Pos -> J
  All_ : Frame -> J
  # : J
infix 10 All_


------------------------------------
--  TERMS
------------------------------------
infix 8 _⊢_
infix 13 _[_]
```

```
codata _⊢_ : Ctx -> J -> Set where
  -- values
  _[_] : ∀ {Γ A Δ}
    -> Δ ⊩ A -> Γ ⊢ All Δ
    -> Γ ⊢ True A
  -- continuations
  con_ : ∀ {Γ A}
    -> (∀ {Δ} -> Δ ⊩ A -> Γ ,, Δ ⊢ #)
    -> Γ ⊢ False A

  -- substitutions
  sHole : ∀ {Γ A} -> Γ ⊢ False A -> Γ ⊢ All • A
  sNil : ∀ {Γ} -> Γ ⊢ All ·
  sJoin : ∀ {Γ Δ₁ Δ₂}
    -> Γ ⊢ All Δ₁ -> Γ ⊢ All Δ₂ -> Γ ⊢ All Δ₁ , Δ₂

  -- expressions
  throw : ∀ {Γ A}
    -> • A ∈∈ Γ -> Γ ⊢ True A
    -> Γ ⊢ #
  ℧ : ∀ {Γ} -> Γ ⊢ #

------------------------------------
--  SOME EASY LEMMAS
------------------------------------
appSub : ∀ {Γ Δ}
  -> Γ ⊢ All Δ
  -> (∀ {A} -> • A ∈ Δ -> Γ ⊢ False A)
appSub (sHole K) here ~ K
appSub sNil ()          -- the empty frame has no variables, so this case is refuted
appSub (sJoin σ₁ σ₂) (left κ)  ~ appSub σ₁ κ
appSub (sJoin σ₁ σ₂) (right κ) ~ appSub σ₂ κ

sub_ : ∀ {Γ Δ}
  -> (∀ {A} -> • A ∈ Δ -> Γ ⊢ False A)
  -> Γ ⊢ All Δ
sub_ {Δ = ·} f ~ sNil
sub_ {Δ = Δ₁ , Δ₂} f ~ sJoin (sub \x -> f (left x)) (sub \x -> f (right x))
sub_ {Δ = • _} f ~ sHole (f here)

appCon : ∀ {Γ A}
  -> Γ ⊢ False A
  -> ∀ {Δ} -> Δ ⊩ A -> Γ ,, Δ ⊢ #
appCon (con K) p ~ K p

trans∈ : {Δ₁ Δ₂ Δ₃ : Frame}
        -> Δ₁ ∈ Δ₂ -> Δ₂ ∈ Δ₃ -> Δ₁ ∈ Δ₃
trans∈ Δ₁ here = Δ₁
trans∈ here Δ₁ = Δ₁
trans∈ Δ₁ (left Δ₂)  = left (trans∈ Δ₁ Δ₂)
trans∈ Δ₁ (right Δ₂) = right (trans∈ Δ₁ Δ₂)
```

```
trans∈∈ : ∀ {Δ₁ Δ₂ Γ} -> Δ₁ ∈ Δ₂ -> Δ₂ ∈∈ Γ -> Δ₁ ∈∈ Γ
trans∈∈ x (x0 y) = x0 (trans∈ x y)
trans∈∈ x (xS y) = xS (trans∈∈ x y)

------------------------------------
--  IDENTITY TERMS
------------------------------------
mutual
  IdCon : ∀ {Γ A} -> • A ∈∈ Γ -> Γ ⊢ False A
  IdCon κ ~ con \p -> throw (xS κ) (p [ IdSub (x0 here) ])

  IdSub : ∀ {Γ Δ} -> Δ ∈∈ Γ -> Γ ⊢ All Δ
  IdSub {Δ = • A} κ ~ sHole (IdCon κ)
  IdSub {Δ = ·} σ ~ sNil
  IdSub {Δ = Δ₁ , Δ₂} σ ~ sJoin (IdSub (trans∈∈ (left here) σ))
                                (IdSub (trans∈∈ (right here) σ))

------------------------------------
--  CONTEXT SPLITTING
------------------------------------
-- to define composition (and weakening), we first
-- need to define a notion of context splitting
data split : Ctx -> Ctx -> Frame -> Ctx -> Set where
  here : ∀ {Δ Γ}
       -> split (Γ ,, Δ) Γ Δ ··
  skip : ∀ {Γ Γ₁ Δ Γ₂ Δ'}
       -> split Γ Γ₁ Δ Γ₂
       -> split (Γ ,, Δ') Γ₁  Δ (Γ₂ ,, Δ')
  assoc : ∀ {Γ Δ₁ Δ₂ Γ₁ Δ Γ₂}
       -> split (Γ ,, Δ₁ ,, Δ₂) Γ₁ Δ Γ₂
       -> split (Γ ,, ( Δ₁ , Δ₂ )) Γ₁ Δ Γ₂
  nil : ∀ {Γ Γ₁ Δ Γ₂}
       -> split Γ Γ₁ Δ Γ₂
       -> split (Γ ,, ·) Γ₁ Δ Γ₂

_++_ : Ctx -> Ctx -> Ctx
Γ₁ ++ ·· = Γ₁
Γ₁ ++ (Γ₂ ,, Δ) = (Γ₁ ++ Γ₂) ,, Δ
infixr 12 _++_

data Either (A B : Set) : Set where
  Inl : A -> Either A B
  Inr : B -> Either A B

casevar : ∀ {Γ Γ₁ Δ Γ₂ A}
        -> split Γ Γ₁ Δ Γ₂ -> • A ∈∈ Γ
        -> Either (• A ∈ Δ) (• A ∈∈ Γ₁ ++ Γ₂)
casevar here (x0 κ) = Inl κ
casevar here (xS κ) = Inr κ
casevar (skip s) (x0 κ) = Inr (x0 κ)
casevar (skip s) (xS κ) with casevar s κ
...           | Inl κ' = Inl κ'
...           | Inr κ' = Inr (xS κ')
casevar (assoc s) (x0 (left κ)) = casevar s (xS (x0 κ))
casevar (assoc s) (x0 (right κ)) = casevar s (x0 κ)
casevar (assoc s) (xS κ) = casevar s (xS (xS κ))
casevar (nil s) (xS κ) = casevar s κ
casevar (nil s) (x0 ())

------------------------------------
--  WEAKENING
------------------------------------
weakvar : ∀ {Γ Γ1 Δ Γ2 Δ0}
     -> split Γ Γ1 Δ Γ2 -> Δ0 ∈∈ Γ1 ++ Γ2
     -> Δ0 ∈∈ Γ
weakvar here κ = (xS κ)
weakvar (skip s) (xS κ) = xS (weakvar s κ)
weakvar (skip s) (x0 κ) = x0 κ
weakvar (assoc s) κ with weakvar s κ
...           | x0 κ' = x0 (right κ')
...           | xS x0 κ' = x0 (left κ')
...           | xS xS κ' = xS κ'
weakvar (nil s) κ = xS (weakvar s κ)

weaktm : ∀ {Γ Γ1 Δ Γ2 J}
     -> split Γ Γ1 Δ Γ2 -> Γ1 ++ Γ2 ⊢ J
     -> Γ ⊢ J
weaktm s (p [ σ ]) ~ p [ weaktm s σ ]
weaktm s (con φ) ~ con (\p -> weaktm (skip s) (φ p))
weaktm s (sHole K) ~ sHole (weaktm s K)
weaktm s (sNil) ~ sNil
weaktm s (sJoin σ1 σ2) ~ sJoin (weaktm s σ1) (weaktm s σ2)
weaktm s (throw κ V) ~ throw (weakvar s κ) (weaktm s V)
weaktm s ℧ ~ ℧

------------------------------------
--  COMPOSITION
------------------------------------
mutual
  Con∘Val : ∀ {Γ A} -> Γ ⊢ False A -> Γ ⊢ True A -> Γ ⊢ #
  Con∘Val K (p [ σ ]) ~ Tm∘Sub here (appCon K p) σ
  Tm∘Sub : ∀ {Γ Γ1 Δ Γ2 J} -> split Γ Γ1 Δ Γ2 -> Γ ⊢ J
              -> Γ1 ++ Γ2 ⊢ All Δ -> Γ1 ++ Γ2 ⊢ J
  Tm∘Sub s (throw κ V) σ with casevar s κ
  ...         | Inl κ' ~ Con∘Val (appSub σ κ') (Tm∘Sub s V σ)
  ...         | Inr κ' ~ throw κ' (Tm∘Sub s V σ)
  Tm∘Sub s (p [ σ0 ] ) σ ~  p [ Tm∘Sub s σ0 σ ]
  Tm∘Sub s sNil σ ~ sNil
  Tm∘Sub s (sJoin σ1 σ2) σ ~ sJoin (Tm∘Sub s σ1 σ) (Tm∘Sub s σ2 σ)
  Tm∘Sub s (sHole K) σ ~ sHole (Tm∘Sub s K σ)
  Tm∘Sub s (con K) σ ~ con \p -> Tm∘Sub (skip s) (K p) (weaktm here σ)
  Tm○Sub s ℧ σ ~ ℧

------------------------------------
--  ENVIRONMENT SEMANTICS
------------------------------------
data Env : Ctx -> Set where
  emp : Env ··
  _bind_ : ∀ {Γ Δ} -> Env Γ -> Γ ⊢ All Δ -> Env (Γ ,, Δ)
infixl 8 _bind_

lookup : ∀ {Γ A}
      -> Env Γ -> • A ∈∈ Γ -> Γ ⊢ False A
lookup emp ()   -- impossible pointer into empty environment
lookup (γ bind σ) (x0 κ) = weaktm here (appSub σ κ)
lookup (γ bind σ) (xS κ) = weaktm here (lookup γ κ)

data Prog : Set where
  prog : ∀ {Γ} -> Env Γ -> Γ ⊢ # -> Prog

codata Result : Set where
  abort : Result
  step : Result -> Result

eval : Prog -> Result
eval (prog γ (throw κ (p [ σ ]))) ~
     step (eval (prog (γ bind σ) (appCon (lookup γ κ) p)))
eval (prog γ ℧) ~ abort

------------------------------------
--  SAFETY TESTING
------------------------------------
data Bool : Set where
  True : Bool
  False : Bool

data Nat : Set where
  Z : Nat
  S_ : Nat -> Nat
infixr 12 S_

safeN : Result -> Nat -> Bool
safeN R Z = True
safeN abort (S _) = False
safeN (step R) (S n) = safeN R n
data Void : Set where
data Unit : Set where
  u : Unit

isTrue : Bool -> Set
isTrue True = Unit
isTrue False = Void
```

```
isFalse : Bool -> Set
isFalse True = Void
isFalse False = Unit

------------------------------------
--  EXAMPLES
------------------------------------
selfapp : ∀ {Γ} -> Γ ⊢ False dom
selfapp ~ con K
  where
    K : ∀ {Δ} -> Δ ⊩ dom -> _ ,, Δ ⊢ #
    K (#dk hole) ~ throw (x0 here) (#dk hole [ sHole selfapp ])
    K (#dn _) ~ ℧

ωω : Prog
ωω = prog (emp bind sHole selfapp)
          (throw (x0 here) (#dk hole [ sHole selfapp ]))

safe10ωω : isTrue (safeN (eval ωω) (S S S S S S S S S S Z))
safe10ωω = u

isEven : ∀ {Γ} -> • bool ∈∈ Γ -> Γ ⊢ False nat
isEven κ ~ con K
  where
    K : ∀ {Δ} -> Δ ⊩ nat -> _ ,, Δ ⊢ #
    K #z ~ throw (xS κ) (#tt [ sNil ])
    K (#s #z) ~ throw (xS κ) (#ff [ sNil ])
    K (#s #s n) ~ K n

branch : ∀ {Γ} -> Γ ⊢ # -> Γ ⊢ # -> Γ ⊢ False bool
branch E₁ E₂ ~ con K
  where
    K : ∀ {Δ} -> Δ ⊩ bool -> _ ,, Δ ⊢ #
    K #tt ~ weaktm here E₁
    K #ff ~ weaktm here E₂

even9 : Prog
even9 = prog (emp
      bind sHole selfapp
      bind sHole (branch (throw (x0 here) (#dk hole [ sHole selfapp ])) ℧)
      bind sHole (isEven (x0 here)))
        (throw (x0 here) (#s #s #s #s #s #s #s #s #s #z [ sNil ]))

safe1even9 : isTrue (safeN (eval even9) (S Z))
safe1even9 = u
safe2even9 : isTrue (safeN (eval even9) (S S Z))
safe2even9 = u
unsafe3even9 : isFalse (safeN (eval even9) (S S S Z))
unsafe3even9 = u
```

Appendix B

Twelf embedding of $\mathcal{L}^+$

```
%%%%%%%%%%%%%%%%%%%%
%  TYPES
%%%%%%%%%%%%%%%%%%%%
pos : type.
int :  pos.                             % primitive integers
+ :    pos -> pos -> pos.               % binary sums
void : pos.                             % void
* :    pos -> pos -> pos.               % binary products
unit : pos.                             % unit
¬ :    pos -> pos.                      % continuations
rec : (pos -> pos) -> pos.              % recursive types

%infix right 13 +.
%infix right 14 *.

%% some type definitions
bool : pos                         % booleans
    = unit + unit.
nat : pos                          % unary nats
    = rec [X] unit + X.
list : pos -> pos                  % cons lists
    = [A] rec [X] unit + A * X.
→ : pos -> pos -> pos              % CBV-CPS functions
    = [A] [B] ¬ (A * ¬ B).
    %infix right 12 →.
D : pos                            % domain D = D -> D
    = rec [X] X → X.

%% "primitive" arithmetic
i : type.
z : i.
s : i -> i.   %prefix 10 s.

add : i -> i -> i -> type.
%mode add +M +N -P.
add/z : add z N N.
add/s : add (s M) N (s P) <- add M N P.
```

```
%worlds () (add M _ _).
%total (M) (add M _ _).
%unique add +M +N -P.

%%%%%%%%%%%%%%%%%%%%
%  FRAMES
%%%%%%%%%%%%%%%%%%%%
frame : type.
· : frame.
, : frame -> frame -> frame.
● : pos -> frame.
%infix right 11 ,.

%%%%%%%%%%%%%%%%%%%%
%  PATTERNS
%%%%%%%%%%%%%%%%%%%%
⊩ : frame -> pos -> type.
%infix none 9 ⊩.

n   : i -> · ⊩ int.
inl : DΔ ⊩ A -> DΔ ⊩ A + B.
inr : DΔ ⊩ B -> DΔ ⊩ A + B.
u   : · ⊩ unit.
pair : DΔ1 ⊩ A -> DΔ2 ⊩ B -> DΔ1 , DΔ2 ⊩ A * B.
fold : DΔ ⊩ A (rec A) -> DΔ ⊩ rec A.
hole : ● A ⊩ ¬ A.

%% some pattern definitions
tt : · ⊩ bool = inl u.
ff : · ⊩ bool = inr u.
zz : · ⊩ nat = fold (inl u).
ss : DΔ ⊩ nat -> DΔ ⊩ nat = [p] fold (inr p).  %prefix 9 ss.
nil : · ⊩ list A = fold (inl u).
cons : DΔ1 ⊩ A -> DΔ2 ⊩ list A -> DΔ1 , DΔ2 ⊩ list A
 = [p1] [p2] fold (inr (pair p1 p2)).

%%%%%%%%%%%%%%%%%%%%
%  JUDGMENTS
%%%%%%%%%%%%%%%%%%%%
j : type.
true : pos -> j.    %prefix 10 true.
false : pos -> j.   %prefix 10 false.
all : frame -> j.   %prefix 10 all.
# : j.


%%%%%%%%%%%%%%%%%%%%
%  TERMS & TERMS-IN-CONTEXT
%%%%%%%%%%%%%%%%%%%%
tm : j -> type.
⊢ : frame -> j -> j.
```

```
%infix right 9 ⊢.

λ_ : tm J -> tm (DΔ ⊢ J).
λ, : tm (DΔ_1 ⊢ DΔ_2 ⊢ J) -> tm (DΔ_1 , DΔ_2 ⊢ J).
λcon : (tm (false A) -> tm J) -> tm (• A ⊢ J).
λsub : (tm (all DΔ) -> tm J) -> tm (DΔ ⊢ J).
%prefix 9 λ_. %prefix 9 λ,. %prefix 9 λcon. %prefix 9 λsub.

%% values %%
val : DΔ ⊩ A -> tm (all DΔ) -> tm (true A).

%% substitutions %%
shole : tm (false A) -> tm (all • A).
snil : tm (all ·).
sjoin : tm (all DΔ1) -> tm (all DΔ2) -> tm (all DΔ1 , DΔ2).

%% expressions %%
throw : tm (false A) -> tm (true A) -> tm #.
let : tm (all DΔ) -> tm (DΔ ⊢ #) -> tm #.
℧ : i -> tm #.

%% the "apply function" for continuations
%% we will give it interesting clauses later...
body : tm (false A) -> DΔ ⊩ A -> tm (DΔ ⊢ #) -> type.
%mode body +K +P -E.
%worlds () (body K P _).
%total P (body K P _).
%unique body +K +P -E.

%%%%%%%%%%%%%%%%%%%
%  OPERATIONAL SEMANTICS
%%%%%%%%%%%%%%%%%%%
result : type.
halt : i -> result.

load : tm (all DΔ) -> tm (DΔ ⊢ J) -> tm J -> type.
%mode load +Sσ +T -T'.

ld/tm : load Sσ (λ_ T) T.
ld/join : load (sjoin Sσ_1 Sσ_2) (λ, T) T''
          <- load Sσ_1 T T'
          <- load Sσ_2 T' T''.
ld/con : load (shole K) (λcon T*) (T* K).
ld/sub : load Sσ (λsub T) (T Sσ).
%worlds () (load _ _ _).
%total (Sσ) (load Sσ _ _).
%unique load +Sσ +T -T.

eval : tm # -> result -> type.
%mode eval +E -R.
```

```
ev/load : eval (let Sσ E) R
       <- load Sσ E E'
       <- eval E' R.
ev/throw : eval (throw K (val P Sσ)) R
       <- body K P E
       <- eval (let Sσ E) R.
ev/℧ : eval (℧ N) (halt N).

%worlds () (eval _ _).
%covers eval +E -R.

%%%%%%%%%%%%%%%%%%%%
%  EXAMPLE CONTINUATIONS
%%%%%%%%%%%%%%%%%%%%
ignore : tm (false A).
ignore/p : body ignore P (λ_ ℧ z).

exit : tm (false int).
exit/n : body exit (n N) (λ_ ℧ N).

%% Check the above definitions are exhaustive...
%total P (body K P _).
%unique body +K +P -E.

iszero : tm # -> tm # -> tm (false int).
iszero/z  : body (iszero Ez Enz) (n z) (λ_ Ez).
iszero/nz : body (iszero Ez Enz) (n (s _)) (λ_ Enz).

plus : tm (false int) -> tm (false (int * int)).
plus/mn : body (plus K) (pair (n M) (n N)) (λ_ throw K (val (n P) snil))
             <- add M N P.

succ : tm (false nat) -> tm (false nat).
succ/n : body (succ K) N (λsub [σ] throw K (val (ss N) σ)).

plus' : tm (false nat) -> tm (false (nat * nat)).
plus'/zn : body (plus' K) (pair zz N) (λ, λ_ λsub [σ]
                throw K (val N σ)).
plus'/sn : body (plus' K) (pair (ss M) N) (λsub [σ]
                throw (plus' (succ K)) (val (pair M N) σ)).

add1 : tm (false int) -> tm (false int).
add1/n : body (add1 K) (n N) (λ_ throw K (val (n (s N)) snil)).

n2i : tm (false int) -> tm (false nat).
n2i/zz : body (n2i K) zz (λ_ throw K (val (n z) snil)).
n2i/ss : body (n2i K) (ss N) (λsub [σ] throw (n2i (add1 K)) (val N σ)).

%% Check the above definitions are exhaustive...
%total P (body K P _).
%unique body +K +P -E.
```

```
%%%%%%%%%%%%%%%%%%%
%  EXAMPLE EVALUATIONS
%%%%%%%%%%%%%%%%%%%%
%query 1 *
  eval (throw (plus exit)
         (val (pair (n (s s z)) (n (s s z))) (sjoin snil snil)))
        R.

%query 1 *
  eval (throw
         (plus' (n2i exit)) (val (pair (ss ss zz) (ss ss ss zz)) (sjoin snil snil)))
        R.

%%%%%%%%%%%%%%%%%%%%
%  ENCODING OF DIRECT-STYLE
%%%%%%%%%%%%%%%%%%%%
% We begin by defining some continuation combinators...
fst : tm (false A) -> tm (false (A * B)).
fst/xy : body (fst K) (pair P1 P2) (λ, λsub [σ₁] λsub [σ₂] let σ₁ E)
         <- body K P1 E.
snd : tm (false B) -> tm (false (A * B)).
snd/xy : body (snd K) (pair P1 P2) (λ, λsub [σ₁] λsub [σ₂] let σ₂ E)
         <- body K P2 E.

case : tm (false A) -> tm (false B) -> tm (false (A + B)).
case/inl : body (case K1 K2) (inl P) E1
        <- body K1 P E1.
case/inr : body (case K1 K2) (inr P) E2
        <- body K2 P E2.

not : (tm (false A) -> tm #) -> tm (false ¬ A).
not/k : body (not K) hole (λcon [k] K k).

con : ({Δ} Δ ⊩ A -> tm (all Δ) -> tm #) -> tm (false A).
con/x : body (con K) P (λsub [σ] K _ P σ).

conV : (tm (true A) -> tm #) -> tm (false A).
conV/x : body (conV K) P (λsub [σ] K (val P σ)).

uncurry : (tm (false B) -> tm (false A)) -> tm (false (A * ¬ B)).
uncurry/pk : body (uncurry K) (pair P hole) (λ, λsub [σ₁] λcon [k] throw (K k) (val P σ₁)).

%total P (body K P _).
%unique body +K +P -E.

% Now we define macros for programming in direct-style...
%abbrev cmp : pos -> type = [A] tm (false A) -> tm #.

%abbrev Lift : tm (true A) -> cmp A
   = [x] [k] throw k x.
```

```
%abbrev
 Pair : cmp A -> cmp B -> cmp (A * B)
   = [e1] [e2] [k] e1 (con [_] [p1] [σ1] e2 (con [_] [p2] [σ2]
                 throw k (val (pair p1 p2) (sjoin σ1 σ2)))).

%abbrev
 Fst : cmp (A * B) -> cmp A
   = [e] [k] e (fst k).
%abbrev
 Snd : cmp (A * B) -> cmp B
   = [e] [k] e (snd k).

%abbrev
 Inl : cmp A -> cmp (A + B)
   = [e] [k] e (con [_] [p] [σ1] throw k (val (inl p) σ1)).
%abbrev
 Inr : cmp B -> cmp (A + B)
   = [e] [k] e (con [_] [p] [σ2] throw k (val (inr p) σ2)).

%abbrev
 Case : cmp (A + B) -> (tm (true A) -> cmp C) -> (tm (true B) -> cmp C) -> cmp C
   = [e] [f] [g] [k] e (case (conV [x] (f x) k) (conV [y] (g y) k)).

%abbrev
 Fn : (tm (true A) -> cmp B) -> cmp (A → B)
   = [f] Lift (val hole (shole (uncurry [k’] conV [x] (f x) k’))).

%abbrev
 App : cmp (A → B) -> cmp A -> cmp B
   = [f] [e] [k]
       f (not [kf]
       e (con [_] [p] [σ1] throw kf (val (pair p hole) (sjoin σ1 (shole k))))).

%abbrev
 Z : cmp int
   = Lift (val (n z) snil).
%abbrev
 S : cmp int -> cmp int
   = [e] [k] e (add1 k).
%prefix 9 S.

%abbrev
 Plus : cmp int -> cmp int -> cmp int
   = [e1] [e2] [k] e1 (conV [n1] e2 (conV [n2]
      (Pair (Lift n1) (Lift n2)) (plus k))).

%abbrev
 Plus* : cmp int -> cmp int -> cmp int
   = [e1] [e2] [k] e1 (con [_] [n1] [σ1] e2 (con [_] [n2] [σ2]
      throw (plus k) (val (pair n1 n2) (sjoin σ1 σ2)))).
```

```
%abbrev
 Abort0 : cmp A
   = [k] ℧ z.
%abbrev
 Abort1 : cmp A
   = [k] ℧ (s z).

%%%%%%%%%%%%%%%%%%%%
%  EXAMPLE EVALUATIONS
%%%%%%%%%%%%%%%%%%%%
%abbrev
  run  : cmp int -> i -> type = [t] [n] eval (t exit) (halt n).

%query 1 *
  run (Plus (S S Z) (S S S Z)) N.

%query 1 *
  run (Plus (S S Z) Abort1) N.

%query 1 *
  run (Plus Abort0 Abort1) N.

%query 1 *
  run (App (Fn [x] Plus (Lift x) (S Z)) (S Z)) N.
```