A Dualities-Consolidating Framework to Support Systematic Programming Language Design

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät

der Eberhard Karls Universität Tübingen

zur Erlangung des Grades eines Doktors der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

Julian Jabs

aus Tübingen

Tübingen

2021

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	13.07.2021
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Klaus Ostermann
2. Berichterstatter:	Prof. Dr. Zena Ariola

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die zur Promotion eingereichte Arbeit selbständig verfasst, nur die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen als solche gekennzeichnet habe. Ich erkläre, dass die Richtlinien zur Sicherung guter wissenschaftlicher Praxis der Universität Tübingen (Beschluss des Senats vom 25.5.2000) beachtet wurden. Ich versichere an Eides statt, dass diese Angaben wahr sind und dass ich nichts verschwiegen habe. Mir ist bekannt, dass die falsche Abgabe einer Versicherung an Eides statt mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft wird.

Ort, Datum:

____

Unterschrift:

____

Abstract

In the theory of programming languages, *duality* is increasingly recognized as being important for improving economy, offering the theoretical development for one of two dual concepts "for free". Two prevalent dualities are the extensibility duality, related to the *Expression Problem*, and the De Morgan duality, related to evaluation strategies and control flow; for instance, a language which is symmetric with respect to the extensibility duality has both a facility which allows for easy extension with new variants, similar to how classes implement an interface in certain object-oriented languages, and a dual facility which allows for easy extension with new operations, as in functional programming with algebraic data types. However, this theoretical knowledge arguably has yet to be made more accessible to the practician. In particular, the design of programming languages does not yet really benefit from it in a systematic way.

As a step to improve this situation, building on these prior results, the present work presents a prototype of a, in the conceptual sense rather economical, foundational system, in which the extensibility duality and the De Morgan duality are consolidated. In particular, the system is inherently highly symmetric with respect to both dualities and their consolidation quite naturally allows to carve out the essence of the extensibility duality, thereby further optimizing the meta-level economy. As will be demonstrated, this system can serve as a framework in which various language features known from practical programming languages can be recovered (by local syntactic abstractions, a.k.a. macros) and systematically compared, including algebraic data types and function types as known from functional programming, classes and objects, and exception handling, in combination with the evaluation strategies employed by the respective languages. This is intended to facilitate a systematic analysis of programming language concepts which may aid in the design of parsimonious languages which are symmetric with respect to one or both of the mentioned dualities. For the more short-term perspective, the system may also serve as a cornerstone for the systematic development of tools which automatically semantically compare (and convert between) programs in different languages by means of analyzing the results of embedding them into the framework.

Zusammenfassung

In der theoretischen Betrachtung von Programmiersprachen wird *Dualität* als zunehmend wichtig für die Verbesserung der Ökonomie betrachtet, da diese ermöglicht, die Theorie-Entwicklung für eines von zwei dualen Konzepten "umsonst" zu erhalten. Zwei vorherrschende Dualitäten sind die Extensibilitäts-Dualität, die im Zusammenhang mit dem *Expression Problem* steht, und die De Morgan-Dualität, die im Zusammenhang mit Auswertungsstrategien und Kontrollfluss steht; zum Beispiel bietet eine Sprache, die symmetrisch in Bezug auf die Extensibilitäts-Dualität ist, sowohl ein Konstrukt, das die einfache Hinzufügung von neuen Varianten ermöglicht, ähnlich dazu wie in gewissen Objekt-Orientierten Sprachen Klassen ein Interface implementieren, als auch ein duales Konstrukt, das die einfache Hinzufügung von neuen Operationen ermöglicht, wie in der Funktionalen Programmierung mit algebraischen Datentypen. Dieses theoretische Wissen muss wohl allerdings dem Praktiker noch besser zugänglich gemacht werden. Insbesondere profitiert die Entwicklung von Programmiersprachen noch nicht wirklich auf eine systematische Weise davon.

Als Schritt auf dem Weg dahin, diese Situation zu verbessern, präsentiert diese Arbeit, auf diesen bisherigen Resultaten aufbauend, ein grundlegendes, im konzeptuellen Sinne recht ökonomisches System, in dem die Extensibilitäts-Dualität und die De Morgan-Dualität miteinander vereinigt sind. Insbesondere ist dieses System inhärent höchst symmetrisch in Bezug auf beide Dualitäten und deren Vereinigung ermöglicht auf recht natürliche Weise die Essenz der Extensibilitäts-Dualität herauszuarbeiten, was die Ökonomie auf der Meta-Ebene weiter verbessert. Wie dargestellt werden wird, kann dieses System als Framework dienen, in dem sich verschiedene Sprach-Features aus in der Praxis relevanten Programmiersprachen darstellen lassen (durch lokale syntaktische Abstraktionen, auch bekannt als Macros) und in dem man diese vergleichen kann, wie etwa algebraische Datentypen und Funktionstypen, wie man sie aus der Funktionalen Programmierung kennt, Klassen und Objekte, sowie Exception-Handling, in Verbindung mit den Auswertungsstrategien die von den jeweiligen Sprachen verwendet werden. Dies soll dem Zweck dienen, eine systematische Analyse von Programmiersprachen-Konzepten zu ermöglichen, welche bei der Entwicklung von kompakten Sprachen helfen kann, die symmetrisch in Bezug auf eine oder beide der erwähnten Dualitäten sind. Für die kurzfristigere Perspektive bietet es das System auch als Grundstein für die systematische Entwicklung von Tools an, welche automatisch Programme in verschiedenen Sprache semantisch vergleichen (und ineinander umwandeln), indem sie die Ergebnisse von deren Einbettung in das Framework analysieren.

Acknowledgements

I would like to thank my advisor Klaus Ostermann for his support, and the colleagues I worked with during my time at the Programming Languages research group at the University of Tübingen. Especially, I want to thank David Binder and Ingo Skupin for the great collaboration on the research project "Uroboro" in the context of which this thesis came to be, and two former members of the research group who started the project together with Klaus Ostermann, Tillmann Rendel and Julia Trieflinger. I also would like to thank all the other members and former members that I met during my time at the research group for all the interesting discussions and generally the great teamwork, especially: Jonathan Brachthäuser, Philipp Schuster, Luzia Leifheit, Steven Lolong, and Paolo Giarrusso.

Contents

Eidesstattliche Erklärung iii

Abstract v

Acknowledgements ix

1 Introduction: Dualities in the Programming Language Design Space 1
- 1.1 Dualities in Practice . . . 2
- 1.2 Developing a Design Theory . . . 4
- 1.3 Content Overview and Contributions . . . 6

2 Symmetric Extensibility in Logic and Programming Languages 9
- 2.1 The Cut Rule, Or: Engineering, Logician Style . . . 9
  - 2.1.1 Natural deduction . . . 10
  - 2.1.2 Hypothetical judgment . . . 11
  - 2.1.3 Sequent calculus . . . 12
  - 2.1.4 The Cut rule . . . 14
- 2.2 Defunctionalization: A Transformation for the Logical Engineer . . . 15
  - 2.2.1 Defunctionalization: Example . . . 16
  - 2.2.2 First-order function calls are Cutting . . . 18
  - 2.2.3 Defunctionalization to Cuts . . . 19
  - 2.2.4 Applications of de- and refunctionalization . . . 21
- 2.3 The Extensibility Duality . . . 26
  - 2.3.1 Problem: Defunctionalization lacks a total inverse . . . 26
  - 2.3.2 The Expression Problem . . . 27
  - 2.3.3 Generalizing de- and refunctionalization . . . 28
  - 2.3.4 Transforming between Cuts for user-defined rules . . . 30
- 2.4 Symmetric Language Design Using the Extensibility Duality . . . 32
  - 2.4.1 A language with data and codata . . . 33
  - 2.4.2 Symmetric spots in the type system design space . . . 36

3 From the De-Morgan Duality to Dialogue-Inspired Systems 41
- 3.1 The Negation Mirror . . . 41
- 3.2 Programming in the Sequent Calculus . . . 44
  - 3.2.1 Sequent calculus embodies the negation mirror . . . 44
  - 3.2.2 Refutation/proof and left/right rules . . . 45
  - 3.2.3 Term assignments for the sequent calculus . . . 46
  - 3.2.4 Caveats of sequent calculus programs . . . 48
  - 3.2.5 Reduction requiring non-local reasoning . . . 51
- 3.3 Dialogical Polarity and Symmetry . . . 54
  - 3.3.1 Polarity . . . 55
  - 3.3.2 Meaning theories . . . 56
  - 3.3.3 The Calculus of Unity . . . 58

3.3.4 The positive and negative fragments . . . 60
3.3.5 Shifts . . . 65

4 Polarized Flow: A Framework Consolidating both Symmetries 67
4.1 Informal Introduction to Polarized Flow . . . 68
4.1.1 Computation in $\mathcal{PF}$ . . . 68
4.1.2 Daimons . . . 70
4.1.3 Program transposition for $\mathcal{PF}$ . . . 71
4.2 Core Polarized Flow . . . 73
4.2.1 Practical version . . . 74
4.2.2 Logical core . . . 75
4.2.3 Daimonic extension . . . 79
4.2.4 Soundness of the practical version . . . 81
4.3 Pragmatic Extensions . . . 83
4.3.1 Local annotations . . . 83
4.3.2 Whole-value patterns . . . 85
4.4 Polymorphic Extension . . . 87
4.4.1 Practical version . . . 87
4.4.2 Logical core . . . 92

5 Recovering Surface Languages in Polarized Flow 95
5.1 First-Order Functions . . . 97
5.1.1 Embedding call-by-value . . . 97
5.1.2 CBV reduction correspondence . . . 99
5.1.3 Embedding call-by-name . . . 102
5.1.4 Embedding call-by-need . . . 105
5.2 Data and Codata . . . 106
5.2.1 Data fragment . . . 106
5.2.2 Codata fragment and transposition . . . 109
5.2.3 Codata's greater degree of macro freedom . . . 112
5.2.4 Practical surface transposition . . . 115
5.2.5 GA(Co)DT example: interpreter transposition . . . 117
5.3 Control Effects . . . 120
5.3.1 First-order CBV language enhanced with let/cc . . . 120
5.3.2 A higher-order language with let/cc . . . 122
5.3.3 Passing first-class functions to call/cc . . . 123
5.4 Case Study: Transposing a Java Subset . . . 128
5.4.1 Embedding a Java subset with mutability . . . 130
5.4.2 Functional update restriction and codata . . . 132
5.4.3 Program transposition and the Expression Lemma . . . 134
5.4.4 Haskell vs. data fragment in $\mathcal{PF}$ . . . 140
5.4.5 Embedding exception handling . . . 143

6 Outlook: Towards Systematic and Symmetric Design 149

A Lemmas for Soundness Proofs 157
A.1 Logical core . . . 157
A.2 Polymorphic logical core . . . 157

B Proof Details for Reduction Correspondence 159

C Reduction Correspondence for let/cc 161

Bibliography **163**

List of Figures

1.1 Java example from Lämmel and Rypacek (2008). . . . 2
1.2 Haskell example from Lämmel and Rypacek (2008). . . . 3

2.1 Gentzen's LK (propositional fragment, i.e. without quantifiers). . . . 13
2.2 Utilizing CUT for reuse. . . . 14
2.3 Result of defunctionalizing map$((\lambda x.x+1),[1,2,3])$. . . . 18
2.4 Transformed proof tree ($\psi_g(f) =$ ...apply(f,t)...; ζ is body of apply). . . . 21
2.5 Recognizer for $\{0^n1^n\}$. . . . 22
2.6 Recognizer, defunctionalized to a push-down automaton. . . . 22
2.7 Dyck recognizer before and after preprocessing. . . . 23
2.8 Dyck recognizer, refunctionalized and back in direct style. . . . 25
2.9 Program matrix. . . . 30
2.10 Example program in the language of Binder et al. (2019). . . . 33

3.1 Sub-judgments for proof, refutation, contradiction. . . . 44
3.2 Sub-judgments for contingent proof, refutation, contradiction. . . . 46
3.3 Downen's core sequent calculus language. . . . 47
3.4 Left rules for disjunction and negation with their term assignments (Downen, 2017). . . . 51
3.5 *Tertium non datur* proof as a program. . . . 53
3.6 Disjunction elimination rule in natural deduction. . . . 59
3.7 Continuation example. . . . 62

4.1 Example data fragment program (left) and a corresponding $\mathcal{PF}$ program (right). . . . 68
4.2 Console interaction example for I/O daimon. . . . 71
4.3 Example $\mathcal{PF}$ program in matrix form (only one matrix, for Nat). . . . 71
4.4 The two possible linearizations for the example $\mathcal{PF}$ program from Fig. 4.3 (top: positive polarity, bottom: negative polarity). . . . 72
4.5 Overview of soundness proofs. . . . 74
4.6 Syntax of $\mathcal{PF}$. . . . 75
4.7 Typing rules for $\mathcal{PF}$. . . . 76
4.8 Reduction rules for the practical version of $\mathcal{PF}$ (identified by superscript 0 where necessary). . . . 77
4.9 Addition example. . . . 77
4.10 Reduction rules for the logical core of $\mathcal{PF}$. . . . 78
4.11 Mutable state daimon reduction rules. . . . 81
4.12 Addition example, translated to the logical core. . . . 82
4.13 Addition example with local annotation. . . . 84
4.14 Meta-circular interpreter (top) and interpreter with closures (bottom). . . . 86
4.15 Syntax for the polymorphic system. Differences to the syntax of the non-polymorphic system (Fig. 4.6) are highlighted. . . . 88

4.16 Typing rules for $\mathcal{PF}$ with parametric polymorphism. Type parameter bindings $\langle \overrightarrow{A} \rangle$ are omitted whenever they are not directly used by the respective rule. Differences to the non-polymorphic system (Fig. 4.7) are highlighted (except $\Sigma; \Sigma_T$ in the term judgments). The M-DCP rule for DECOMPOSE is straightforwardly extended to cover Σ_T^{elem}. 89
4.17 Computing the most general unifier (mgu). 90
4.18 Meta-circular interpreter with user-defined generic function type (top) and interpreter with closures (bottom). 91

5.1 Syntax, contraction and reduction rules of a simple first-order CBV language. 97
5.2 Addition function example. 97
5.3 Syntactic abstractions for the embedding into $\mathcal{PF}$. 98
5.4 Addition function example from Fig. 5.2, embedded into $\mathcal{PF}$. 99
5.5 Translating context frames and machine configurations 101
5.6 Syntax, contraction and reduction rules of a simple first-order CBN language (differences to CBV highlighted). 102
5.7 Syntactic abstractions for the CBN embedding. 103
5.8 CBN reduction correspondence for an example with $f \mapsto (y)$. Top: Thunk forcing term reduction step in between two macro-steps. Bottom: Corresponding $\mathcal{PF}$ steps. 103
5.9 Translating context frames and machine configurations (CBN). 105
5.10 Macros for call-by-need. 106
5.11 Syntax and operational semantics (data fragment). 107
5.12 Data fragment example: natural numbers with addition. 107
5.13 Addition reduction example. 107
5.14 Type system (data fragment). 108
5.15 Syntactic abstractions (data fragment). 108
5.16 Syntax and operational semantics (codata fragment). 109
5.17 Syntactic abstractions for the codata fragment (top), and modified abstractions for the data fragment isomorphic to these (bottom). 110
5.18 Addition function example from Fig. 5.2 and its transposition (left), and the respective embeddings (right). The embeddings are themselves related by transposition. 111
5.19 Simpler syntactic abstractions for the codata fragment. 113
5.20 Codata addition example from Fig. 5.18 (left; simplified version of the codata fragment specification), and its simpler embedding (right). . . . 113
5.21 From the data to the codata fragment by example. 116
5.22 Meta-circular interpreter with user-defined generic function type as one of the linearized forms of a GA(Co)DT matrix program. 118
5.23 Result of embedding the program in Fig. 5.22 into $\mathcal{PF}$. 118
5.24 Result of transposing the program in Fig. 5.23 w.r.t. Fun. 119
5.25 Un-embedding the program in Fig. 5.24. 119
5.26 Syntactic abstractions for continuation constructs. 121
5.27 Let/cc example in the enhanced first-order CBV language. 121
5.28 Example reduction and embedded reduction in $\mathcal{PF}$ for the example in Fig. 5.27 (e.g. $\mathcal{E}_0 := []$, $k_0 := \bar{\mu}\{n \Rightarrow \textbf{Result}(n)\}$). 122
5.29 Let/cc example with CBV first-class functions. 123
5.30 Example reduction and embedded reduction in $\mathcal{PF}$ for the example in Fig. 5.29 (e.g. $\mathcal{E}_0 := []$, $k_0 := \bar{\mu}\{n \Rightarrow \textbf{Result}(n)\}$). 124
5.31 Call/cc coroutines example. 126

5.32 Example reduction and embedded reduction in $\mathcal{PF}$ for the coroutines example in Fig. 5.31 (e.g. $\mathcal{E}_0 := []$, $k_0 := \bar{\mu}\{() \Rightarrow \textbf{Result}(())\}$). 127
5.33 Translating a Java subset into $\mathcal{PF}$. 131
5.34 Java example from Lämmel and Rypacek (2008). 132
5.35 Java example from Lämmel and Rypacek (2008), translated to functional update style (differences shaded in gray). 133
5.36 Translating a functional update Java subset into $\mathcal{PF}$. (Syntactic abstractions are a subset of those in Fig. 5.33.) 134
5.37 Running example in the codata fragment (top) and data fragment (bottom). 135
5.38 Categorical semantics for the data fragment. 135
5.39 Categorical semantics for the codata fragment. 136
5.40 Haskell program corresponding to the Java program in Fig. 5.34 (Lämmel and Rypacek, 2008). 140
5.41 Macro embedding for a Haskell-lookalike language. 141
5.42 "Polarity Pocket Dictionary" of Zeilberger (2008b) (added third column: $\mathcal{PF}$ types). 141
5.43 Extending the Java subset embedding with exception handling. 143
5.44 Java example with exception handling (modified version of Fig. 5.35). . 145
5.45 Exception handling macros for an ML-lookalike. 146
5.46 ML program obtained via $\mathcal{PF}$-transposing the Java program in Fig. 5.44. 146
5.47 ML program adapted from that of Fig. 5.46, using call/cc more liberally for more localized error responses. 147

Chapter 1

Introduction: Dualities in the Programming Language Design Space

For the beginner wanting to get into programming and software development, but also for the more experienced developer or consultant who needs to make the correct technological choice for their company or customer, programming and programming languages and their underlying paradigms nowadays appear as a vast jungle. In the worst case the decision for one or the other language may turn out to be the wrong one only after having already made a considerable investment. Specifically, programming languages often force upon the programmer, or at least strongly favor, a certain way to structure (in the most general sense) their program, leading the programmer to go against the structure suggested by the problem considered and/or its analysis. That is, unless they do decide to abandon the language in favor of a better fit. However, even if they had been able to better anticipate their needs, there might arise the situation where one sub-problem lends itself well to one way of structuring, and another sub-problem lends itself well to another way.[1] The central question that this work attempts to answer, or at least open a path from it to other questions leading us towards a solution, is thus: How can we help *language designers* develop parsimonious, yet diverse programming languages in a systematic way?

Turning this into a slightly more precise research program, what we will be looking for are ways to reduce the complexity of the language design space, with a focus on relating this back to the practical reality of the programmer. This work will argue for an explicit systematic approach to language design, overcoming the sometimes semi-systematic, but more often than not ad hoc approach prevalent today. Especially, as part of it a simple foundational framework, called $\mathcal{PF}$, is presented, in which existing languages can be recovered by macro embeddings, which aids the systematic exploration of the design space. Besides serving as a design aid, $\mathcal{PF}$ may also serve as a rather lightweight foundation for tools which allow to automatically compare and convert between programs in different languages.

Instrumental in the quest to reduce complexity will be the consideration of *dualities*, which, in the words of Wadler, can offer "two-for-the-price-of-one economy"

[1] For instance, for the particular challenge of *separation of concerns*, or finding the best way to *decompose systems into modules* (Parnas, 1972), Tarr et al. remark that "existing formalisms at all lifecycle phases provide only small, restricted sets of decomposition and composition mechanisms, and these typically support only a single, "dominant" dimension of separation at a time" (Tarr et al., 1999), and refer to this situation as the "tyranny of the dominant decomposition." While Tarr et al. (1999) proposed ways to overcome this, real world languages and systems arguably have only incorporated this to a limited extent.

```
abstract class Expr { abstract int eval(); abstract void modn(); }

class Num extends Expr {
    int v;
    Num(int v0) { v = v0; }
    int eval() { return v; }
    void modn(int v0) { v = v % v0; }
}

class Add extends Expr {
    Expr l; Expr r;
    Add(Expr l0, Expr r0) { l = l0; r = r0; }
    int eval() { return l.eval() + r.eval(); }
    void modn(int v0) { l.modn(v0); r.modn(v0); }
}
```

FIGURE 1.1: Java example from Lämmel and Rypacek (2008).

(Wadler, 2003). Concretely, we will consider two dualities important in programming: the *extensibility* duality, which is related to the *Expression Problem*, and the *De Morgan*, or *negation* duality, which relates the call-by-value and call-by-name *evaluation strategies* (Wadler, 2003). The framework $\mathcal{PF}$ builds upon previous work on these dualities, especially theoretical calculi developed that exhibit them in a clear way. As we will see, both dualities can be consolidated by recasting them in the unifying framework $\mathcal{PF}$, giving us an additional meta-level economy that further cuts down the complexity of the design space.

The focus here is on the extensibility duality and how to consolidate that with the De Morgan duality which will, in particular, make the extensibility duality even more economic. The role of the De Morgan duality in programming languages is the topic of a lot of research; to provide a thorough analysis of all of its facets is far beyond the scope of this work. However, a chapter is devoted to the necessary background as far as it is relevant for this work, and a different chapter demonstrates the relevance of concepts related to the De Morgan duality for the analysis of practical programming languages.

The remainder of this introduction gives a first taste of the dualities as they appear in practice, then discusses the research philosophy that influenced this work and informed its overall approach, before closing with an overview of the contributions and the content.

1.1 Dualities in Practice

Let us start by considering two programs taken from Lämmel and Rypacek (2008) and shown in Fig. 1.1 and Fig. 1.2, respectively, one written in Java and the other in Haskell. Those familiar with Java and Haskell will realize after a bit of study that these programs intend to solve (and in fact do solve) the same problem, but in two different ways that impact a possible further development in the future. There is a simple expression language made up of literal nodes (`Num`) and addition nodes (`Add`) which recursively have expressions as their children, and one is able to evaluate

```
data Expr = Num Int | Add Expr Expr

eval :: Expr → Int
eval (Num v) = v
eval (Add l r) = (eval l) + (eval r)

modn :: Expr → Int → Expr
modn (Num v) v0 = Num (v % v0)
modn (Add l r) v0 = Add (modn l v0) (modn r v0)
```

FIGURE 1.2: Haskell example from Lämmel and Rypacek (2008).

an expression (`eval`) or, when specifying a number n, to modify an expression by applying mod n to all the numbers contained in literals that appear in the expression (`modn`). Now, if we wanted to add a further operation that takes an expression as input and computes some result from it, e.g., pretty-printing, that would be easy with the Haskell program: just write a new function. Adding a new variant of node, e.g., for multiplication, to the expression language would arguably not be as easy, since it requires one to meddle with existing code by adding a case to each function that consumes an expression. For the Java program, the situation is reversed: one may easily add the multiplication node by writing a new class for it, but for adding pretty-printing one would have to modify existing classes by adding new method implementations to them.

The problem of how to safely and independently, i.e. without meddling with existing code, extend a program with both new variants and new operations has been studied extensively and has become known as the *Expression Problem* (Wadler, 1998). Many solutions with a great variety of benefits and drawbacks, as well as ranging from requiring no to very sophisticated linguistic features, have been proposed to solve it (Krishnamurthi, Felleisen, and Friedman, 1998; Zenger and Odersky, 2001; Torgersen, 2004; Ernst, Ostermann, and Cook, 2006; Swierstra, 2008; Oliveira and Cook, 2012; Wang and Oliveira, 2016). Properly dealing with recursive references is an important aspect in this area of study. However, the present work is not about further exploring the Expression Problem itself. Instead, we will consider the *extensibility dimension* associated with a particular language. There may be clever ways to achieve extensibility in the other dimension, with more or less practical usefulness, but often these are not intended by the designer of the language (such approaches are often based on the *visitor pattern*, see e.g. Palsberg and Jay (1998)). The language itself directly offers the method by which to add new operations, as in the case of Haskell with function definitions, or new variants, as in the case of Java with classes.

With this in mind, this author believes the following goal is worth pursuing, even though it is not a solution to the Expression Problem: Design a language that allows to extend some data (like the expression language) with either new variants or new operations. In this scenario, you cannot add both easily, and you still have to make the decision which extensibility matters for you beforehand, but you can pick to represent your data by either something like a data type in Haskell or something like an abstract class or interface in Java. Such a language could give you back a lot of the freedom that you could otherwise only regain by digging out some Expression Problem solution even though you may not need *simultaneous* extensibility in both

dimensions. As this work will argue for, the duality of the dimensions allows to design such a language without blowing the structural complexity of the language. Further, an idealized version of such a language shall serve us as a foundational system in which existing languages can be recovered and compared, which hopefully will further the understanding of the design space.

A brief remark about the source of the example just used is in order. In the relevant paper, Lämmel and Rypacek (2008) consider the question of *how exactly* the two programs are equivalent, in a categorical framework. Our take on this is presented at the end of Chapter 5 (which concerns recovering existing languages in our foundational system).

As for the second duality, De Morgan duality, this has been demonstrated to relate the two opposite evaluation strategies call-by-value and call-by-name (Wadler, 2003). To quickly summarize the strategies, under call-by-value the function argument is evaluated before this evaluation result gets substituted into the function's body, while under call-by-name the unevaluated argument is substituted. In the present work, the De Morgan duality of the two presents itself as call-by-value arguments corresponding, in a Curry-Howard sense (Curry, 1934; Howard, 1980), to a certain kind of proof and passing the argument as combining that proof with a certain form of refutation, and call-by-name arguments and what they are passed to corresponding to proofs and refutations with the structural forms swapped; refer to Chapter 3 and Chapter 5 for details.

Regarding the practical implications of evaluation strategies, call-by-name or variants thereof can be more efficient due to delaying evaluation until it is really necessary, particularly when additionally caching the evaluation result as in the so called *call-by-need* as used, e.g., in standard implementation of Haskell. Further, call-by-name (or a variant like call-by-need) is essential for a certain way of programming with infinite data structures (Hughes, 1989). With call-by-value, on the other hand, it is arguably easier to understand the control flow, particularly in the presence of computational effects, as one can statically see in the code when computation needed for the evaluation of the function argument will happen.

In our foundational system $\mathcal{PF}$ all control flow is explicit, and surface languages employing call-by-value, call-by-name, or both (by annotating functions accordingly, as is possible, e.g., in Scala) can be recovered by macro embeddings, as demonstrated in chapter 5. These embeddings are directly informed by the relation to structural forms of proofs and refutations mentioned above. It is thanks to this broader view of the Curry-Howard correspondence that the two dualities discussed can be consolidated by presenting them in a single, rather sleek foundational system that is strongly inspired by the careful analysis of meaning theories for classical logic by Zeilberger (2008b). Following in the footsteps of the Curry-Howard program, in doing so trying to be as consistent and applying the correspondence as universally as possible in order to achieve the maximal economy, is one building block of the methodology of this work, which we now turn to in some more depth.

1.2 Developing a Design Theory

The basic approach of this work is to look at programming languages and their features as they exist today, and, with the help of previously developed theoretical analyses of aspects of programming (languages) open a path towards a systematic theory of programming language design. As pointed out above, an important objective

here is to control the complexity of the feature space while generally being as inclusive as possible when it comes to capturing existing features. One way to achieve such economy is to exploit dualities. This work will focus on features that can be put in a direct relation to the two dualities discussed above, which arguably already account for a lot of what programming languages have to offer today. Key to this endeavor will be to reframe the conception of design features in logical, that is often *proof-theoretical*, terms. The ping-pong between logic and programming languages that started with Curry-Howard has proven to be very fruitful in the past, and this author thinks that it is worthwhile to attempt to setup a system through which to explore the design space of programming languages driven by the Curry-Howard approach and with a focus on duality where applicable. The outlook in Chapter 6 discusses some ideas on how to extend this beyond the two dualities which are, together with their consolidation, the focus of this work.

Developing a theoretical framework starting from observations is a process that in itself has been the topic of considerable research. The following sketches how the present work was informed by such research and at what stage of theory development it arguably is situated, followed by a look at a parallel development in the foundations of mathematics with comparable aims.

Gregor (2006) classifies the *stages of development* of a theory in the context of information systems into the following types: A type I stage is defined as being about analyzing and describing, a type II stage as being about explanation, but not yet precise prediction, a type III stage as being about prediction, but not yet offering explanation, a type IV stage combines both offerings of type II and III, i.e. such a theory allows to explain what has been observed as well as predicting what will be observed, and finally in a type V stage the theory explicitly prescribes how to design artifacts.

In our concrete case, type I specializes to be about describing practical aspects of programming as related to features of programming languages, and then analyzing such features in some way with the intent to bring the theory to the next development stage. As mentioned, this analysis will be conducted by looking through the (widened) Curry-Howard lens and letting this guide us in the exploitation of dualities as well as their meta-economic consolidation. The intention is that cutting down the design space of programming languages in this way will improve our understanding of the design space, and thus help us explain (type II) why an existing language has benefits or drawbacks, as well as predict (type III) whether some language design potentially leads to such. Overall, what the present work aims for is thus an early form of theory of stage type IV, with still a bit stronger focus on the bridge from analysis to explanation rather than prediction, i.e. on the type II stage.

However, as mentioned, the ultimate goal is design prescription or at least design aid, i.e. achieving stage type V. We will thus preliminarily also consider what our improved perception of the design space seems to indicate for prescribing or aiding with design. One can consider the theory presented herein as a kind of *vertical prototype* which still requires work on multiple stages but where preliminary attempts at developing a latter stage inform those at a prior stage.

In a sense the overall endeavor that is to lead us to design prescriptions/aids is a step in the direction of systematically *reverse engineering* programming language features. There is a comparable development in the foundations of mathematics, in the field of *reverse mathematics*. In his recent book on this topic, aimed at a non-specialist audience, Stillwell (2019) opines that the foundations of mathematics have unfortunately been neglected for some time now and he intends to demonstrate the importance of a particular aspect of this foundational work, the reverse engineering

of mathematical proofs. In a nutshell, reverse mathematics is the search for the *right axiom* to add to some base set of axioms to be able to prove some given theorem but not stronger ones, thereby structuring, e.g., various theorems of analysis according to their strength. By analogy, the present work is an attempt to reverse engineer the wild jungle of programming language concepts manifesting in features of existing languages.

Arguably, in contrast to the reverse mathematics program, the biggest problem in this area is not one of neglect of foundations *per se*, as there has been a lot of (Curry-Howard-inspired) work on theoretical calculi that reflect aspects of programming in an idealized form, which we can now build upon. Rather, what arguably is the greatest hinderance to the development of a design theory is the lack of striving for a bigger picture, of a systematic program like that of reverse mathematics that allows to relate the various aspects. Studying one of these in isolation can be a useful approach to deal with complexity, but ideally this should be accompanied by sometimes "zooming out" to discover dualities and other similarities as a complementary way to cut down complexity. Studying the interplay of aspects like computability, arithmetic, and analysis has proven to be highly interesting and fruitful for improving our understanding of each, enabling the systematic exploration of the space of mathematical proofs (Stillwell, 2019), and applying a similar, more systematic approach to programming (language) concepts could be just as impactful for the language design space. In a sense, the present work is intended as a preliminary proof of concept for this.

1.3 Content Overview and Contributions

Content Overview The rest of this work is structured as follows:

- Chapter 2 provides an in-depth analysis of the extensibility duality, specifically looking at how extensibility presents itself on the logic side.
- Chapter 3 discusses De Morgan duality to the extent it is relevant for this work; it focusses on the way the duality is embodied by sequent calculus and then on calculi (to serve as a basis for programming languages) based on the sequent calculus.
- Chapter 4, building on the system **CU** discussed at the end of Chapter 3, presents the foundational system $\mathcal{PF}$ which embodies both the extensibility and the De Morgan duality.
- Chapter 5 demonstrates how (idealized forms of) existing programming languages can be recovered in $\mathcal{PF}$ and how $\mathcal{PF}$ can serve as a framework to analyze and compare languages and their features; besides aiding in the design of languages, this may also facilitate the systematic development of tools for comparing and converting between programs of different languages, with different but related features.
- Chapter 6 gives an outlook on how to further develop this framework to better aid in systematic language design, by incorporating further sources of theoretical knowledge about programming languages.

Contributions The central contribution of this work is the foundational system $\mathcal{PF}$ (see Chapter 4) that consolidates the extensibility duality and the De Morgan

duality; this system is grounded in the analyses of the two dualities in Chapter 2 and Chapter 3, respectively. The utility of $\mathcal{PF}$ is demonstrated in Chapter 5 by giving macro embeddings of various existing surface language features into $\mathcal{PF}$ and, in Section 5.4, showing how it facilitates analyzing and comparing such features as they are realized in real programming languages. As a potential practical application, the framework might thus serve as the basis for automatic tools with which one can compare programs in different programming languages such as Java, Haskell, or ML, recognize the (partial) semantic equivalence of such programs and automatically convert between them. For more details on what $\mathcal{PF}$ contributes to the study of programming languages, see the respective paragraphs at the beginning of Chapter 4 and Chapter 5.

Chapter 2

Symmetric Extensibility in Logic and Programming Languages

Disclaimer: Section 2.4 of this chapter summarizes works coauthored by the author (Ostermann and Jabs, 2018; Binder et al., 2019). Any quote from these works is explicitly marked as such.

This chapter provides an in-depth analysis of the first of the two dualities considered in this work, the extensibility duality. It discusses work related to this duality from the 1970s until today, and, as a minor novel contribution, it looks at how extensibility presents itself on the logical side. Specifically, we will see how the Cut rule can be seen as the embodiment of the "Don't repeat yourself" (DRY) principle and hence how it is related to linguistic constructs that facilitate abstraction for the purpose of avoiding such redundancy. We will also begin to consider what logical duality means for the extensibility duality via this Cut rule correspondence; the next chapter will then pick up this issue, giving a new perspective on it that is in line with the consolidation of the extensibility and De Morgan dualities presented in that chapter.

Chapter overview The chapter is structured as follows:

- Section 2.1 considers the Cut rule itself and its logical background.
- Section 2.2 discusses defunctionalization and its dual, refunctionalization, as well as how to recast these on the logic side, featuring the Cut rule.
- Section 2.3 discusses how de- and refunctionalization are special cases of extensibility switching.
- Section 2.4 concludes by summarizing initial steps in designing symmetric languages by systematically exploiting the extensibility duality.

2.1 The Cut Rule, Or: Engineering, Logician Style

This section presents the necessary background for the Cut rule (which is also useful background for the next chapter) and concludes with a suggestion on how one can better understand the purpose of that rule once realizing what the programmer and the logician have in common. It is the author's intent to make this section understandable to readers with only limited exposure to logic; if you are already familiar with natural deduction and sequent calculus, feel free to skip to its last subsection.

2.1.1 Natural deduction

For the logician, an essential question is how to design the deductive system. One possible approach, called *natural deduction*, attempts to stay close to what is (nowadays) traditionally perceived of as "natural" reasoning; here the ideal is to have *inference rules* that resemble the way one would go about proving some proposition intuitively.[1] For example, to prove a conjunction ("and") of two propositions A and B, the *premises* of the rule, one would need to have established that both A and B are true. Natural deduction captures this formally by the following inference rule, where the premises one has to have established already are written above the line and the proposition the rule allows to prove from these, the *conclusion*, below it.

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

For a disjunction (non-exclusive "or") it suffices to have established the single premise A which allows to use the rule shown below on the left, or alternatively, to have established B which allows to use the rule on the right.

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

When we say that the truth of a proposition above the rule line is established, we mean that this itself happens by using inference rules only. More precisely, in the rules, the letters A and B are placeholders for propositions which are to be replaced[2] by a tree recursively formed from uses of rules; in summary, a proof is a tree with nodes labelled by propositions and where edges are constituted via the lines in the rules. As an example, where we close off the tree via a rule for $\top$ (the true constant) with zero premises, we can deduce $\top \vee (\top \wedge \top)$ with the tree shown below.

$$\frac{\dfrac{\dfrac{}{\top} \qquad \dfrac{}{\top}}{\top \wedge \top}}{\top \vee (\top \wedge \top)}$$

Now we saw some inference rules, but there are many more logical connectives that we can semantically conceive of, for instance by means of a truth table, even if we only consider connectives with arity ≤ 2. Considering its truth table, the arguably simplest connective we have not seen yet is negation. Can we give an inference rule for negation that has the same form as the ones we saw already? Looking at them again, we can discern that they all conform to this pattern:

$$\frac{J_1 \qquad \ldots \qquad J_n}{J}$$

In each rule, the $J_1, ..., J_n$ are a subset of the parameters of the connective, e.g. $J = A \wedge B$, $J_1 = A$, $J_2 = B$ for the conjunction rule and $J = A \vee B$, $J_1 = A$ and $J = A \vee B$, $J_1 = B$ for the first and second disjunction rules, respectively. Trying to write a semantically valid rule for negation that conforms to this specification is impossible, because there is only one parameter, say, A; we can either use no premise, but this would mean that the negation of every proposition is true, or the single premise $J_1 = A$ but this would mean that the negation of a proposition is true when that proposition is, and both outcomes are obviously semantically invalid.

[1] The design principle of previously developed deductive systems, now called *Hilbert-style* systems, was to minimize the necessary inference rules by resorting to axioms or axiom schemas wherever possible.

[2] In other words, the inference rule is *schematic*.

2.1.2 Hypothetical judgment

Historically, the treatment of negation necessitated not only to generalize the form of rules, but also to change the deduction mechanics which we so far saw described as simply a tree of propositions built up from the rules. Specifically, the negation rule looks as follows in this presentation of natural deduction, where $\bot$ is the false constant:

$$\dfrac{\begin{matrix}[A] \\ \vdots \\ \bot\end{matrix}}{\neg A}$$

This is to be read as: "Discharge the unproven premise A somewhere up in the tree." When a premise is *discharged*, this means it does not have to be proven anymore. Different from all other rules we saw so far, this negation rule retroactively changes the recursively constructed tree that it is used on, by carrying out such a discharging somewhere in the tree. We therefore call this rule highly *non-local*; we come back to this aspect later, for now we just remark our observation from the negation rule that non-locality is potentially problematic and should be handled with care. The negation rule complicates our definition of what constitutes a proof because for it to be applicable, sometimes a premise has to be left open such that the negation rule may discharge it. The absolute judgment of a proposition to be true, backed up by a tree, is therefore relaxed to a *hypothetical judgment* which may be contingent on the truth of some hypotheses yet to be discharged. A tree backing this up is allowed to have these hypotheses as open propositions, and may even consist of only an open proposition, with no rule used at all. For instance, with the degenerate tree below we can hypothetically judge $\bot$ to be true, contingent on the truth of $\bot$.

$$\bot$$

To this we can apply the negation rule, discharging $\bot$ and obtaining the tree:

$$\frac{[\bot]}{\neg\bot}$$

Thus we have proven $\neg\bot$ by first assuming $\bot$, which we would never have been able to prove. But we only did assume it for the sake of discharging it again to show that its negation holds, this leads to a valid result.

While this presentation of natural deduction may be fruitful for obtaining certain intuitions, the retroactive tree manipulation it requires is at the very least unwieldy, so we will now turn to a different presentation. From now on, we will still use trees but their labels themselves will be hypothetical judgments, which we write as a *sequent*,[3] where Γ is a set[4] of open propositions:

$$\Gamma \vdash A$$

For example, our judgment with the degenerate tree above becomes $\bot \vdash \bot$, and for judgments with no open hypotheses the Γ is empty, which we notationally express by just leaving it away, e.g. $\vdash \top \wedge \top$. The $\vdash$ symbol is called *turnstile*. As for the rules, for instance, the direct translation of the negation rule is:

[3] Not to be confused with the term *sequent calculus* that we will get to shortly.

[4] I.e. order and multiplicity do not matter.

$$\frac{A \vdash \bot}{\vdash \neg A}$$

As we will see, in this presentation, we are finally able, for all connectives, to give rules that conform to the simple pattern we saw before:

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

This time, the J_i and the J are all sequents, so we did not magically improve our situation without any tradeoff, but we avoided retroactive and non-local tree manipulation. Nonetheless, it arguably is a valid observation that in a certain sense we only shifted difficulties to the treatment of the sequents, but at least we have achieved a separation of concerns: describing the overall creation of the tree, which is simple because all rules are local, and dealing with the sequents, which we have thus isolated as a concept and can now investigate more efficiently.

2.1.3 Sequent calculus

If we look at the form of our sequents through a naive symmetry-desiring lens, we see that what is to the left of the turnstile consists of multiple formulas while to the right of it there is only a single formula. Reminding us that all formulas on the left-hand side, i.e. in the set Γ, are open hypothesis, i.e. the right-hand side formula holds only contingent to *all* of these holding, we can characterize Γ as a conjunction. Taking a cue from De Morgan duality that we will discuss in depth in the next chapter, what we symmetrically want on the right-hand side should be a set, say Δ, which is to be interpreted disjunctively; the form of our sequents becomes:

$$\Gamma \vdash \Delta$$

This form of the sequent is one of the two defining characteristics of *classical sequent calculus*.[5] The other concerns the formulas allowed in the Γ and the Δ. In the natural deduction rules we saw so far, when writing them in sequent style (again, not the same as sequent calculus, just a different presentation of natural deduction), no connectives appeared in the open hypotheses, i.e. these were always schematic placeholders (A, B, ...). And this is actually a defining characteristic of natural deduction that distinguishes it from sequent calculus, where one hypothesis can be built using a connective. A rule that has the connective to the left of the turnstile is called a *left rule* and a rule that has the connective to the right of the turnstile is called a *right rule*. Thus natural deduction has only right rules, while sequent calculus has both left and right rules. However, it is limited in a different fashion. The natural deduction rules we saw so far were all *introduction rules*, which means that the connective only appears in the conclusion of the rule. We have not seen their opposite, called *elimination rules*, yet; for now it suffices to know that there the connective appears in a premise, allowing to deduce, e.g, A from $A \land B$. This form of rules is allowed in natural deduction, but not in sequent calculus. In sequent calculus, only introduction rules exist; one reason for why this is desirable is that it makes all logical arguments always “flow” in one direction, in this case “upwards”. In summary, classical sequent calculus

(1) has multiple formulas on both side of the turnstile and

(2) allows only introduction rules but these can be left and right rules.

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge L_1 \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee R_1$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge L_2 \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee R_2$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi} \vee L \qquad \frac{\Gamma \vdash A, \Delta \qquad \Sigma \vdash B, \Pi}{\Gamma, \Sigma \vdash A \wedge B, \Delta, \Pi} \wedge R$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \to B \vdash \Delta, \Pi} \to L \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \to R$$

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

FIGURE 2.1: Gentzen's LK (propositional fragment, i.e. without quantifiers).

Let us now turn to concrete sequent calculus rules. In Gentzen's system LK (Gentzen, 1935), the prototypical classical sequent calculus system, which is shown in Figure 2.1, a proposition can be deduced if and only if it is valid, in other words the system is *sound* and *complete* and since the set of connectives that appears in the system is semantically complete, conditions (1) and (2) are sufficient for the allowed rules of a deductive system, i.e. you only need rules conforming to these for a system that captures all of classical logic; note that this also vindicates our naive decision for condition (1) to an extent. In particular, there is no need for any *structural rules*, which are rules that, roughly speaking, are not directly related to the semantical meaning of the connectives.[6] Overall, the author thinks that it is not an overstatement to say that LK is a rather elegant system; nevertheless, especially for the sake of making more clear the relation to data and codata later on, let us look at a modular, generic presentation of sequent calculus using a meta-schema for the rules. This system is parametric in the list of connectives, and for each of its connectives $\mathfrak{C}$ the logician—the "user" if you want—has to define left rules and right rules, as shown below, where $\Pi \subseteq \overline{\overline{A}, \overline{B}}$.

$$\frac{\overline{\Gamma, \overline{A} \vdash \overline{B}, \Delta}}{\overline{\Gamma}, \mathfrak{C}(\Pi) \vdash \overline{\Delta}} \mathfrak{C}L_i \qquad \frac{\overline{\Gamma, \overline{A} \vdash \overline{B}, \Delta}}{\overline{\Gamma} \vdash \mathfrak{C}(\Pi), \overline{\Delta}} \mathfrak{C}R_j$$

It is the user's responsibility to pick rules that semantically fit with the connective.[7] What is *not* in their responsibility is to make sure the empty sequent $\vdash$ is not deducible. This would make the system unsound, because it is interpreted as one formula of an empty disjunction of formulas holding true not contingent to any (conjunction of) hypotheses. But since the form of the rules is prescribed in such a way that the empty sequent never appears in the conclusion, this is ruled out by the system.

[5]Intuitionistic logic was not discussed yet, but we will come to its characteristics soon.

[6]LK *does* have structural rules for weakening, contraction, and permutation, but the latter two can be dispensed with when using sets of formulas as we do. In some later sections we will consider what happens when certain structural rules are removed, but this is not relevant here. Therefore, we also just gloss over weakening and silently allow it to happen.

[7]We can retroactively view Gentzen as such a user when he defined the rules of LK, and he of course did make sure the rules fit semantically, which one can easily convince oneself using the intuition of the conjunctively interpreted left-hand sides and the disjunctively interpreted right-hand sides.

$$\dfrac{\begin{matrix}\mathcal{D}_1 \\ \vdash A\end{matrix} \quad \begin{matrix}\mathcal{D} \\ A \vdash B\end{matrix}}{\vdash B}\ \textsc{Cut} \qquad\qquad \dfrac{\begin{matrix}\mathcal{D}_2 \\ \vdash A\end{matrix} \quad \begin{matrix}\mathcal{D} \\ A \vdash B\end{matrix}}{\vdash B}\ \textsc{Cut}$$

FIGURE 2.2: Utilizing CUT for reuse.

2.1.4 The Cut rule

The previous statement that all rules of LK are introduction rules was a lie, however in a moral sense it was not really. Gentzen (1935) also included the so-called AXIOM and CUT rules in LK, but the latter is entirely redundant, while the former can at least be restricted to propositional variables without losing any proof strength.

$$\dfrac{}{A \vdash A}\ \textsc{Axiom} \qquad\qquad \dfrac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \textsc{Cut}$$

For the moment, we take no further interest in the AXIOM rule. However, the CUT rule is of great interest even though being redundant. In fact, the exact way it is redundant will be important in the next chapter. In this chapter, we focus on *why* logicians care about this rule in the first place, and how this mirrors what every programmer naturally wants to achieve: "Don't repeat yourself (DRY)."

In our generic sequent calculus presentation, we add CUT as a non-user-defined rule. We have to find a different way to show soundness because it does not directly guarantee that the empty sequent cannot be derived simply by its form. This is actually achieved by showing that CUT is redundant, no matter what user-defined rules there are; this property, called *cut elimination*, will be discussed in the next chapter, for now we just remark that cut elimination viewed through the Curry-Howard lens is reduction, albeit with the qualification that their precise relation is complicated. Summing up, this generic presentation of classical sequent calculus is made up of

- module (a): restricted user-defined (thus itself modular) left and right rules and
- module (b): CUT, equipped with cut elimination.

Now, why do we want to have the CUT rule around at all? Consider the two proofs shown in Figure 2.2. They are for the same sequent and both use CUT. The difference is in their respective left premises, where we assume that $\mathcal{D}_1$ and $\mathcal{D}_2$ not to be the same. But using the CUT rule it was possible to factor out the part $\mathcal{D}$ of the proof as a common right premise. If we eschewed CUT, this reuse would not have been possible, and we would be forced to somehow merge $\mathcal{D}_1$ and $\mathcal{D}$, and $\mathcal{D}_2$ and $\mathcal{D}$. This would mean that we would have to carry out certain proof steps that amounted to the same in the end, essentially repeating what we did in one proof tree in the other, but we would have no representation of this as a standalone proof tree $\mathcal{D}$. In other words, we would violate DRY. This should remind the programmer of how they are able to get by without using any (first-order) functional abstraction, but at the cost of duplicating code with no hope of reuse. Also, the reduction of a function call, a single reduction step, corresponds to a single step in a certain elimination procedure for cuts that gets rid of that one particular cut, i.e. it is a certain local version of cut elimination; more on this in the next chapter. For now, in particular observe that reduction, like cut elimination, in general produces a result that violates DRY, which neither the programmer nor the logician will be too worried about since in both cases practical high-level engineering work with programs/proofs concluded before getting rid of CUT.

There is a connective that is related to CUT in the sense that it allows to turn the $A \vdash B$ sequent into a first-class entity of the system, which in this case is a formula. With the perspective on CUT as reuse we just considered, we can expect that we can utilize such a connective for the purpose of first-class functional abstraction; if we were not in a classical but an intuitionistic setting, Curry and Howard would tell us which one that is: implication. In the next chapter we will see a system loosely based on the classical sequent calculus with rules with which a CUT can be turned into a first-class entity by binding a free variable, which in particular allows to recover first-class functions. But even now we can already see that we are likely on the right track with the connection between CUT and implication. Compare the implication left rule with CUT: It is like a variant of CUT that "cuts" between two different propositions A and B by internalizing this cut as an implication formula in the hypotheses.

$$\frac{\Gamma \vdash \Delta, A \qquad B, \Sigma \vdash \Pi}{\Gamma, \Sigma, A \to B \vdash \Delta, \Pi} \to L$$

What is important here is that this internalization means that the kind of functional abstraction corresponding to implication is not first-order, but higher-order, and, as intended, these kinds of functions are first-class entities just like a proof of an implication, as opposed to a use of CUT, which is not internalized. For the purposes of this chapter, where we consider these ideas more generally in the light of the extensibility duality, we will work in the intuitionistic natural deduction setting, but eventually seek a way (and find it in the next chapter) to improve the symmetry which will lead us to the sequent calculus based systems.

The next section takes the first step into concretely exploring the extensibility duality. Consider that just as the logician might not always want to have the internalizing implication connective around, the programmer has reasons to avoid first-class functions. Both are sometimes interested in transforming these away by using (what corresponds to) CUT. This transformation will be the subject of the next section. The sections after the next will be about improving aspects of the transformation, and thus achieving symmetry and increasing its economy. The first underlying problem is that the transformation is very specifically, and unnecessarily so, tailored to the function type/implication connective. This will be solved by generalizing the transformation in the spirit of the modular user-defined rules presentation of this section. And then, to get the maximum symmetry economy, we turn to our other duality, which will carry us over into the next chapter.

2.2 Defunctionalization: A Transformation for the Logical Engineer

In the early 1970s, Reynolds (1972) was the first to consider the question of how to turn an interpreter exhibiting a certain set of properties, like the use of higher-order functions, or dependence of the evaluation strategy in the defined language upon that of the defining language, into an otherwise equivalent interpreter with different properties; one could call this engineering work on the interpreter. In particular, it was his goal to define transformations to get rid of the use of higher-order functions in the interpreter and of the evaluation strategy dependence. This was motivated by the idea that transforming such features away helps explaining them and by the arguably related fact that this brings the interpreter closer to a form that can be translated to machine code.

For the first goal, the relevant transformation is the so-called *defunctionalization* that replaces all first-class functions with values of a data type and an appropriately defined first-order *apply* function. For the second, the relevant transformation is the *continuation-passing style (CPS)* transform; in the next chapter we will encounter a language that could be described as inherently requiring CPS. But in this section, we focus on defunctionalization and what it has to do with the perspective on the Cut rule described in the previous section.

2.2.1 Defunctionalization: Example

Before we delve into the Cut rule connection, let us briefly see how defunctionalization exactly works, by means of a simple example. We follow the original presentation of Reynolds (1972) who uses a pseudo-mathematical notation for the abstract syntax of his language, but will point out the modern equivalents along the way. Reynolds (1972) uses defunctionalization on an interpreter, and we will see this kind of example again in later chapters; however, for now it suffices to consider a simpler example of a use of first-class functions well-known from functional programming practice.

$$\texttt{map}((\lambda x.x+1),[1,2,3])$$

Here we map an anonymous first-class function, written using a lambda abstraction, that adds one to its input, over a list containing some numbers; computing the result of this term produces $[2,3,4]$. There are no lists in the language of Reynolds (1972), so we just use the well-known square bracket notation, and, later on, the symbol :: for cons-ing an element to a list. Also, in his language, *all* functions are first-class, including the `map` function, but we do not make essential use of this function as a first-class entity, i.e. we do not pass it to other functions unlike the anonymous addition function, so we disregard the first-class status of `map` for the purpose of this example.

Our intent is to *eliminate the first-class use of functions*, which we do in the following steps (following Reynolds (1972)):

1. Add a new data set `FUN` for functions, with one data record per lambda abstraction.

2. Add a (not used in a first-class way) `apply` function that takes a `FUN` value as input, distinguishes between the records, and, for each case, has the body of the corresponding lambda abstraction as the body for that case (slightly adapted to use field accessors, see below).

3. Replace lambda abstractions with instances of their respective records and applications of first-class functions with calls to `apply`.

Let us now consider these steps in some more detail.

In our concrete example, the data set is $\texttt{FUN} = \texttt{PLUS}$, where $\texttt{PLUS} = []$ is the data record for the lambda abstraction $\lambda x.x+1$. Generally, such data records created by defunctionalization have a field per free variable that appears in the body of the lambda abstraction. For instance, if in the example we had instead mapped $(\lambda x.x+y)$ over the list, with y being a variable bound somewhere further up in the abstract syntax tree (e.g. our example term could be the body of a function that has y as a parameter), we would have $\texttt{PLUS} = [y : \texttt{INTEGER}]$. In general, properly dealing with free variables requires, at least conceptually, to first carry out *lambda lifting* before the defunctionalization; lambda lifting turns all such implicitly closed over variables into explicit additional arguments of the function, which then become record fields.

Reynolds's language is not really statically typed (though data records do specify the field types), but defunctionalization also applies to the statically typed setting in pretty much the same way (see, e.g., (Danvy and Nielsen, 2001)). Thus, in a modern style, we can express the data set `FUN` as a *data type* `Fun` which has a single constructor, corresponding to $(\lambda x.x+1)$ and `PLUS`, which we call `PlusOneFun`:

$$\textbf{data}\ \texttt{Fun}\ \{\ \texttt{PlusOneFun}\ \}$$

For the alternative lambda abstraction with the free y variable, the constructor signature would change to `PlusFun(Int)`. Generally, constructors take the place of records, so defunctionalization produces a data type with one constructor per lambda abstraction, where the arguments of the constructor correspond to the free variables of the body.

The `apply` function accordingly has a case for each data record (constructor); in our example, there is just a single case, for `PLUS`, and its body is that of $(\lambda x.x+1)$ where the functional parameter x is replaced by the second argument of `apply` (here also conveniently called x):

$$\texttt{apply} = \lambda(f,x).(\texttt{plus?}(f)\ \rightarrow\ x+1)$$

The `plus?` is a predicate that checks whether f is an instance of `PLUS`. In general, there is a predicate for each of the data records that constitute `FUN`. For the alternative lambda abstraction with a free variable y, the corresponding record would additionally induce a field accessor `y` to be applied to f (written as $\texttt{y}(f)$); in the body of `apply` for the `plus?` case, this would have to be appropriately used, i.e., the body is $x+\texttt{y}(f)$. In general, there would be field accessors for each record field.

In modern languages with data types and pattern matching, we could write the `apply` in the following way:[8]

$$\begin{aligned}&\textbf{function}\ \texttt{apply}(f,x)\ := \textbf{match}\ f\ \{\\&\quad \texttt{PlusOneFun}\ \Rightarrow\ x+1\\&\}\end{aligned}$$

In general, there would be a case for each constructor, and instead of the field accessors we could directly write the variables as part of the constructor patterns, e.g. $\texttt{PlusFun}(y)\ \Rightarrow\ x+y$.

Finally, in the term in which the lambda abstraction appeared above, we replace it with an instantiation `mk-plus()` of `PLUS`, or, in modern style, a call to the constructor `PlusOneFun` of data type `Fun`:

$$\texttt{map}(\texttt{PlusOneFun}, [1,2,3])$$

We also have to replace the application of the first-class function parameter f in the definition of `map` with a call to `apply`, i.e. we now have $\texttt{apply}(f,x)\ ::\ \texttt{map}(f,xs)$ instead of $(f\ x)\ ::\ \texttt{map}(f,xs)$. Fig. 2.3 shows the result of the entire transformation for our example $\texttt{map}((\lambda x.x+1),[1,2,3])$.

As we saw, the basic idea of this transformation is easy enough. It gets slightly more complicated when we also consider lambda abstractions with free variables, which means that we have to use lambda lifting first, or when we consider functions with different pairs of input and output types, for which we can use a polymorphic `Fun` data type and `apply` function. A bit more involved and interesting for the extensibility symmetry we consider is the defunctionalization of polymorphic functions,

[8] We also use this and similar notation for non-first-class function declarations and pattern matches (plus the notation for data types seen above) further on in this work (in particular it is similar to the notation used by Rendel, Trieflinger, and Ostermann (2015) and virtually identical to that of Binder et al. (2019) whose works are discussed in the following sections).

```
data Fun { PlusOneFun }

function apply(f, x) := match f {
    PlusOneFun ⇒ x + 1
}

function map(f, xs) := match xs {
    x :: xs ⇒ apply(f, x) :: map(f, xs)
}

map(PlusOneFun, [1, 2, 3])
```

FIGURE 2.3: Result of defunctionalizing $\texttt{map}((\lambda x.x+1),[1,2,3])$.

which requires *Generalized Algebraic Data Types (GADTs)* in the transformation target (Pottier and Gauthier, 2004). We will come back to this point in Section 2.4.2 and Section 4.4.

2.2.2 First-order function calls are Cutting

Our aim is now to demonstrate how using first-class functions logically corresponds to using implication rules and using first-order functions definition to using the Cut rule, and from this it will follow that defunctionalization, viewed logically, is a transformation for the "logical engineer" (mentioned in the previous section) eliminating uses of the implication rules by uses of the Cut rule (and a disjunctive elimination rule for the data type replacing the function type). We focus on the high-level picture; as outlined in the previous section, there is not a perfect match between DRY engineering in a programming language and the Cut rule, because the former is carried out in what logically corresponds to intuitionistic natural deduction while the latter is a rule of classical sequent calculus, but we consider the basic idea of the correspondence to be valid nevertheless, plus we will move to bridge the gap in the next chapter.

We begin our demonstration with the simple example term from above, but with some details abstracted away; we also leave away the second argument of the outer function call (just think, e.g., that we specialize to a specific list).

$$g(\lambda x.\phi(x))$$

In particular, the call to g is intended to be a first-order one, i.e. g is not a first-class function, or any first-class entity, for that matter. Let us further say that x and $\phi(x)$, which is just a placeholder for a term with free variable x, have type A, and the overall term has type B. Then we can (try to) write this term, in the sequent presentation of natural deduction (basic idea shown in the previous section), as the following proof, where, as usual, we annotate left-hand side formulas with variables and right-hand side formulas with their assigned proof terms:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ x : A \vdash \phi(x) : A\end{array}}{\vdash \lambda x.\phi(x) : A \to A} \to I \qquad \begin{array}{c}\mathcal{D}_2\\ g\end{array}}{\vdash g(\lambda x.\phi(x)) : B} \; ?$$

To go from $A \vdash A$ to $\vdash A \to A$, we use the implication introduction rule $\to I$ (which generally does not require the two subformulas to be the same, of course) from intuitionistic natural deduction. Its classification as *intuitionistic* can be defined by its rules' right-hand sides using only a single formula; semantically, this definition makes sense (and can also be applied to sequent calculus) because such a restriction of the right-hand sides leads to the *tertium non datur* not being derivable in the logical system. This then is the underlying reason for why implication in intuitionistic natural deduction and its proofs directly correspond to the (first-class) function type and its *constructive* terms, a.k.a. the original Curry-Howard correspondence.

But what should the currently omitted rule be that we only marked by a "?", and what about the g? It does not seem to be a proof term assigned to a formula. Clearly, we must find a way to turn our sloppy made-up notation into something that can at least in a high-level way be semantically understood in our logical framework. There is a well-known answer to this question, but it is not a rule usually found in some system based on intuitionistic natural deduction, but rather a derivable theorem: the *substitution theorem* (or *substitution lemma*). For our purposes, it tells us that if the body of g, say $\psi_g(f)$, with f the functional parameter of g, typechecks as T_2 (in our example $T_2 = B$) in a context containing $f : T_1$ (in our example $T_1 = A \to A$), and there is some term t that we want to substitute for the x that typechecks as A, then the substitution of some t for f within $\psi_g(f)$ typechecks as T_2. Defining $g(t)$ to be $\psi_g[f \mapsto t]$, i.e. the substitution of t for f in the body ψ_g of g, we can thus finish our proof tree using this substitution theorem instead of a rule:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ x : A \vdash \phi(x) : A\end{array}}{\vdash \lambda x.\phi(x) : A \to A} \to I \qquad \begin{array}{c}\mathcal{D}_2\\ f : A \to A \vdash \psi_g(f) : B\end{array}}{\vdash g(\lambda x.\phi(x)) : B} \text{SubstTheorem}$$

In the sequent calculus, we have a rule with a form exhibiting the same principle idea of "cutting" out a formula appearing on the right-hand side of one premise and on the left-hand side of other: the Cut rule. And just as in the case of the substitution theorem for intuitionistic natural deduction, this rule is derivable. But it is nevertheless often made an explicit rule, or *internalized* in the system, for the practical "engineering" reasons mentioned. Hence our direct use of the theorem as a representation of a first-order function call is somewhat reasonable. With this initial understanding of the logical perspective on first-class and first-order functions, we can finally consider the transformation itself.

2.2.3 Defunctionalization to Cuts

In order to transform the proof tree, we first have to introduce the logical analogue of the data type, i.e. a set of rules for `Fun`. In our simple example, we have only one constructor without arguments and hence the following rule suffices:

$$\dfrac{}{\vdash \texttt{PlusOneFun} : \texttt{Fun}} \texttt{PlusOneFun}$$

We replace the use of the implication introduction with a use of this rule, just as we replaced the first-class function with the constructor call, and we replace occurrences of $A \to A$ with `Fun`.

$$\dfrac{\dfrac{}{\vdash \texttt{PlusOneFun} : \texttt{Fun}} \texttt{PlusOneFun} \qquad \begin{array}{c}\mathcal{D}_2\\ f : \texttt{Fun} \vdash \psi_g(f) : B\end{array}}{\vdash g(\texttt{PlusOneFun}) : B} \text{SubstTheorem}$$

Of course, we now also have to replace what corresponds to the function application in the body of g, i.e., in $\mathcal{D}_2$. More specifically, in that subtree, we have to insert the subtree $\mathcal{D}_1$ used as a premise for the implication introduction rule. Consider that we have $(f\ t)$ appearing somewhere in the body of g (where f is a parameter of g, hence our use of AXIOM), then we have a use of the implication elimination rule $\rightarrow E$ somewhere in $\mathcal{D}_2$.

$$\dfrac{\dfrac{}{f : A \rightarrow A \vdash f : A \rightarrow A}\ \textsc{Axiom} \qquad \begin{array}{c}\mathcal{D}'_2 \\ \vdash t : A\end{array}}{f : A \rightarrow A \vdash (f\ t) : A} \rightarrow E$$

This gets transformed to what corresponds to the first-order function call $\texttt{apply}(f, t)$, i.e. the Cut variation called the substitution theorem, in this case in the two-argument form; similarly to what we did with g above, we refer to the body of `apply` as $\xi(f, x)$, where f and x are the two parameters of `apply`.

$$\dfrac{\dfrac{}{f : \texttt{Fun} \vdash f : \texttt{Fun}}\ \textsc{Ax.} \qquad \begin{array}{c}\mathcal{D}'_2 \\ \vdash t : A\end{array} \qquad \begin{array}{c}\mathcal{D}'_1 \\ f : \texttt{Fun}, x : A \vdash \xi(f, x) : A\end{array}}{f : \texttt{Fun} \vdash \texttt{apply}(f, t) : A}\ \text{SubstTh.}$$

What remains to do is to define the subtree $\mathcal{D}'_1$ for ξ, where the mentioned $\mathcal{D}_1$ should go into. Note that ξ is the body of `apply`, which is constructed from the body ϕ of the lambda abstraction, and $\mathcal{D}_1$ is the subtree for ϕ. More precisely, the body ξ is a match with ϕ in the case for `PlusOneFun`, which logically corresponds to a variation of the elimination rule for disjunction, just as `Fun` is a variation of disjunction. Generally, for a type `Fun` with n constructors which have the argument type α_1, ..., α_n, respectively, this rule would look like this:

$$\dfrac{\Gamma \vdash \texttt{Fun} \qquad \Gamma_1, \alpha_1 \vdash B \qquad \ldots \qquad \Gamma_n, \alpha_n \vdash B}{\Gamma, \Gamma_1, \ldots, \Gamma_n \vdash B}$$

But since we only have a single constructor with no argument for `Fun` in this example, this degenerates to (leaving away the trivial $\alpha_1 = \top$ representing the empty argument list):

$$\dfrac{\Gamma \vdash \texttt{Fun} \qquad \Gamma_1 \vdash B}{\Gamma, \Gamma_1 \vdash B}$$

This rule we can use in $\mathcal{D}'_1$ to complete our proof tree with the proof tree $\mathcal{D}_1$ for ϕ:

$$\dfrac{\dfrac{}{f : \texttt{Fun} \vdash f : \texttt{Fun}}\ \textsc{Ax.} \qquad \begin{array}{c}\mathcal{D}'_2 \\ \vdash t : A\end{array} \qquad \dfrac{\dfrac{}{f : \texttt{Fun} \vdash f : \texttt{Fun}}\ \textsc{Ax.} \qquad \begin{array}{c}\mathcal{D}_1 \\ x : A \vdash \phi(x) : A\end{array}}{f : \texttt{Fun}, x : A \vdash \xi(f, x) : A}}{f : \texttt{Fun} \vdash \texttt{apply}(f, t) : A}\ \text{SubstTh.}$$

This completes our transformation of our example proof tree. The overall derivation, putting together the tree for the call to g with that of its body (seen in the upper part, connected with the dotted line), is shown in Fig. 2.4.

Summing up, we have eliminated all uses of implication rules at the cost of introducing a use of a new elimination rule, which in general is a variation of disjunction elimination, and using the Substitution Theorem a.k.a. Cutting. In particular, this means that we are now able to remove implication and its rules from our system, but have to introduce disjunctive rules into the system. This situation is analogous

$$
\dfrac{\dfrac{}{f : \texttt{Fun} \vdash f : \texttt{Fun}}\,\text{Ax.} \qquad \begin{array}{c}\mathcal{D}_2' \\ \vdash t : A\end{array} \qquad \dfrac{\dfrac{}{f : \texttt{Fun} \vdash f : \texttt{Fun}}\,\text{Ax.} \qquad \begin{array}{c}\mathcal{D}_1 \\ x : A \vdash \phi(x) : A\end{array}}{f : \texttt{Fun}, x : A \vdash \xi(f, x) : A}}{f : \texttt{Fun} \vdash \texttt{apply}(f, t) : A}\,\text{SubstTh.}
$$

$$
\vdots
$$

$$
\dfrac{\dfrac{}{\vdash \texttt{PlusOneFun} : \texttt{Fun}}\,\texttt{PlusOneFun} \qquad f : \texttt{Fun} \vdash \psi_g(f) : B}{\vdash g(\texttt{PlusOneFun}) : B}\,\text{SubstTheorem}
$$

FIGURE 2.4: Transformed proof tree ($\psi_g(f) = ...\texttt{apply}(f, t)...$; ξ is body of `apply`).

to that after defunctionalizing a program term, where we need a new data type made up of alternatives and we have to use the first-order `apply` function. Note that even though using the Substitution Theorem (corresponding to applying `apply`) appears to introduce a use of a (derivable) rule that has not been there before the transformation, the implication elimination it replaces is already also related to Cutting, which is a connection that depends on whether we are operating with intuitionistic natural deduction or the classical sequent calculus; this will become clearer in the next chapter.

2.2.4 Applications of de- and refunctionalization

To finish up this section, we take a look at applications of defunctionalization beyond the original Reynolds work (Reynolds, 1972). We also consider the partial inverse of defunctionalization, *refunctionalization* (Danvy and Millikin, 2009), and applications that have been found for it.

Reynolds (1972) had combined defunctionalization and CPS transformation to bring programs closer to machine code. What followed was work in the more general direction of inter-deriving *semantic artifacts*, which includes turning high-level code into code resembling various abstract machines or automata and syntactically described binary relations (Danvy and Nielsen, 2001; Danvy and Millikin, 2009; Danvy, Johannsen, and Zerny, 2011). For this, defunctionalization and CPS transformation were used, as well as, for moving in the respective other direction in the semantic artifact space, their opposites *refunctionalization* and *direct-style transformation*. We only vaguely call them opposites here since they, as originally presented, are not a perfect match, i.e. they are not full inverses of each other. There has been extensive work regarding the language fragments that one lands in after CPS or direct-style transformation from certain other language fragments. More important for the present work is the mismatch between defunctionalization and refunctionalization, which we consider in the next section. For now, we just give an impression of inter-deriving with one example featuring defunctionalization and another featuring refunctionalization.

As our example featuring defunctionalization, we consider the string recognizer for the language $\{0^n 1^n\}$ from Danvy and Nielsen (2001); their presentation is slightly simplified here and we use the idealized function and data type declaration and pattern match notation briefly introduced above (the original code is given in Standard ML). Their starting point is a recursive-descent parser `rec1`, which they then CPS transform, as shown in Figure 2.5. There is an auxiliary function called `walk` that traverses the input string (a list of bits). The idea behind it is that it *tracks how many*

```
function walk(xs : BitString, k : BitString → Bool) := match xs {
    Cons(0, xs')        ⇒ walk(xs', λxs.match xs {Cons(1, xs'') ⇒ k xs''; _ ⇒ false})
    xs                  ⇒ k xs
}

function rec1(xs : BitString) :=
    walk(xs, λxs'.match xs' {EmptyString ⇒ true; _ ⇒ false})
```

FIGURE 2.5: Recognizer for $\{0^n 1^n\}$.

```
data Stack { StackBottom ; StackPush(Stack) }

function apply(k : Stack, xs : BitString) := match (k, xs') {
    (StackBottom, xs')              ⇒ match xs' {EmptyString ⇒ true; _ ⇒ false}
    (StackPush(k), Cons(1, xs''))   ⇒ apply(k, xs'')
    (StackPush(k), _)               ⇒ false
}

function walk(xs : BitString, k : Stack) := match xs {
    Cons(0, xs')                    ⇒ walk(xs', StackPush(k))
    xs                              ⇒ apply(k, xs)
}

function rec1(xs : BitString) := walk(xs, StackBottom)
```

FIGURE 2.6: Recognizer, defunctionalized to a push-down automaton.

zeros it has seen by accumulating this state in the current continuation.[9] If it sees a 1, it just passes the current string to the current continuation. If it sees a 0, it recursively (thanks to the CPS transformation this is a tail call) walks over the rest of the string, with a continuation given that composes the current continuation with the request for seeing one more 1 in front, to match the 0 just seen. More precisely, the continuation given to this recursive call itself distinguishes between a 1 and a 0 as the first bit; in the 1 case, it passes the rest to the current continuation, and in the 0 case it has `walk` return `false` (since there is now an unmatched 0). This `walk` function is called in the body of the recognizer function `rec1`, passing over the input string and giving it a starting continuation that simply returns true if and only if its input is the empty string, which one can perceive of as representing the initial state with no zeros seen yet.

With the intuition behind the continuations representing the accumulated state, and the way they are composed, one might hazard a guess as to what the resulting semantic artifact after defunctionalization might be: a *push-down automaton*. And

[9] The original direct-style program has this accumulation happening implicitly on the call stack (it case matches on the recursive `walk` result) and uses an exception to escape the `walk` function in case of a mismatch.

```
data Nat { Zero ; Succ(Nat) }

function run(ps : Dyck, n : Nat) := match (ps, n) {
    (Empty, Zero)               ⇒ true
    (Empty, Succ(c))            ⇒ false
    (Cons(L, ps), c)            ⇒ run(ps, Succ(c))
    (Cons(R, ps), Zero)         ⇒ false
    (Cons(R, ps), Succ(c))      ⇒ run(ps, c)
}

function rec2(ps : Dyck) := run(ps, Zero)
```

```
data Nat { Zero ; Succ(Nat) }
data MaybeWord { None ; Some(Dyck) }

function run(ps : Dyck, n : Nat) := match (ps, n) {
    (Empty, c)                  ⇒ runAux(c, None)
    (Cons(L, ps), c)            ⇒ run(ps, Succ(c))
    (Cons(R, ps), c)            ⇒ runAux(c, Some(ps))
}

function runAux(n : Nat, mps : MaybeWord) := match (n, mps) {
    (Zero, Empty)               ⇒ true
    (Zero, Some(ps))            ⇒ false
    (Succ(c), None)             ⇒ false
    (Succ(c), Some(ps))         ⇒ run(ps, c)
}

function rec2(ps : Dyck) := run(ps, Zero)
```

FIGURE 2.7: Dyck recognizer before and after preprocessing.

that would be absolutely correct; taking the two functional abstractions that appear in the program and turning them into constructors for a data type leads to the simple stack type with a one-element alphabet, isomorphic to Peano natural numbers, shown in Figure 2.6. The starting continuation testing for the empty string becomes the bottom of the stack or the Peano zero, and the accumulating abstraction within `walk` becomes a constructor with one recursive argument, i.e. stacking one more element or the Peano successor. As usual, the bodies of the corresponding functional abstractions move to the `apply` function, also shown in Figure 2.6. This resembles a push-down automaton using the stack just described with two states, represented by the two functions `walk` and `apply`. The tail-recursive calls transition between the two functions (and hence the states) according to what symbol is seen next and what is on the stack at that time. The function `rec1` is the entry point redirecting to `walk` with the empty stack.

As our example featuring refunctionalization, we consider the Dyck word recognizer shown in Figure 2.7, i.e. a recognizer for a language over the alphabet {`L`, `R`} of left, i.e. open, and right, i.e. close, parenthesis that consists only of the well-balanced

words. This refunctionalization example is due to Danvy and Millikin (2009). The recognizer they start with is, with some squinting, recognizable as a push-down automaton, similarly to the result of defunctionalization we just saw. We again have a single-element alphabet stack, i.e. a counter, that reflects the number of open parentheses seen so far, for which we simply use the Peano natural number data type `Nat`. What we want to do now is see how the Dyck word recognizer could have resulted from a defunctionalization and then undo it, bringing it to a form that uses first-class functions; specifically, we want the `Nat` type to serve as our defunctionalized function type, and hence its constructors `Zero` and `Succ` to correspond to the lambda abstractions in the refunctionalization result. What we need for the transformation is a function that serves as the apply function, consuming and matching on `Nat`. In principle, we could use the `run` function for that; however, this would mean that we duplicated the third right-hand side since we would then need to split the third case in two by splitting the pattern variable c in $(\texttt{Cons}(\texttt{L}, ps), c)$ into the constructor patterns for `Zero` and `Succ`. Instead, we preprocess our program a bit, resulting in the program shown at the bottom of Figure 2.7. We introduce a new function `runAux` which takes care of the case distinction between `Zero` and `Succ` for the cases of `run` where it matters, i.e. in all cases except that for pattern $(\texttt{Cons}(\texttt{L}, ps), c)$. This is the only case where the first parenthesis, if any, is an `L`. Thus, in `runAux` we only need to distinguish between the empty word, where we do not need to pass further information from `run` to `runAux`, and non-empty words beginning with an `R`, where we pass along the suffix that follows the `R`. For this purpose we introduce a simple data type `MaybeWord` with constructors representing these two alternatives; one can think of `None` as standing for "all balanced, no leftover suffix", and of $\texttt{Some}(ps)$ as standing for "leftover suffix $\texttt{Cons}(\texttt{R}, ps)$".

The function `runAux` can serve as an apply function for `Nat`; refunctionalizing `Nat` with this apply function leads to the program shown at the top of Figure 2.8. Just like the starting point for our defunctionalization example, this program is in continuation-passing style. The `run` function now has a continuation k as one of its inputs corresponding to the `Nat` input before the transformation. If one were somehow able to get back to direct style, the program could conceivably have a rather nice high-level form, with just a few simple pattern matches and no stack or counter, which we already got rid of. Continuations are almost everywhere used linearly, hence bringing to direct style is not an issue here (Danvy, 1994), however the second right-hand side of `run` presents an obstacle: In the `None` case within the continuation passed to the recursive `run` call, the continuation is not used and instead `false` is directly returned; semantically, this makes sense because at this point we know there is an unmatched left (open) parenthesis and no right (close) parenthesis leftover to balance it with. Danvy and Lawall (1992) showed how to deal with this case by enlarging the target language fragment of direct-style tranformation with `callcc`, and Danvy and Millikin (2009) make use of this for this example: This return of `false` becomes a **throw** that interrupts the normal control flow and has to be caught (with **catch**) in the surrounding code.[10] In summary, refunctionalization helped us get a program from an automaton-like form to a more high-level one making essential use of a high-level control construct.

[10]Danvy and Millikin use a form of ML error handling with call/cc (Danvy and Lawall, 1992) for this purpose; here we just use the names "throw" and "catch" for the keywords to convey an intuitive idea of the control flow. This hopefully reminds the reader of comparable constructs in well-known languages, e.g. `throw` and `try-catch` in Java; note that Section 5.4.5 discusses a foundation for automatic conversion between this kind of ML error handling and Java exception handling that clarifies their relation.

```
data MaybeWord { None ; Some(Dyck) }

function run(ps : Dyck, k : MaybeWord → Bool) := match (ps, n) {
    (Empty, c)          ⇒ c None
    (Cons(L, ps), c)    ⇒ run(ps, λmps.match mps {None ⇒ false; Some(ps) ⇒ run(ps, c)})
    (Cons(R, ps), c)    ⇒ c Some(ps)
}

function rec2(ps : Dyck) :=
    run(ps, λmps.match mps {None ⇒ true; Some(ps) ⇒ false})
```

```
data MaybeWord { None ; Some(Dyck) }

function run(ps : Dyck) throws Bool := match ps {
    Empty           ⇒ None
    Cons(L, ps)     ⇒ match (run(ps)) {None ⇒ throw(false); Some(ps) ⇒ run(ps)}
    Cons(R, ps)     ⇒ Some(ps)
}

function rec2(ps : Dyck) := catch Bool {
    match (run(ps)) {None ⇒ true; Some(ps) ⇒ false}}
```

FIGURE 2.8: Dyck recognizer, refunctionalized and back in direct style.

2.3 The Extensibility Duality

Let us reflect upon what we saw in programs before and after defunctionalization, considering its relation to Cutting. As we saw in section 2.1 this is about reusing duplicate proofs and hence relates to code reuse, and more concretely corresponds to first-order function calls as demonstrated in section 2.2. Specifically, we now focus on an important property of (code) reuse: A reuse mechanism should ideally not make it necessary to break apart existing structure when introducing a new reusable instance.

Before defunctionalization, we are able to add new instances of first-class functions, i.e., new function abstractions, without touching existing code; afterwards, all bodies of function abstractions are in the first-order `apply` function, and if we wanted to add a new constructor of the `Fun` type we would have to add a case to the `apply` function. Thus, after defunctionalization, independently (without touching code like the case match in the `apply` function) adding a new `Fun` entity is not possible, whereas before adding a new function abstraction independently is possible. It is not as one-sided as it looks, however. For the `Fun` data type after defunctionalization, one may independently add new first-order functions that have a `Fun` as their input; before defunctionalization, the not independently extensible semantic equivalent are *destructors* (or *observations*) of so-called *codata*, which has multiple destructors instead of just `apply` (more on this idea soon).

Thus, defunctionalization actually switches between two dual extensibility tradeoffs, or *extensibility dimensions* as we will from now on refer to them. This connection was first explicitly pointed out by Rendel, Trieflinger, and Ostermann (2015), who used it to fix the invertibility issue of defunctionalization by generalizing first-class functions to *codata* types, the dual of data types. This result is a major prerequisite for the present work, thus we will consider it in some detail in this section. Note how, starting out with the connection between Cuts and code reuse and then more specifically first-order functions with which the program may be independently extended, one obtains a bigger picture which raises the possibility of a connection between the transformations and extensibility questions. Thus, arguably, this logic perspective paves the road for a fundamental explanation of the result uncovered by Rendel, Trieflinger, and Ostermann (2015). The end of this section provides an elaboration of this idea.

2.3.1 Problem: Defunctionalization lacks a total inverse

Consider the result of defunctionalizing the example from the previous section (see Fig. 2.3), which in particular included the following `apply` function:

```
data Fun { PlusOneFun }

function apply(f, x) := match f {
    PlusOneFun ⇒ x + 1
}
```

As we saw in the previous section, refunctionalization requires there to be such an `apply` function such that we can take it apart and move each of its cases' bodies into its own lambda abstraction. More precisely, the only thing required of such an `apply` in order to be usable for refunctionalization is the form of its signature, i.e. it should have two input arguments, one of which is of type `Fun`, the replacement for the first-class function type. When we only use refunctionalization to go back from

programs in the image of defunctionalization, everything works out fine. However, assume that we wanted to *change* the program before refunctionalizing it again. This is not something that is in general allowed because one has to be careful about two things:

- You cannot delete the `apply` function or modify its signature such that it does not have the described form anymore.
- You cannot introduce other functions which case match on `Fun`, like `apply` does, because the transformation would not eliminate them, but afterwards there would be no cases to match on anymore.

All these restrictions look rather ad hoc, and as Rendel, Trieflinger, and Ostermann (2015) point out, there are at least two reasons to generalize the transformations such that defunctionalization and refunctionalization are full inverses of each other. The first is the relative ease with which these transformations can now be automatized; the work of Rendel, Trieflinger, and Ostermann (2015) *almost* makes this just matrix transposition, and in the next chapters we will see how to eliminate the "almost" qualification. The second is the relation to the so-called Expression Problem: the generalized transformations switch between the two extensibility dimensions.

2.3.2 The Expression Problem

The duality of independently extending a program with new observations, or *consumers*, of some type and independently extending it with new *producers* of that type, and ways to achieve both simultaneously has been considered by various authors since at least the 1980s. The canonical example became what was called by Wadler the *Expression Problem* Wadler (1998). In its simplest form, we start with a recursive data type `Exp` that describes the abstract syntax tree (AST) for expressions in a simple term language, with a constructor `Num` for the number literal, for which we assume some natural number type `Nat`, and another constructor `Add` for the recursive addition node.

$$\textbf{data}\ \texttt{Exp}\ \{\ \texttt{Num}(\texttt{Nat})\ ;\ \texttt{Add}(\texttt{Exp},\texttt{Exp})\ \}$$

We can now write a simple recursive first-order function `eval` that consumes an `Exp` and computes its value (a natural number; we assume that addition + is defined on natural numbers).

$$\begin{array}{ll}
\textbf{function}\ \texttt{eval}(e:\texttt{Exp}):\texttt{Nat} & := \textbf{match}\ e\ \{ \\
\quad \texttt{Num}(n) & \Rightarrow n \\
\quad \texttt{Add}(e_1,e_2) & \Rightarrow \texttt{eval}(e_1)+\texttt{eval}(e_2) \\
\} &
\end{array}$$

In general, it is easy to add new functions consuming `Exp`; for instance, we could add a function `pretty` that takes an expression and computes its pretty-printed form. However, what if we want to add a new node to our AST, say, multiplication? We can add a new constructor `Mul` to the data type, but then we have to adapt all functions that consume and match on `Exp` to add a new case for `Mul`.

There have been many attempts to solve this problem to allow easy extension with both producers and consumers simultaneously (Krishnamurthi, Felleisen, and Friedman, 1998; Zenger and Odersky, 2001; Torgersen, 2004; Ernst, Ostermann, and Cook, 2006; Swierstra, 2008; Oliveira and Cook, 2012; Wang and Oliveira, 2016), in particular also in the presence of parametric polymophism and subtyping, as well as reflections on the criteria on when such a solution is reasonable, i.e. making precise what independent or "easy" extension is and what (type) guarantees are expected of a solution (Zenger and Odersky, 2004; Rendel, Brachthäuser, and Ostermann, 2014).

Here, however, we only want to look at the (idealized) language fragments which represent the two extensibility dimensions, i.e. the one in which extension with a consumer is easily possible and the one where extension with a producer is easily possible. We already saw that for the first dimension: for data types, one can easily add functions consuming data type arguments and matching upon them. In the literature on the Expression Problem, the other dimension is frequently represented by object-oriented programming.

In our example, instead of representing expressions by a data type, we could also have an interface for them in which we specify what operations on expressions are possible.

$$\textbf{interface}\ \texttt{Exp}\ \{\ \texttt{eval} : \texttt{Nat}\ \}$$

This interface can be implemented by a class, e.g. for `Num`, which amounts to give an implementation for each operation:

$$\textbf{class}\ \texttt{Num}(n : \texttt{Nat})\ \{\ \texttt{eval} := n\ \}$$

Similarly we can write another class implementing the interface for the recursive node `Add`. And we can now also easily add further producers of `Exp` like `Mul`, which we were not able to do when using a data type. However, easy extension by consumers is not possible anymore. If we wanted to add a pretty-printing operation `pretty` now, we would have to add it to the interface and then adapt all classes implementing the interface to implement `pretty` in them.

Summing up, we saw how programming with data types and case distinction corresponds to the consumer extensibility dimension while object-oriented programming corresponds to the producer extensibility dimension. Note that we only used an idealized subset of object-oriented programming, which is sufficient for this characterization and is precisely captured by the notion of *codata* (Hagino, 1989). Codata is also the language fragment that will serve as the target of the generalized refunctionalization; as hinted at, this transformation *is* switching from consumer to producer extensibility, and the same goes for generalized defunctionalization and the other direction.

2.3.3 Generalizing de- and refunctionalization

Consider again the question of what should happen with a modified defunctionalized program. If the modification puts it outside the domain of refunctionalization as originally defined, we cannot use this transformation. For instance, we might have added a second function consuming `Fun` (we also add a second constructor to make things a bit less degenerate):

```
data Fun { PlusOneFun ; MulTwoFun }

function apply(f, x) := match f {
    PlusOneFun          ⇒ x + 1
    MulTwoFun           ⇒ x * 2
}

function isPlusFun(f) := match f {
    PlusOneFun          ⇒ true
    MulTwoFun           ⇒ false
}
```

The new function simply checks if the given function is an addition function or not. Transforming this program with the generalization of refunctionalization presented by Rendel, Trieflinger, and Ostermann (2015), we obtain, in object-oriented jargon, not simply a first-class function abstraction for `PlusOneFun` and `MulTwoFun`, but rather an *object*, i.e. more specifically an instance of an anonymous class, which implements both the `apply` method and the `isPlusFun` method. That is, first-class functions are generalized to objects, and `apply` is now just one particular method; the type of first-class functions is now *defined* as a particular interface (e.g., as introduced in the standard library of Java 8). Carving out the essential aspect here, as hinted at before, we can present this as codata, and instead of having to resort to an object-oriented explanation (which shall only serve to provide some intuition as many are familiar with OO) we can rather directly and elegantly see how programming with codata is dual to programming with data.

```
codata Fun { apply(Nat) ; isPlusFun }

function PlusOneFun := comatch {
    apply(x)          ⇒ x + 1
    isPlusFun         ⇒ true
}

function MulTwoFun := comatch {
    apply(x)          ⇒ x * 2
    isPlusFun         ⇒ false
}
```

In this presentation, what we referred to as an interface is a declaration of a *codata type*, with a *destructor* signature corresponding to that of a method. Implementing the interface corresponds to giving *producer functions* (which we simply write using the keyword **function** as we did for consumer functions) for the codata type which implement each of its destructors, like a class would provide implementations of the methods. We refer to this destructor distinction as copattern matching (**comatch**)[11], due to Abel et al. (2013), who devised copatterns as patterns of observations dual to constructor patterns.[12] Note how this is similar to a case distinction for the different constructors of a data type. The difference is that the comatch does not have an entity that it comatch-es upon; rather, the implementation by copattern matching *constitutes* an entity by itself. On the other side, dually, the match absolutely *requires* some entity, it cannot stand by itself; in the next chapters we will set out to further align data and codata by overcoming such differences.

Strikingly, going back and forth between data and codata is now straightforward and can be seen as a simple rearrangement of the case implementations into functions. In our example, on the data side these are arranged such that $x + 1$ and $x * 2$

[11]The notation of Rendel, Trieflinger, and Ostermann (2015) does not use a **comatch** or **match** keyword since their language does not have *local* (co)matches, so they do not need these keywords; local (co)matches are discussed in the next section. Instead, the non-local setting enables Rendel, Trieflinger, and Ostermann (2015) to suggestively write the pattern match cases as *equations* (as known from top-level declarations in Haskell or ML), e.g. $\texttt{apply}(\texttt{PlusOneFun}(), x) = x + 1$. Viewing a program as consisting of equations might facilitate conceptually understanding extensibility switching (see below); this view of programs as collections of equations will also play a central role in the semantic considerations related to the work of Lämmel and Rypacek (2008) in Section 5.4.3.

[12]In the work of Abel et al., copatterns (and destructors) are in fact simply projections without arguments, but they treat first-class functions specially. This is not the case in the work of Rendel, Trieflinger, and Ostermann who allow destructor arguments in order to be able to capture first-class functions with codata (and more generally to make data and codata symmetric, of course).

	PlusOneFun	MulTwoFun
apply(x)	$x + 1$	$x * 2$
isPlusFun	true	false

FIGURE 2.9: Program matrix.

are together in one consumer function, and `true` and `false` are in the other. On the codata side, $x + 1$ and `true` are grouped together in one producer function, as well as $x + 1$ and `false`. Rendel, Trieflinger, and Ostermann (2015) not only pointed out their form of de- and refunctionalization was extensibility switching, but also that when presenting programs in the forms of two-dimensional matrices, where the two dimensions are the extensibility dimensions (Cook (1990) was presumably the first to explicitly present programs this way), then their transformations are simply transpositions of the program matrix.[13] Figure 2.9 shows the matrix for our example program. Reading it row-by-row results in the consumer functions of the data side, and reading it column-by-column results in the producer functions of the codata side, thus both *linearizations* of the program can be directly obtained from the matrix.

2.3.4 Transforming between Cuts for user-defined rules

When we considered the logic equivalent of defunctionalization in the previous section, we saw uses of implication elimination rules being replaced by Cutting (or more precisely, the Substitution Theorem), and uses of implication introduction rules by uses of introduction rules for the disjunctive `Fun`. Let us now consider what happens when we generalize the transformation, as presented in this section, to not just work with first-class functions, but arbitrary codata. To start, we repeat the implication introduction rule which corresponds to (first-class) functional abstraction:

$$\frac{x : A \vdash \phi(x) : B}{\vdash \lambda x.\phi(x) : A \to B} \to I$$

Anticipating the generalization to multiple destructors, let us first explicitly use the name `apply`; we also use curly braces to indicate that this is *some* kind of first-class abstraction and drop the λ.

$$\frac{x : A \vdash \phi(x) : B}{\vdash \{\texttt{apply}(x) \Rightarrow \phi(x)\} : A \to B} \to I$$

Now we can simply add another destructor `intrinsicProperty`, with a corresponding second premise, and obtain a rule that is structurally the same (modulo the parametric formulas) as the codata type `Fun` above which had a destructor `isPlusFun` in addition to `apply`.

$$\frac{x : A \vdash \phi(x) : B \qquad \vdash \psi : C}{\vdash \{\texttt{apply}(x) \Rightarrow \phi(x); \texttt{intrinsicProperty} \Rightarrow \psi\} : A \to^{*(C)} B} \to^{*} I$$

[13] Independently (but closely related), Harper (2016, ch. 26) discusses the two ways (method-based and class-based) to orient a program matrix in the context of dynamic dispatch (which he identifies as a "central concept" of "object-oriented" programming). In particular, Harper (2016, ch. 26) notes, agreeing (at least in principle) with what the present work advocates for: "A bias towards either a class- or method-based organization seems misplaced in view of the inherent symmetries of the situation."

We simply call the new ternary connective $\to^*$, indicating it is an enhanced implication. In the same way we can add introduction rules for arbitrary connectives with arbitrary destructors. Compare this to our generic rules from section 2.1.3 with which one could add arbitrary connectives to the sequent calculus system.

In sections 2.3.2, we saw the class `Num`, which served as an abstraction over all natural numbers n, each instance implementing `eval` to return the particular n. Presented in the codata way, a class is a producer function that possibly has some parameters, in this example the n. We saw that this is similar to consumer functions, albeit with the difference that consumer functions necessarily have one parameter that they consume, whereas there is no mandatory and distinguished producer function parameter. We will find a way to pull down this difference in the next chapter, but now we already nevertheless observe that the similarity of producer and consumer functions is sufficient to have their calls both correspond to the Substitution Theorem, just as we already demonstrated it for first-order functions generally. For instance, we can have a proof tree for $\to^*$ with a free variable/open premise y, for a formula C', that corresponds to a producer function that takes a C' as parameter and produces a value of the type corresponding to $A \to^{*(C)} B$ from it.

$$\dfrac{\begin{matrix}\mathcal{D}' \\ x : A \vdash \phi(x) : B\end{matrix} \qquad \begin{matrix}\mathcal{D}'' \\ y : C' \vdash \psi : C\end{matrix}}{y : C' \vdash \{\mathtt{apply}(x) \Rightarrow \phi(x); \mathtt{intrinsicProperty} \Rightarrow \psi\} : A \to^{*(C)} B} \to^* I$$

Calling this proof tree $\mathcal{D}$ and its corresponding term $\xi(y)$, we can reuse it with the help of the Substitution Theorem:

$$\dfrac{\begin{matrix}\mathcal{D}_y \\ \vdash y : C'\end{matrix} \qquad \begin{matrix}\mathcal{D} \\ y : C' \vdash \xi(y) : A \to^{*(C)} B\end{matrix}}{\vdash \xi(y) : A \to^{*(C)} B} \text{SubstTh}$$

Hence we saw that this follows the same principle that we saw for first-order functions generally.

Now, for both consumer and producer functions, each function is a particular instance of the respective reuse mechanism. As mentioned in the introduction of this section, a sensible reuse mechanism should allow for easy addition of new instances (independent extensibility), and producer and consumer functions exhibit this property, as we saw in section 2.3.2, as does a subtree in a proof that then uses the Substitution Theorem. Coming full circle, since reuse is also the idea behind Cutting, as presented in section 2.1.4, or using the Substitution Theorem, one can arguably expect that (the result of) defunctionalization should be related to the consumer reuse mechanism, as elaborated on in section 2.2.3, and the same goes for the refunctionalization result and the producer reuse mechanism, as we just saw. In particular, one can expect de-/refunctionalization to be related to the extensibility property of these reuse mechanisms. Thus, with our logic perspective on reuse that we built out starting from section 2.1.4, we have started to obtain a fundamental explanation of the initially surprising result of Rendel, Trieflinger, and Ostermann (2015) on the relation of extensibility and de-/refunctionalization. This is not intended to devalue *their* finding, but to *support* the argument for a more thoroughly systematic exploration of the logic corresponding to programming language aspects put forth in the introduction: If you have a good basis for fundamental explanations of important findings such as that of Rendel, Trieflinger, and Ostermann (2015), this basis can be reasonably expected to carry you further.

However, the perspective on reuse developed so far needs further refinement. As pointed out in section 2.1.4, we seek to reconcile intuitionistic natural deduction and sequent calculus to get the best symmetry out of both worlds, and will move towards this in the next chapter. Regarding the insights of this section, there are two (related) aspects that we will keep in mind as needing improvement:

- Consumer and producer functions, as presented until now, are inherently different. It was already remarked above how consumer functions have a distinguished mandatory argument that producer functions lack. Calling a consumer function corresponds to Cutting with terms, one of which is this distinguished argument. We can think of this term as a producer of some form, hence such a call is a meeting of producer and consumer. This is not the case for a producer function call, which is merely creating a producer that then still needs to meet a consumer to get computation going.

- The way a producer function call and a consumer, which is in this case a destructor, meet, is a destructor call corresponding to use of an elimination rule. Thus, on the producer function (codata) side, a meeting of producers and consumers is represented differently from the consumer function (data) side; there is no straightforward universally applicable representation for this meeting. In particular, a consumer function corresponds to a standalone proof tree, whereas a destructor corresponds to just the elimination rule itself and has no standalone representation. As we will see, this is also related to the connection between elimination rules and Cutting depending on the system, i.e. intuitionistic natural deduction or classical sequent calculus, as alluded to at the end of section 2.2.3.

As we will see, as a byproduct, the following section will provide more practical motivation for the quest for improved symmetry and hence economy of the next chapter.

2.4 Symmetric Language Design Using the Extensibility Duality

Rendel, Trieflinger, and Ostermann (2015) envisioned their data and codata fragments to form the core of a language, or rather development environment, in which one could switch extensibility at the press of a button whenever one had access to the whole program. This author was part of the effort to bring this idea closer to a real system, in particular designing a mechanically verified language with both data and codata with convenient local abstractions and integrating it into the desired development environment (Binder et al., 2019). The first part of this section considers this work in more detail and especially its impact on the present work. In the second part of this section we look at how to bring parametric polymorphism to the language fragments of Rendel, Trieflinger, and Ostermann (2015) in order to increase the practical relevance of this symmetric pair; this author co-authored the relevant paper (Ostermann and Jabs, 2018) and will also use the insight gleaned from that work for the parametrically polymorphic version of the system $\mathcal{PF}$ presented in the present work. This is a first example of a duality-utilizing exploration of the design space that goes beyond simple types; it also turns out to be a first candidate for a design prescription in that part of the design space.

```
data Nat { zero; succ(Nat) }
codata Stream { head : Nat; tail : Stream }

consumer function Nat.add(n) := {
    zero                              ⇒ n
    succ(m)                           ⇒ succ(m.add(n))
}

producer function repeat(n) := {
    head                              ⇒ n
    tail                              ⇒ repeat(n)
}

-- example function calls
zero.add(succ(zero))
repeat(succ(zero)).tail.head
```

FIGURE 2.10: Example program in the language of Binder et al. (2019).

2.4.1 A language with data and codata

The exposition at the very beginning of this work talked about programming languages forcing the programmer (or at least strongly favoring it) to structure their program in a certain way. Then the ultimate goal for this work was postulated: Aid the design of programming languages that, among other criteria, avoid this and allow for more diversity. The language (and environment) of Binder et al. (2019) can be seen as a prototype of such a language with inherent diversity. Concretely, it favors neither extensibility in one nor the other dimension, but rather allows the programmer to pick the dimension that fits best with their mental decomposition of the problem considered: one can either represent the relevant entities by a data type, or by a codata type, and these two approaches are symmetric. To that end, as mentioned, it builds upon the data and codata fragments of Rendel, Trieflinger, and Ostermann (2015). The following illustrates the decomposition diversity of Binder et al. (2019) with some examples reminiscent of what we have considered above in this chapter.

First of all, the language of Binder et al. (2019) allows to specify both data and codata types, together with consumer functions for data and producer functions for codata. An example of this is shown in Fig. 2.10. Here we have the usual natural number data type and a codata type for streams of natural numbers, together with a consumer function (the consumed type `Nat` for the matched-upon argument is written in front of the function name in the signature, separated by a dot: `Nat.add`) to add natural numbers and a producer function `repeat` for a stream that simply repeats the given number.

Just like the fragments of Rendel, Trieflinger, and Ostermann (2015), we can now switch the extensibility dimension of one of the two types (or of both). For instance, we can turn the `Stream` type into a data type with constructor `repeat` and consumer functions `head` and `tail`, or we can turn `Nat` into a codata type with destructor `add`

and producer functions `zero` and `succ`. Binder et al. (2019) implemented these transpositions in Coq and verified them to be total and full inverses of each other. From the Coq source a Haskell program was extracted which serves as the backend for a little IDE prototype in which transposition can be carried out at the press of a button.

An aspect that the data and codata language fragments lack, and which Binder et al. (2019) added, is the ability to *locally* specify matches and comatches. These are important to give programmers the convenience they are used to: With local matches, one can encode nested pattern matching; local comatches can be used to encode local lambda abstractions. As an example of the former, consider the Fibonacci function `fib` (here the argument list is empty, since the matched-upon argument is the only one):

```
consumer function Nat.fib() := {
    zero                ⇒ zero
    succ(zero)          ⇒ succ(zero)
    succ(succ(n))       ⇒ fib(succ(n)).add(n.fib())
}
```

While this cannot be expressed in the language of Binder et al. (2019), it can be directly encoded using a local match:

```
consumer function Nat.fib() := {
    zero                ⇒ zero
    succ(n)             ⇒
     match n {
       zero ⇒ succ(zero)
       succ(n0) ⇒ fib(n).add(n0.fib())
     }
}
```

As for local comatches, consider as an example that we want to lift addition of some number, e.g., 1, to a first class function in order to pass it to some higher-order function like `map` (l is some list assumed to be available in the context):

$$\ldots\ \texttt{map}(\lambda x.\ x.\texttt{add}(\texttt{succ}(\texttt{zero})), l)\ \ldots$$

The language of Binder et al. (2019) allows to encode this using a local comatch for the codata type `Fun` (**codata** `Fun` { `apply(Nat)` }):

$$\ldots\ \texttt{map}(\textbf{comatch}\ \texttt{Fun}\ \{\texttt{apply}(x)\ \Rightarrow\ x.\texttt{add}(\texttt{succ}(\texttt{zero}))\},\ l)\ \ldots$$

We now turn to the question of what these local abstractions mean for program transposition.

Program transposition is a global transformation that is arguably best understand as the transposition of a matrix where the field entries consist of the bodies of the cases of the top-level producer or consumer functions. The approach taken by Binder et al. (2019) is thus to temporarily lift local (co)matches to top-level functions and then perform transposition on the result. In order to be able to recover the original program with the local (co)matches when transposing again, the constructors or destructors that correspond to local abstractions are annotated as such. For example, the local comatch shown above becomes a constructor call where the constructor is annotated as local by a leading underscore:

$$\ldots\ \texttt{map}(\texttt{_addOneFun}(), l)\ \ldots$$

The codata type `Fun` is now a data type with this constructor (and potentially others for other comatches, local or top-level, in the program): **data** `Fun{_addOneFun}`. The name `addOneFun` needs to be invented somehow or be provided by the programmer;

for instance, it can be annotated to the relevant local (co)match. The body of the local comatch is now in the `apply` consumer function, as expected.

There is one technical complication with the transposition of local (co)matches that we have not considered thus far. It arises when we have free variables in the body of a local abstraction, i.e. variables not bound by that abstraction itself but somewhere above it. Consider again mapping the lifted addition function over some list, but this time not simply adding 1 but some variable number y with the variable bound somewhere else, say, in a top-level function f.

$$\ldots\ \texttt{map}(\textbf{comatch}\ \texttt{Fun}\ \{\texttt{apply}(x)\ \Rightarrow\ x.\texttt{add}(y)\},\ l)\ \ldots$$

What shall happen with this local comatch when the program is transposed? A possible approach could be to collect all the free variables within the body of the comatch and add corresponding arguments to the constructor that corresponds to the comatch. In the example, we would get a constructor call like this in place of the comatch:

$$\ldots\ \texttt{map}(\texttt{_addFun}(y), l)\ \ldots$$

And in the relevant case of the `apply` consumer function, we would have access to the argument of `_addFun`:

$$\texttt{_addFun}(y) \Rightarrow x.\texttt{add}(y)$$

However, there is a subtle problem with this. Consider that we want to compute the result of the term `_addFun(succ(zero).add(succ(zero)))`. It semantically is the same as `_addFun(succ(succ(zero)))`. This is nothing but a nesting of constructor calls, so there is nothing more to compute and this term is already our final result. But if we transpose a program (with respect to `Fun`) in which the original term appears, the corresponding term should have the variable y substituted with the argument of the constructor call, i.e. it should be

$$\textbf{comatch}\ \texttt{Fun}\ \{\texttt{apply}(x)\ \Rightarrow\ x.\texttt{add}(\texttt{succ}(\texttt{zero}).\texttt{add}(\texttt{succ}(\texttt{zero})))\}.$$

This term cannot be reduced further (unless when taking the unorthodox approach of reducing under binders, which would open a different can of worms). In summary, one term is reducible while its corresponding term under transposition is not, which is not an ideal scenario when wanting to establish a close correspondence between operational semantics before and after transposition.

As a remedy, Binder et al. (2019) introduce explicit substitutions tacked to the (co)match binders. This way the transformation exactly preserves the one-step reduction. In the example, the original constructor call becomes

$$\textbf{comatch}\ \texttt{Fun}\ \textbf{using}\ y := \texttt{succ}(\texttt{zero}).\texttt{add}(\texttt{succ}(\texttt{zero}))\ \{\texttt{apply}(x)\ \Rightarrow\ x.\texttt{add}(y)\}.$$

under transposition, and the right-hand side of the substitution (announced by the keyword **using**) is reducible just like the argument to the constructor call is. Overall, (co)matches are now really just local forms of producer/consumer functions, which especially means that *no binding crosses a binder*. And while one may view this as sacrificing a bit of convenience, it could also be argued that a deep and complicated binding structure is not necessarily particularly convenient anyway, and that in simple instances of binding across some fixed number of binders the explicit substitutions could be hidden by some surface syntax.

With this we have summarized the key aspects of this prototypical symmetric language design. A case study that recreates a semantic artifact interderivation of Danvy, Johannsen, and Zerny (2011) and demonstrates the relevance of working with the automatic transposition is found in the paper (Binder et al., 2019). The present work will revisit practical applications of transposition again in chapters 4 and 5 when discussing the more fundamental language $\mathcal{PF}$.

The paper of Binder et al. (2019) also contains a paragraph, mainly written by this

author, with some initial thoughts on how to further improve the symmetry. The language $\mathcal{PF}$ presented herein is the eventual result of the design process that began there. One indication that intuitionistic natural deduction is not the optimal frame in which to draw up a symmetric language when considering fundamental properties of the language and the transformations was that this author and his coauthors had to always take care of the structural difference between the data side and the codata side when proving such properties. Concretely, structurally a constructor takes some arbitrary number of arguments, whereas a destructor also takes these arguments, but additionally has an *output* type specified. This structural difference presents an obstacle to easy formal treatment, which this author thinks supports aiming for data and codata to be uniform such that definitions and proofs can also be structurally identical. $\mathcal{PF}$ exhibits such structural equivalence of data and codata, leading to a rather elegant formal treatment, as we will see in chapter 4. Chapter 5 presents macro embeddings for natural deduction-framed surface languages, thus demonstrating that $\mathcal{PF}$ is indeed more general than the language of Binder et al. (2019), and that formal treatment of such surface languages can be factored into a treatment for the relevant macro embedding and the one for $\mathcal{PF}$.

Chapter 4 also presents a version of $\mathcal{PF}$ that includes parametric polymorphism, which was informed by a prior work which this author co-authored and which is summarized in the next subsection.

2.4.2 Symmetric spots in the type system design space

As mentioned in Section 2.2.1, Pottier and Gauthier (2004) realized that defunctionalizing parametrically polymorphic first-class functions requires Generalized Algebraic Data Types (GADTs) in the target language. This means that to find a spot in the type system design space stable under extensibility switching, data types have to be generalized to GADTs when allowing producer functions to have type parameters. Now, if we want consumer functions to have type parameters as well, exactly mirroring the generalization of producer functions, it turns out that what we need is a generalization of codata types that is the dual of GADTs (Ostermann and Jabs, 2018), which my coauthor of the relevant paper, Klaus Ostermann, and I decided to call *Generalized Algebraic Codata Types* or short GACoDTs, until a better name surfaces. Notably, this kind of type was already invented at the time in the context of Object-Oriented Programming, integrated into a research extension of C# (Kennedy and Russo, 2005). The following summarizes the key aspects of the GA(Co)DT language of Ostermann and Jabs (2018), especially the matrix formalism employed, by means of some examples, before closing this chapter with some reflections on the type system design space.

Let us start with the codata encoding of first-class functions, now finally parametric in the input type A and output type B:

$$\textbf{codata}\ \texttt{Fun}\langle A, B\rangle\ \{\ \texttt{Fun}\langle A, B\rangle.\texttt{apply}\langle A, B\rangle(A) : B\ \}$$

For each destructor (like `apply`), the instantiation of the type parameters is given in the list between the angle brackets before the dot ($\texttt{Fun}\langle ...\rangle$.). In the `apply` example, the instantiation is generic, thus we specify it with the variables A and B; such variables are bound in the list between the angle brackets following the destructor name ($\texttt{apply}\langle ...\rangle$). As a simple parametric producer function for this type, consider

the producer for the identity function:

$$\textbf{producer function}\ \texttt{id}\langle A\rangle() : \texttt{Fun}\langle A, A\rangle := \{$$
$$\texttt{apply}(a) \quad \Rightarrow a$$
$$\}$$

Let us add the addition of 1 lifted to a first-class function and present all of that as a program matrix for the codata type Fun:

	$\texttt{Fun}\langle A, B\rangle.\texttt{apply}\langle A, B\rangle(a : A) : B$
$\texttt{id}\langle A\rangle : \texttt{Fun}\langle A, A\rangle$	a
$\texttt{add1} : \texttt{Fun}\langle\texttt{Nat}, \texttt{Nat}\rangle$	$\texttt{add}(a, 1)$

The columns correspond to destructors and the rows to producer functions. We can switch the extensibility dimensions by transposing that matrix, obtaining the matrix shown below.

	$\texttt{id}\langle A\rangle : \texttt{Fun}\langle A, A\rangle$	$\texttt{add1} : \texttt{Fun}\langle\texttt{Nat}, \texttt{Nat}\rangle$
$\texttt{Fun}\langle A, B\rangle.\texttt{apply}\langle A, B\rangle(a : A) : B$	a	$\texttt{add}(a, 1)$

Here the columns correspond to constructors and the rows to consumer functions. We see that the constructor signatures specialize the parameters of Fun, so the data type associated with that matrix is a GADT. Using the matrix presentation it becomes obvious how the constructors resulting from defunctionalization have the same signatures as the original first-class functions and that defunctionalizing them thus requires GADTs in the target.

The apply consumer function is parametric, but does not specialize the parameters of the data type Fun of the consumed data, unlike the producer functions we considered. If we allow that same kind of type-theoretic strength, we can for instance write a consumer function add? that checks if a given function (i.e. a value of type Fun) from Nat to Nat is an addition function (this consumer is rather trivial, but adding other functions from Nat to Nat can easily make this example more relevant):

$$\textbf{consumer function}\ \texttt{Fun}\langle\texttt{Nat}, \texttt{Nat}\rangle.\texttt{add?}() : \texttt{Bool} := \{$$
$$\texttt{id} \quad \Rightarrow \texttt{false}$$
$$\texttt{add1} \quad \Rightarrow \texttt{true}$$
$$\}$$

Now, when we consider the matrix form of the resulting program and transpose that, we see that the destructor signature for add?, which is the same as the signature of the original consumer function, specializes the parameters of the destructed codata type to Nat and Nat.

	$\texttt{Fun}\langle A, B\rangle.\texttt{apply}\langle A, B\rangle(a : A) : B$	$\texttt{Fun}\langle\texttt{Nat}, \texttt{Nat}\rangle.\texttt{add?} : \texttt{Bool}$
$\texttt{id}\langle A\rangle : \texttt{Fun}\langle A, A\rangle$	a	false
$\texttt{add1} : \texttt{Fun}\langle\texttt{Nat}, \texttt{Nat}\rangle$	$\texttt{add}(a, 1)$	true

In other words, that destructor add? may only be applied to certain codata values with the appropriately instantiated parameter types; specifically in this example, it cannot be applied unless both parameters are Nat. Generally, just as we need a generalization of data types, namely GADTs, when transposing producer functions which are parametrically polymorphic and allowed to arbitrarily specialize type parameters, since the resulting constructor signatures are identical to those of the producer functions, we also need a generalization of codata types when giving consumer functions the same liberty; as mentioned, we preliminarily chose to name this generalization GACoDT.

Previously we saw how codata can be seen as the essence of Object-Oriented Programming. In OO terms, a codata type can be understood as an interface, but

in ordinary OO languages one cannot specify that some method in an interface is only allowed (and required to be implemented) in implementations of the interface that pick certain types for the parameters as in a GACoDT. However, there has been some research on how to bring exactly such a kind of construct to C# (Kennedy and Russo, 2005), who extended the **where**-clause of C#, which already allowed to specify subtyping constraints on the parameters, to also allow equational constraints.

```
interface Fun⟨A, B⟩ {
    B apply(A);
    Bool add?() where A = Nat, B = Nat;
}
```

In the syntax of Kennedy and Russo (2005), we could write the interface shown above which corresponds to our codata type Fun. Unlike in the case of Kennedy and Russo, who were evaluating a rather ad-hoc linguistic extension inspired by GADTs, we argue that our "rediscovery" of GACoDTs was a direct *necessary consequence* of our intent to extend both producer and consumer functions in the same way and keep the system stable under matrix transposition.

Whenever the parameter specializations of a destructor and producer function or of a constructor and consumer function are incompatible, it is impossible to fill the respective matrix field. As an example, consider adding a (producer function for a) first-class function toStr that has Str(ing) as its output type. This is not compatible with destructor add? which requires that output type to be Nat, which is indicated by graying out the respective matrix field.

	$\texttt{Fun}\langle A, B\rangle\texttt{.apply}\langle A, B\rangle(a : A) : B$	$\texttt{Fun}\langle \texttt{Nat}, \texttt{Nat}\rangle\texttt{.add?} : \texttt{Bool}$
$\texttt{id}\langle A\rangle : \texttt{Fun}\langle A, A\rangle$	a	false
$\texttt{add1} : \texttt{Fun}\langle \texttt{Nat}, \texttt{Nat}\rangle$	$\texttt{add}(a, 1)$	true
$\texttt{toStr} : \texttt{Fun}\langle \texttt{Nat}, \texttt{Str}\rangle$	$\texttt{nativeToStr}(a)$	

Of course, in the linearized forms of the program, the cases in producer or consumer functions corresponding to such grayed out fields can simply be omitted. The notion of compatibility considered here is structurally the same as in Hindley-Milner type systems (Hindley, 1969; Milner, 1978), and can accordingly be checked by a unification algorithm like that of Robinson (1965). The type system of the GA(Co)DT language, including the proper use of unification, was proven to be sound (with the usual progress and preservation theorems) in Coq (Ostermann and Jabs, 2018). The essence of the polymorphism aspect of the proof is also found in the soundness proof for the polymorphic variant of $\mathcal{PF}$ in Section 4.3, which was informed by this prior work.

We have seen two symmetric spots stable under transposition and giving the same type-theoretic strength to both the data and codata side constructs. One is the language of Binder et al. (2019), or the language fragments of Rendel, Trieflinger, and Ostermann (2015) that it is based upon, for the simply typed case, and the other is occupied by the GA(Co)DT language of Ostermann and Jabs (2018). As outlined above, if one wants to add parametric polymorphism at all and arrive at a spot in the design space that exhibits such stability, the minimal language is this GA(Co)DT language. In the future it could be interesting to see what other symmetric spots there are and if it will be possible to re-explore all that we currently know about the type system design space, important aspects of which have been illustrated by the so-called lambda cube (Barendregt, 1991), through the lens of the extensibility symmetry; it could also be interesting to consider this for the foundational system $\mathcal{PF}$ presented in Chapter 4.

There are some preliminary suggestions for the design of programming languages with non-simple types that can be gleaned from the one symmetric spot

considered already. First, it lends additional support for further pursuing experimental languages that contain some form of GACoDTs like that of Kennedy and Russo (2005). Second, it shines new light on the importance of GADTs. In some languages like Haskell, GADTs are relegated to extension status, but the result of refunctionalization is in the core language, which suggests that a reevaluation of what is essential and what is not might be in order. Finally, it is the author's impression that the best polymorphic language design that allows for diversity of decomposition is one that has both GADTs and GACoDTs, that is, it is fundamentally guided by the GA(Co)DT language just summarized. Diversity of decomposition methods available in a language that is not biased by giving one decomposition more powerful type system features than the other requires the language to occupy a spot like that of the GADT / GACoDT pair.

Chapter 3

From the De-Morgan Duality to Dialogue-Inspired Systems

This chapter discusses De Morgan duality, the way it is embodied by sequent calculus, and, most importantly, calculi for programming in the sequent calculus, to the extent they are relevant for this work. Such calculi based on the sequent calculus are different in nature from those based on natural deduction, as is the case for most "normal" programming languages. Specifically, they inherently require the "programmer" to be more explicit when it comes to *control flow*, similarly to continuation-passing style (CPS). In case the reader is looking for a "concrete" intuition (pun intended) before we get to the details, Wadler (2003) compared the nature of this more symmetric setting to modern architecture styles: "[...] like the Pompidou Center in Paris, the plumbing is exposed on the outside. While this can make the expression harder on the eyes, it also – like CPS, and like the Pompidou Center – has the advantage of revealing structure that previously was hidden."

Chapter overview The chapter is structured as follows:

- Section 3.1 sets the stage for De Morgan duality.
- Section 3.2 discusses previous efforts of turning sequent calculus into an (unusual) programming language, focussing mainly on the work of Downen (2017) whose thorough analysis assembles previous attempts into one coherent system, and why these, rather directly following the structure of sequent calculus, have certain drawbacks which make them less suited for the foundational system we seek.
- Section 3.3 discusses an alternative system with *polarized* types, the *Calculus of Unity* (Zeilberger, 2008b). Especially, *polarity* relates to the data/codata dichotomy discussed in the previous chapter and the Calculus of Unity is inherently highly symmetric in both this and the previous chapter's sense, which is why it will serve as the basis for the foundational system introduced in the next chapter. A potentially useful intuition for polarity is how logical proofs and refutations, and following that, data construction and codata destruction in programming languages, can be conceived of as *dialogues* between two "players"; ideas for how the foundational system could possibly be generalized under such a *game semantics* perspective can be found in the outlook in Chapter 6.

3.1 The Negation Mirror

Throughout the previous chapter, it was highlighted how logicians and programmers are doing essentially the same when it comes to engineering concerning reuse

(DRY). Of course, ultimately this all was following the tracks of the now classic Curry-Howard correspondence: *programs as proofs*. Now, from the practical programmer's perspective it is rather clear that we are not always only interested in proving certain properties, but also in *refuting* certain (other) properties. It arguably is more natural to describe what some test engineer wants to achieve as, e.g., refuting that some machine makes a fatal error rather than proving that the machine does *not* make a fatal error. Why use the positively connoted phrase *prove* plus the negatively connoted *not* when one can just use the negatively connoted dual *refute* of *prove*?

In classical logic, a refutation may of course always be simulated by a proof of a negation, however, this does not hold true for other forms of logic and, as we just saw, this simulation is arguably at the expense of linguistic economy (when assuming that the word *refute* is available anyway). More essentially, and what this short introductory section wants to argue for, is that there is no *inherent* reason to favor proof over refutation (or vice versa). Just as a refutation may be classically simulated by proving a negation, one could just as well simulate a proof by refuting a negation. The focus on either proof or refutation is very much a conscious decision made by the programmer/logician while thinking about a certain problem.

Let us go back to the introduction rules for *and* ($\wedge$) and *or* ($\vee$) we saw at the beginning of the previous chapter.

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

There it went almost without saying that these rules are for proving formulas involving the respective connective rather than refuting them. Now, instead of relying on such an easily misunderstandable contextual communication, let us make explicit that we are considering rules for proving things by marking each formula with *true*; i.e. we enhance the judgment from being just the formula itself to being the formula together with the judgmental qualifier *true*.

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \qquad \frac{A \text{ true}}{A \vee B \text{ true}} \qquad \frac{B \text{ true}}{A \vee B \text{ true}}$$

So far, this seems to only have made things more verbose. However, if we now additionally introduce a refutation judgment (*false*) and rules for it, the clarification becomes more obviously useful. This approach of having a separate judgment per kind of proof object, e.g. proof and refutation, is called the *judgmental method* (Martin-Löf, 1996; Pfenning and Davies, 2001); it will become even more relevant over the course of this chapter.

$$\frac{A \text{ false} \quad B \text{ false}}{A \vee B \text{ false}} \qquad \frac{A \text{ false}}{A \wedge B \text{ false}} \qquad \frac{B \text{ false}}{A \wedge B \text{ false}}$$

Looking at these concrete refutation rules, we observe that they are isomorphic to the proof rules from before: Just exchange $\vee$ and $\wedge$! Proving a conjunction requires proving both its subformulas, but refuting a conjunction requires only refuting one of them, and it is exactly the other way around with disjunction. Of course, the heart of this observation is not due to this author or any contemporary work, even though the more modern, judgment-rich presentation will be quite useful and important for the rest of this chapter. Rather, it probably goes back to time immemorial, and is now conventionally attributed to De Morgan.

In his proof-biased presentation, where a refutation is expressed via a negation as usual, the relation between $\vee$ and $\wedge$ is expressed by the following equations, which are called the *De Morgan laws*:

$$\begin{aligned}\neg(A \vee B) &= (\neg A) \wedge (\neg B)\\ \neg(A \wedge B) &= (\neg A) \vee (\neg B)\end{aligned}$$

In this presentation, similarly to the one we saw before, one can obtain the first equation from the second and the second from the first by simply exchanging $\vee$ and $\wedge$. Opting for a radical perspective in the interest of parsimonious language design, we can even go so far as to say that $\vee$ and $\wedge$ are *the same connective* until the previously mentioned conscious decision has been made on whether one cares about proofs or refutations in a certain setting: both connectives spring forth from the same set of (three) rules (as far as introduction rules are concerned) when just considering their structure. More conservatively phrased, the two connectives are *dual* to each other. Going from one to the other is induced by the switch between proof and refutation, which are the underlying dual concepts, i.e. the duality of $\vee$ and $\wedge$ just mentioned is the one of the proof-refutation pair; in the previous chapter we have seen a different duality, the extensibility duality, and by the next chapter we will have obtained a system in which both dualities naturally appear.

Both for simulating proofs with refutations and for simulating refutations with proofs we saw the same connective used: negation ($\neg$); that is, if we were looking for a connective with which to capture the dual pair proof-refutation, $\neg$ would be a good contender. One could say the proof-refutation duality is *the* $\neg$-duality, and $\neg$ is what mediates between two sides of an entirely symmetric mirror; the De Morgan laws succinctly express this mediation relation. Focussing on the structure of the rules in the context of this mirror, consider the $\vee$ introduction rules with just the structure remaining, i.e. without the information that they are for proving $\vee$. We cannot distinguish them from the rules for refuting $\wedge$ (and the same applies for proving $\wedge$ and refuting $\vee$). Looking through the mirror we see the structure of the rule for proving $\wedge$ (the rule for proving $\vee$), which is the structure dual to the rules (the rule) we have on our own side. Only when taking both sides of the mirror into account can we purely *structurally* apprehend the entire system with two connectives and two judgments. Therefore, it makes sense to regard a central role of $\neg$, as a representation of the proof-refutation switch, as essential for our desired design parsimony. As we will see, it will replace implication, which often plays a central role in programming languages and logical systems, but lacks the symmetric properties of $\neg$ outlined in this section. Implication can be recovered, similarly to how codata types allow to define the function type, but does not need a special status. This status was often more due to biases, anyway, rather than due to any (intrinsic or extrinsic) motivations.[1]

[1] An example of how this bias influenced the thinking of researchers when it comes to comparing CBN and CBV is given by Levy (2001). Levy refutes the suggestion that "CBN is "mathematically better behaved but practically less useful" than CBV" by showing that "although the CBN function type is mathematically superior to the CBV function type, the CBN sum type (and boolean type) is inferior to the CBV sum type." Regarding the bias towards function types, he remarks (emphasis added): "Thus, the perception of CBN's superiority is actually due to the fact that *function types have been considered more important, and hence received more attention*, than sum types or even ground types."

$\vdash A$	Prove A
$A \vdash$	Refute A
$\vdash$	Contradiction

FIGURE 3.1: Sub-judgments for proof, refutation, contradiction.

3.2 Programming in the Sequent Calculus

3.2.1 Sequent calculus embodies the negation mirror

A system that is not biased towards proof or refutation is the classical sequent calculus we saw in section 2.1. Using the mirror analogy on the sequents, we can say the turnstile plays the role of the mirror. Consider the sequent $A, B \vdash C, D$. Remember that the left-hand side is interpreted conjunctively and the right-hand side is interpreted disjunctively, hence the turnstile that sits between the two sides relates the two interpretation similarly to how negation relates the conjunction and disjunction connectives. One can read $A, B \vdash C, D$ as: If A and B are true, then either C or D or both must be true. However, the sequents are not biased towards such a proof-oriented reading, and we can just as well read it as: If C and D are false, then either A or B or both must be false. One can easily convince oneself that the two readings are logically equivalent: If it is the case that from the truth of both A and B follows the truth of (at least) one of C and D, then if both C and D are false it cannot have been the case that both A and B were true.

Of course, the approach generalizes to arbitrary sequents $\Gamma \vdash \Delta$ with more than two formulas on either side. The most radical cases arise when either Γ or Δ is empty. $\vdash \Delta$ is very familiar, it just means that no hypotheses are needed to prove (the disjunctively interpreted) Δ. $\Gamma \vdash$, on the other hand, looks rather strange at first sight, but has completely natural readings, though one is more straightforward. The proof-oriented one is: If all of Γ is true, then *something* in the empty set must be true. The conclusion of this conditional sentence is of course impossible, hence its premise must have been false, so there is a formula in Γ that is false. And this brings us to the more straightforward reverse reading, the refutation-oriented one: If everything in the empty set is false, then something in Γ must be false. Since the condition is vacuously tautological (universal statements about members of the empty set are always unconditionally true), the implication sentence is equivalent to just its conclusion. Thus the arguably most straightforward reading of $\Gamma \vdash$ is a pure refutation: Something in Γ is false.

Specializing to one formula on one side and zero formulas on the other, one obtains the sub-judgments shown in Figure 3.1 for proof and refutation, as well as the *contradiction* judgment with zero formulas on both sides with the readings: If something in the empty set is true, then the empty set contains at least one true formula; if something in the empty set is false, then the empty set contains at least one false formula. Of course, both theses sentences are obviously false (because the empty set does not contain anything), justifying to call it the contradiction judgment and demanding that there is no way to derive it; we have briefly considered cut elimination as a way to show this property in section 2.1, and will soon return to it. Overall, these three sub-judgments will be instrumental in the following sections.

In this section, we consider the relation between the proof and refutation sub-judgments and the dichotomy of left and right rules briefly touched upon in section 2.1. Based upon this, we then consider term assignments for the sequent calculus

and the problems arising when trying to find one that is suitable for programming "in the sequent calculus". Specifically, we compare it to the ease with which one can find such a term assignment when restricting oneself to intuitionistic, or constructive, logic, in particular the natural deduction breed. What will be crucial in these considerations is the role of *reduction* and *locality of reduction* as a central property of practically usable programming languages. Intuitionistic natural deduction obeys such a property, but it is not as well suited for our symmetry purposes as the classical sequent calculus. Thus this section will close with putting forth the goal of reconciling the two, which was hinted at in section 2.1, and which will be made possible by the observations of the last section of this chapter.

3.2.2 Refutation/proof and left/right rules

As pointed out in section 2.1, all rules in sequent calculus are introduction rules, i.e. they introduce a connective on either the left-hand side or the right-hand side. The rules where the connective appears to the left of the turnstile are called *left rules*, and the rules where the connective appears to its right are called *right rules*. An example of a left rule is the first left rule for conjunction (from Gentzen's LK, see Fig. 2.1):

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge L_1$$

With the understanding of proof and refutation sub-judgments we just developed, where we saw that having a formula to the left of the turnstile can be desired as refuting it, we can give the following simplified gloss of this rule: If we can refute A in some circumstances, as expressed by the Γ and the Δ, we can also refute $A \wedge B$ in these same circumstances. More precisely, the circumstances are that all of Γ is true and that all of Δ is false. In other words, the refutation is *contingent* since it happens modulo the truth values of Γ and Δ, but we can straightforwardly describe the left rule as a refutation rule nevertheless, and the same goes by a similar argument for all the left rules. We consider one more example due to its special role that we explored in the previous parts of this chapter: the left rule for negation.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg L$$

What is interesting here is that in the premise the subformula A of the negation appears not on the left-hand side, as it is with the subformulas of conjunction and disjunction, but on the right-hand side. Intuitively, this rule makes sense since having a contingent proof of A means that one also has a contingent refutation of the negation of A. This also demonstrates how in a certain sense the $\neg$ connective internalizes the meaning, i.e. the relation to the parts of the sequent, of the turnstile itself: Formulas can always simply wander across the turnstile by wrapping them in a negation, as the left-hand side and the right-hand side are $\neg$-dual to each other, just as the two sides of the negation mirror discussed in section 3.1.

A completely analogous argument applies for right rules and the proof sub-judgment. Overall, we have seen how we can conceive of left rules being concerned with refutation, contingent upon formula contexts on the left-hand side and right-hand side, and of right rules being concerned with contingent proofs. We can thus enhance our sub-judgments and suggestively rearrange the respective parts of the sequents and add explicit qualifications, as shown in Figure 3.2, to give a first approximation of a formal embodiment of these intuitive ideas. This is actually already rather close to the system we will eventually arrive at, as we will see in section

Sequent	Rearranged	Meaning
$\Gamma \vdash A, \Delta$	$\Gamma; \Delta \vdash A$ true	Prove A
$\Gamma, A \vdash \Delta$	$\Gamma; \Delta \vdash A$ false	Refute A
$\Gamma \vdash \Delta$	$\Gamma; \Delta \vdash$ contradiction	Contradiction

FIGURE 3.2: Sub-judgments for contingent proof, refutation, contradiction.

3.4. However, we have not yet considered if the structure of these proofs is in any way suitable for a programming language under a Curry-Howard-like reading, and we will now explore this issue, utilizing the intuitions for reading off sequents in a proof-like or a refutation-like way and our first approximation of a formal system for this.

3.2.3 Term assignments for the sequent calculus

When we think about a term assignment, we need to consider what this means for the computation embodied in the terms. There is a long line of theoretical work on this topic. Since the present work is primarily concerned with finding a consolidated system for both the extensibility duality and the negation duality, and especially with how the latter can inform a highly symmetric perspective of the former, an in-depth discussion of this historical development is beyond its scope; what follows is a (very) quick summary of the most important works. The rest of this section will then follow Downen's thorough analysis of a programming language for the sequent calculus (Downen, 2017), which builds upon these and arguably forms a preliminary end point of this line of work.

On the road towards programming in the sequent calculus, the $\overline{\lambda}\mu\tilde{\mu}$-calculus (Curien and Herbelin, 2000) has proven to be highly influential. It was itself influenced by the $\lambda\mu$-calculus (Parigot, 1992), an early system that paved the road for widening the Curry-Howard program by relating non-linear control flow and classical logic, albeit in natural deduction style, in a straightforward manner. The dual μ and $\tilde{\mu}$ control operators will be of particular interest in this section. Logically, μ allows to prove a statement by leading all its possible refutations to contradictions, while $\tilde{\mu}$ allows to refute a statement by leading all its possible proofs to contradictions. Computationally, μ is related to call-by-name evaluation, since it allows to encapsulate and thus delay a computation, while $\tilde{\mu}$ is related to call-by-value evaluation, since it encodes a continuation that (via the rules of $\overline{\lambda}\mu\tilde{\mu}$) necessarily triggers evaluation (only a value can be "passed" to $\tilde{\mu}$). The equational theory of a system with these two operators allows to make formally precise the statement that call-by-value is dual to call-by-name, a result that was independently achieved for a slightly different, but likewise sequent calculus based system by Wadler (2003).

The starting point for the programming language for the sequent calculus of Downen (2017) is a core system with only the μ and $\tilde{\mu}$ operators, due to Herbelin (2005). In section 2.1 we saw how to separate sequent calculus into two modules, one for CUT and one for user-definable rules for connectives. It is the first of these that forms the core of Downen's system. The key idea behind the reduction rules for this core part is that reduction is somehow related to elimination of cuts. We have seen this idea at play in the previous chapter, albeit with the Substitution Theorem of intuitionistic natural deduction rather than the CUT rule of classical sequent calculus. Getting rid of a use of the CUT rule is a local version of cut elimination in the

$$\frac{\Gamma | \, e : A \vdash \Delta \qquad \Gamma' \vdash v : A | \, \Delta'}{\langle v \| e \rangle : (\Gamma, \Gamma' \vdash \Delta, \Delta')} \text{ CUT}$$

$$\frac{}{| \, \alpha : A \vdash \alpha : A} \text{ AXIOM L} \qquad \frac{}{x : A \vdash x : A \, |} \text{ AXIOM R}$$

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

FIGURE 3.3: Downen's core sequent calculus language.

sense that combining these local eliminations ultimately adds up to the elimination of cuts entirely, and hence reduction can be characterized as the entity that the CUT module should come equipped with in order to provide a consistency proof for the system.[2]

Downen's (and Herbelin's) approach is somewhat similar to the characterization of the sub-judgments we saw in section 3.2.2. What he does is split the judgment into three judgments, which correspond to our sub-judgments as far as it concerns the core system; here c, v, and e are metavariables for syntactic expressions and each in formula in the contexts Γ and Δ is annotated with a variable, named x_i for Γ and α_i for Δ.

$$\Gamma \vdash v : A \mid \Delta$$
$$\Gamma \mid e : A \vdash \Delta$$
$$c : (\Gamma \vdash \Delta)$$

However, in Downen's full system including the rules for connectives, the proof and refutation judgments are not merely rearrangements with one formula standing out, but rather this one formula, marked by separating it from the rest of the context with a pipe symbol (|), is treated specially. Essentially, the proof is split up into different phases, and the rules are formulated in such a way that a proof with some distinguished formula requires proofs where the distinguished formula is a subformula of the original one. This approach, called *focussing*, was originally devised by Andreoli (2001), who intended it as a way to cut down non-essential nondeterminism in proof search; we come back to focussing and a different way of using it to develop a term assignment in section 3.4.

For now we concentrate on the term assignment Downen gives for the rules he obtains from the CUT and AXIOM rules via splitting the judgment. These rules are shown in Figure 3.3. Downen refers to the proof-related judgment $\Gamma \vdash v : A \mid \Delta$ and the refutation-related judgment $\Gamma \mid e : A \vdash \Delta$ as an *active sequent on the right/left*, respectively, and the assigned syntactic expressions v and e as *term* (or *producer*) and *co-term* (or *consumer*), respectively. The simple judgment without a distinguished formula he refers to as a *passive sequent* and the syntactic expression assigned to it as *command*. The CUT rule now has as premises one active sequent on the left and another active sequent on the right, and results in a passive sequent. Syntactically, the command assigned to the result of this rule is thus a pair of a term v and a co-term e written as $\langle v \| e \rangle$. The AXIOM rule is split into two with one concerned with

[2]Such a description of cut elimination does not apply to Gentzen's LK (Gentzen, 1935), but only to the calculi with term assignment that we now look at. Other than in those systems, in LK a cut does not necessarily obtain from a meeting of a left and right rule for the same connective. Therefore reduction of cuts in the sense described here is only sufficient for some cases of cuts in LK; in other cases, cut elimination does not remove the cut in a single step, but rather first makes the relevant subproofs smaller (the cut can then be removed iteratively by further shrinking the subproofs).

active sequents on the left and the other with active sequents on the right; in both cases, the (co-)term assigned to the distinguished formula is simply the variable for that formula from the context. For going from a passive sequent to an active one by picking a distinguished formula there are two rules that we name ACTIVATE, one for the left, one for the right, which are not present in plain sequent calculus (naturally, since the need for them only arises when splitting judgments in this way).

It is the term assignment for these last two rules that arguably requires the greatest amount of creativity. The author thinks that these are best understood by considering the computational, i.e. reduction, perspective. We said that we started out with the CUT module of sequent calculus because a local version of cut elimination relates to a reduction step. But we now saw that a cut constitutes a meeting of a proof and a refutation (and active sequent on the right and one on the left), so we can easily characterize computation as being triggered by this meeting of opposing forces. Let us sketch what we expect of the reduction rule for commands, which are the terms assigned for uses of CUT, where we use $\rightsquigarrow$ for the reduction relation:

$$\langle v \| e \rangle \rightsquigarrow v \text{ and } e \text{ merge somehow}$$

Assuming the command is really just pairing the two forces and leaves it up to them how to carry out this merging, and further assuming that both forces are created equal, then both v and e, when they are not a variable, should "know" how to react to the respective other one in order to merge with it. This respective other entity is what they bind with the μ or $\tilde{\mu}$ binder. It suffices to consider μ for the rest of this argument, the other side is readily dualized. Since we said that we μ-bind the opposing force and then need to know how to merge with it, we can express the body of the μ-abstraction with a unary function M which we apply to the entity bound with μ. We refine our reduction rule according to the state of our discussion:

$$\langle \mu\alpha.M(\alpha) \| e \rangle \rightsquigarrow M(e)$$

But we know that the result of the reduction should again be a meeting of opposing forces, and hence a command. For the binding structure we use the one we already have, for the variables in the contexts; this brings the variable into context and makes the command contingent on it, just as intended. And thus we obtain the term assignment for the ACTIVATE rules, as well as the reduction rules for the μ- and $\tilde{\mu}$-abstractions, where we write substitution of a variable using a $\mapsto$:

$$\langle \mu\alpha.c \| e \rangle \rightsquigarrow c[\alpha \mapsto e] \qquad\qquad \langle v \| \tilde{\mu}x.c \rangle \rightsquigarrow c[x \mapsto v]$$

To sum up, starting from the perspective on the sequent calculus as enabling a proof- and a refutation-reading equally well, with some careful analysis one can relatively automatically arrive at a system with a term assignment and reduction rules. The definitions of the typing and reduction rules look easy enough, and it is also rather straightforward to enhance the system to actually cover the left and right rules for connectives so as not to operate on empty air anymore. However, the setup of the core μ and $\tilde{\mu}$ is already problematic if we want to employ this system as a (basis for a) practical programming language, for reasons that we consider in the next subsections.

3.2.4 Caveats of sequent calculus programs

To understand why programming in the classical sequent calculus, at least using the system Downen arrived at by a rather straightforward development, can be problematic, we first discuss the fundamental issue of $\mu/\tilde{\mu}$ compared to constructive systems by considering a well-known non-constructive proof. In the next and final subsection we will then concretely explore reduction in the non-constructive system built on these constructs and how it requires non-local reasoning. Specifically, we

will see that it can be regarded as a low-level programming language with *jumps*. It is important to note, though, that an important application for Downen's system *is* in this low-level area, specifically, as a compiler intermediate language (Downen et al., 2016; Ariola and Downen, 2020). As such, the following discussion is not intended as a general criticism of this work, but rather merely serves to make clear one of the reasons for turning to a different system on which to build the foundational system $\mathcal{PF}$ presented in the next chapter, which does not focus on efficient compilation.

More precisely, proof arguments such as the one we will now consider exhibit non-local control flow with no warning sign of it on the type level. That is, $\mu/\tilde{\mu}$ is *very* powerful, and one needs to be careful when using it. It will be our goal to arrive at a system where some of this carefulness is moved from the programmer to the system itself, especially regarding making control flow more explicit on the type level.

Our example makes use of two logical rules of Downen's system for connectives, specifically, for disjunction and negation. Particularly negation, with the rule that carries formulas over the turnstile, will be instrumental in demonstrating how we can obtain a *non-constructive* proof and what this means for the understandability of the program.

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1(v) : A \vee B \mid \Delta} \vee R_1 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_2(v) : A \vee B \mid \Delta} \vee R_2$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A \mid \Delta} \neg R$$

We now want to obtain a proof term for $X \vee \neg X$, where X is a logical atom. We have not considered logical atoms yet, but it suffices to say that one can think of it like a type variable; Downen's system has rules for universal quantification, and thus we can conceive of our example formula as a completely generic statement that holds for all X. In other words, what we want to generically prove here is nothing else but the *tertium non datur*, which famous rejection as non-constructive by Brouwer helped trigger the foundational crisis of mathematical logic at the turn of the 20th century.

If we attempt to prove $X \vee \neg X$ in constructive logic, we fail. From a programming perspective, in any "standard" programming language based on constructive logic we can also not create a term corresponding to this proof. For instance, in a language with first-class functions we would model negation of X as the function type $X \to \bot$. In order to prove $X \vee (X \to \bot)$ we then would have to either generically provide an X or provide a function that takes an X and cannot produce anything, which both is not possible. However, in a language that supports non-local control flow, as is the case for Downen's classical sequent calculus language, there is the possibility to first decide to enter one branch and later backtrack, which ultimately enables to prove the *tertium non datur*, among other things, but requires non-local reasoning to understand what is happening.

As a first attempt at finding the sequent calculus term for $X \vee \neg X$, we could try to use one of the rules for disjunction, followed by the rule for negation. But this sounds just like the constructive logic attempt that we know fails, and indeed it cannot work since the premise of the right negation rule is an active sequent on the left, which we do not obtain from the right disjunction rules.

Since it appears that there is no way to prove what we want by starting with an injection ι_i, the only other option is to activate a passive sequent using a μ abstraction. This means that we think of the proof of $X \vee \neg X$ as something that is being

requested. Remember that a command $\langle v \| e \rangle$ pairs a proof and a refutation of the same formula, and hence is evidence for a contradiction. Now, a μ abstraction requires a reaction to an arbitrary refutation α, that we know nothing about but what formula it refutes, hence it being a variable. This reaction is to provide evidence for a contradiction given α, and hence (assuming the system is sound, which it is in our case) we have shown that there could not have been any refutation that instantiates the α. Thus (since our system is complete) the formula under consideration is proven.

This ability to prove a formula by demonstrating that it cannot be refuted is the core aspect that makes proofs in the system indirect and hence non-constructive. In our example, we will make essential use of it by reacting to the refutation variable α *twice*; it is easy to show that a singular use of α does not provide additional provability strength over direct proofs, i.e. proofs only using the rules used for forming canonical values. We can sum up the proof of the *tertium non datur* as follows:

1. In a μ abstraction, react to α by a left injection ι_1, now requiring to prove X.
2. The argument to the injection is itself a μ abstraction with variable α_1 for a refutation of X.
3. In this abstraction, react to α again (and not yet α_1!) by a right injection.
4. The argument of this injection uses the proof term for negation not with α_1.

Intuitively, we first try to prove X in order to counter α. And this we do by countering α_1 but this time we do not form a contradiction by finding a proof for X as a direct counterpart to α_1. Rather, we provide different evidence for a contradiction by pairing α with a right injection, and for this injection we need to prove $\neg X$, so we use α_1 and embed this refutation into a proof using negation. That is, we contradict a refutation of $X \vee \neg X$ by a proof of $\neg X$ obtained from the refutation of X, which we had assumed in order to show that no such refutation exists. With the final remark that inserting a variable that we have in context somewhere amounts to using AXIOM, we are ready to translate our idea of the proof into a formal derivation.

$$
\dfrac{
  \dfrac{
    \dfrac{}{\textsc{Ax}\ \alpha}
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{}{\textsc{Ax}\ \alpha}
          \qquad
          \dfrac{
            \dfrac{\mid \alpha_1 : X \vdash \alpha_1 : X}{\vdash \mathsf{not}(\alpha_1) : \neg X \mid \alpha_1 : X}\,\neg R
          }{\vdash \iota_2(\mathsf{not}(\alpha_1)) : \mathrm{TND} \mid \alpha_1 : X}\,\vee R_2
        }{\langle \iota_2(\mathsf{not}(\alpha_1)) \| \alpha \rangle : (\vdash \alpha : \mathrm{TND}, \alpha_1 : X)}\,\textsc{Cut}
      }{\vdash \mu\alpha_1.\langle \iota_2(\mathsf{not}(\alpha_1)) \| \alpha \rangle : X \mid \alpha : \mathrm{TND}}\,\textsc{Activate R}
    }{\vdash \iota_1(\mu\alpha_1.\langle \iota_2(\mathsf{not}(\alpha_1)) \| \alpha \rangle) : X \vee \neg X \mid \alpha : \mathrm{TND}}\,\vee R_1
  }{\langle \iota_1(\mu\alpha_1.\langle \iota_2(\mathsf{not}(\alpha_1)) \| \alpha \rangle) \| \alpha \rangle : (\vdash \alpha : \mathrm{TND})}\,\textsc{Cut}
}{\vdash \mu\alpha.\langle \iota_1(\mu\alpha_1.\langle \iota_2(\mathsf{not}(\alpha_1)) \| \alpha \rangle) \| \alpha \rangle : \mathrm{TND} \mid}\,\textsc{Activate R}
$$

Here we abbreviate $X \vee \neg X$ as TND and a use of AXIOM L for some variable α as AX α. Note that in the first use of CUT (counting bottom-up), we implicitly duplicate the α variable as needed.[3]

We can also describe what we did as trying to prove the disjunction by going into the left branch (i.e., pairing with α), but then "semi-backtracking" into the right branch (i.e., again pairing with α) after learning that to show the left branch we need to react to a refutation of X. Thus we actually are in both branches when finishing the proof. To put it like an electrical engineer, the output of the term, that is captured by α, is wired to two points, where for one we went to the left branch and for the other to the right branch. One can also conceive of this as the "events" in the two

[3] This is ultimately just contraction which we announced in chapter 2 to be admissible due to us using *sets* of formulas, which we also do for Downen's system to simplify things.

$$\frac{\Gamma \mid e : A \vdash \Delta \qquad \Gamma \mid e' : B \vdash \Delta}{\Gamma \mid [e, e'] : A \vee B \vdash \Delta} \vee L \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid \mathsf{not}[v] : \neg A \vdash \Delta} \neg L$$

FIGURE 3.4: Left rules for disjunction and negation with their term assignments (Downen, 2017).

branches not being independent of each other, but being able to *interact* via the α. Let us now consider this in terms of the reduction steps that happen when we pair the proof

$$\mu\alpha.\langle\iota_1(\mu\alpha_1.\langle\iota_2(\mathsf{not}(\alpha_1))\|\alpha\rangle)\|\alpha\rangle$$

with some refutation, and how these are essentially different from reductions in constructive systems.

3.2.5 Reduction requiring non-local reasoning

Triggering a reduction always requires a proof and a refutation to meet. There is computational content inside the μ abstractions, but without a refutation to trigger it the computation does not get started. Of course, it is not possible to refute the *tertium non datur*, thus we will make use of a trick: We specify the refutation as far as possible and when we arrive at a point where we would need to specify a command but cannot do so, we simply "give up" and act as if we somehow did have the necessary command. We will use $\mathfrak{D}$ to refer to such so-called daimonic commands.

Side remark on daimons. This idea goes back to Girard (2001) who likened it to some supernatural force and thus referred to such evidence for a contradiction as *daimonic*. The underlying philosophical idea is that one might want to engage with some proof by challenging it with some refutation in order to find out more about the proof. And even if no such refutation would be logically possible, the ability to challenge the proof for such a purpose is still desirable, hence the introduction of the daimon. From such a wider perspective, daimonic commands form part of a larger logical framework that also allows "good mistakes", as Girard (2001) puts it. We will make further use of daimons later on, but for now we just use them to challenge the *tertium non datur* proof.

Using daimonic commands, we construct the refutation e for the *tertium non datur* shown below; it is a case analysis, written using square braces ([...,...]), that is the term assignment for the left rule for disjunction shown in Fig. 3.4, i.e. with case analysis one can refute a disjunction by refuting both its alternatives.

$$e := [\tilde{\mu}x.\mathfrak{D}_1(x), e_2], \quad e_2 := \tilde{\mu}x.\langle x\|\mathsf{not}[\mu\alpha.\mathfrak{D}_2(\alpha)]\rangle$$

Command-pairing e with the proof developed in the previous subsection leads to the following (possible, see the remark on ambiguity below) reduction sequence:

$$\begin{aligned}
&\langle\mu\alpha.\langle\iota_1(\mu\alpha_1.\langle\iota_2(\mathsf{not}(\alpha_1))\|\alpha\rangle)\|\alpha\rangle \parallel e\rangle\\
\rightsquigarrow\ &\langle\iota_1(\mu\alpha_1.\langle\iota_2(\mathsf{not}(\alpha_1))\|e\rangle) \parallel [\tilde{\mu}x.\mathfrak{D}_1(x), e_2]\rangle\\
\rightsquigarrow\ &\langle\mu\alpha_1.\langle\iota_2(\mathsf{not}(\alpha_1))\|e\rangle \parallel \tilde{\mu}x.\mathfrak{D}_1(x)\rangle\\
\rightsquigarrow\ &\langle\iota_2(\mathsf{not}(\tilde{\mu}x.\mathfrak{D}_1(x))) \parallel [\tilde{\mu}x.\mathfrak{D}_1(x), e_2]\rangle\\
\rightsquigarrow\ &\langle\mathsf{not}(\tilde{\mu}x.\mathfrak{D}_1(x)) \parallel \tilde{\mu}x.\langle x\|\mathsf{not}[\mu\alpha.\mathfrak{D}_2(\alpha)]\rangle\rangle\\
\rightsquigarrow\ &\langle\mathsf{not}(\tilde{\mu}x.\mathfrak{D}_1(x)) \parallel \mathsf{not}[\mu\alpha.\mathfrak{D}_2(\alpha)]\rangle\\
\rightsquigarrow\ &\langle\mu\alpha.\mathfrak{D}_2(\alpha) \parallel \tilde{\mu}x.\mathfrak{D}_1(x)\rangle\\
\rightsquigarrow\ &\mathfrak{D}_2(\tilde{\mu}x.\mathfrak{D}_1(x))
\end{aligned}$$

Reduction of μ- and $\tilde{\mu}$-abstractions paired with some term happens by instantiation of the respective variable that was abstracted over with the term, as outlined in section 3.2.3. To resolve ambiguity we always have μ take precedence over $\tilde{\mu}$.[4] Eventually, we arrive at the meeting of a not proof term and a not refutation term; the not refutation is like the not proof explained above, just the other way around: it embeds a proof into a refutation. It thus is the term assignment for the left negation rule also shown in Fig. 3.4. Computationally, the two not terms cancel out in a reduction step that leads to the command that pairs the proof embedded in the refutation with the refutation embedded in the proof. After this, there is only one further step possible (that is, without some specification of daimonic reduction), instantiating a refutation variable to produce the daimonic command $\mathfrak{D}_2(\tilde{\mu}x.\mathfrak{D}_1(x))$.

The reduction shows again how the proof involves moving into the left branch and then the right and using the refutation variable α twice. Let us now try to relate this to, and differentiate it from, reductions of the more familiar intuitionistic natural deduction proofs for which there is a direct relation to familiar (functional) programming languages.

In such languages, we have introduction and elimination rules, and reduction amounts to simplifying terms by simplifying meetings of introduction and elimination. For instance, we can have an injection and a case analysis on it, and if it is a left injection this meeting reduces in one step to the left branch of the analysis, and likewise for the right injection and the right branch. This phenomenon, and the analogous one for conjunction, can also be realized in the sequent calculus language we are considering, by pairing injection and case analysis in a command as we saw above.

But this computation, like any proof-reduction computation, must be encapsulated in a μ-abstraction ($\tilde{\mu}$ in the case of refutations). For simple instances of such abstractions, we can draw a direct correspondence to intuitionistic natural deduction by conceiving of the α in $\mu\alpha.c$ as the *return* which signifies the end of the computation. As a simple example of such a correspondence, consider the two terms below, one from intuitionistic natural deduction shown on the left (where tt is the only value of the unit type) and the other from sequent calculus shown on the right.

$\texttt{inl}(\textbf{match}\ (\texttt{inr}(\texttt{tt}))\ \{$ $\quad\texttt{inl}(x) \Rightarrow x;$ $\quad\texttt{inr}(x) \Rightarrow x\})$	$\iota_1(\mu\alpha.\langle\iota_2(()) \,\|[$ $\quad\tilde{\mu}x.\langle x\|\alpha\rangle,$ $\quad\tilde{\mu}x.\langle x\|\alpha\rangle]\rangle)$

The term on the left obviously reduces to $\texttt{inl}(\texttt{tt})$ in one step, by reducing the match. The term on the right is not reducible, but if we trigger computation by pairing it to a command with some refutation, say $[\tilde{\mu}x.\mathfrak{D}(x), ...]$, this command reduces to

$$\langle\iota_2(()) \,\|\, [\tilde{\mu}x.\langle x\|\tilde{\mu}x.\mathfrak{D}(x)\rangle, \tilde{\mu}x.\langle x\|\tilde{\mu}x.\mathfrak{D}(x)\rangle]\rangle$$

in two steps (the first step is just for the case analysis of the ι_1); α is now instantiated with our outer refutation (for the relevant injection ι_1). Now this reduces in one step, by case distinction, to:

$$\langle() \,\|\, \tilde{\mu}x.\langle x\|\tilde{\mu}x.\mathfrak{D}(x)\rangle\rangle$$

And this reduces to $\mathfrak{D}(())$ in two steps, so the result of the reduction passes the unit value to the daimon, and the unit value was also the result of the intuitionistic natural deduction example reduction. Especially, the positions behind the $\Rightarrow$ in the two cases of this example are the return positions of the entire match and hence

[4] This ambiguity is called the *fundamental dilemma of computation* and there exist well-known fixes for it. In the system we eventually arrive at, this dilemma is circumvented, hence we will not further consider it for now.

$$\begin{array}{l}\textbf{function}\ \texttt{tnd}\langle T\rangle() : T + \texttt{JumpPoint}(T)\ := \\ \quad \textbf{mark}_{\alpha}\ \{\texttt{inl}(\texttt{tnd_aux}())\} \\ \\ \textbf{function}\ \texttt{tnd_aux}\langle T\rangle() : T\ := \\ \quad \textbf{mark}_{\alpha_1}\ \{\textbf{jump}_{\alpha}\ \texttt{inr}(\alpha_1)\}\end{array}$$

FIGURE 3.5: *Tertium non datur* proof as a program.

determine what the argument value to `inl` is, and this return directly corresponds to the use of α in the $\tilde{\mu}$, as promised.

More generally, however, such an α need not be used only for such simple returns. In the *tertium non datur* proof we saw α being used twice, and this not simply in two separate cases of a case analysis. We can conceive of such a more liberal use of α as allowing us to jump around in the program, giving us a powerful way to alter the control flow. Figure 3.5 shows a translation of this proof into some hypothetical programming language with *jumps*.[5] Here we only use (parameterless) functions to give names to subterms and easily show their types. Function `tnd` is for the overall term, it is for a disjunction, which we translated to a sum type, with generic T bound by a type parameter using angle brackets ($\langle T\rangle$). Since `tnd` is at the position of the overall result that is to be paired with the refutation α, we mark it as a jump target α with $\textbf{mark}_{\alpha}$. Its body wraps a left injection around a use of `tnd_aux`, which similarly is a jump target marked with α_1. The most important feature of our hypothetical programming language is its treatment of jump markers as *first-class* entities, which we type as `JumpPoint`(T), corresponding to the negation of the type at the position of the marker. We use this in `tnd_aux` to provide something that typechecks at position (marker) α, by jumping to α with a right injection of the first-class marker α_1. This jump to α directly corresponds to the sequent calculus term's second use of α, and the way we can pass around the first-class marker α_1 corresponds to wrapping the refutation variable α_1 with neg to obtain a proof from a refutation.

Overall, we can say that there is a way to think of the sequent calculus system as a programming language, but programs in it in general are allowed to have arbitrary jumps, meaning that there is no guarantee of the ability to reason in a locally restricted way. Thus this language does not look much like the high-level declarative functional programming languages, but more like a low-level language with finer control over the control flow.

In the following section, we turn to a different system inspired by the sequent calculus but not directly following its structure, the Calculus of Unity (**CU**) (Zeilberger, 2008b); **CU** is likewise entirely symmetric in the negation duality sense. Crucially, **CU** does not have the μ and $\tilde{\mu}$ operators; it is possible to recreate proofs involving such operators, but this requires to use the constructs of **CU** in such a way that the control flow becomes more explicitly reflected in the types. **CU** is also more generally presented in a different way, build up from judgments and fragments, that in particular allows for easily combining it with the program transposition of the previous chapter, to arrive at the foundational system $\mathcal{PF}$ presented in the next chapter.

[5] The relation between classical proofs and typed programs with control operators allowing for jumps was first described by Griffin (1989) for a typed variant of Scheme containing an operator similar to call/cc.

The two fragments that **CU** is built from represent the *positive* and *negative polarity*, respectively. The next chapter discusses polarity in so far as it is relevant for the present work; a key aspect for the development of $\mathcal{PF}$ is the relation between polarity and the data/codata dichotomy.[6] As we will see, the different judgments also delineate precisely the boundaries of intuitionistic and classical logic within the system. Together with the typing precision, this will allow to easily recover existing languages in $\mathcal{PF}$, as we will see in a later chapter.

3.3 Dialogical Polarity and Symmetry

When we considered sequent calculus and ways to program "in" the sequent calculus, what we have neglected thus far is how we made the decision which rules we pick for a certain connective. In the sequent calculus, to fully specify some connective one has to provide left rules and right rules for it, which are explained in Section 2.1.3. This means that there are actually two ways to specify what is semantically (think of truth tables) the *same* connective. In this section, we consider how this aspect eventually leads us to an alternative way to combine programming and classical logic, one in which both the extensibility and the negation duality appear quite natural. The key to this is to embrace the idea that the two different ways to specify the connective actually indicate that these are two different connectives, of opposite *polarity*, i.e. one has *positive* polarity and the other has *negative* polarity. This, as Zeilberger (2008b) demonstrated, can be given a philosophical justification via two dual meaning theories of logic. His considerations lead him to design a fully symmetric system, the Calculus of Unity (**CU**), which integrates two structurally identical fragments, one for each polarity, via so-called *shifts*. This system **CU** will be the basis for the foundational, duality-consolidating system $\mathcal{PF}$ presented in the next chapter, which in particular eliminates the structural difference between data and codata by conceptually equating them with positive and negative polarity connectives. The seed of making use of the polarized system **CU** like this is found in Binder et al. (2019) (in a paragraph written by the author, as mentioned in chapter 2).

An interesting connection between (co)data and polarity is also pointed out by Downen et al. (2019): When viewing computation as a dialogue between two actors where polarity defines who gets to move first, data types are related to the proponent side (seeking to prove a statement) and hence to positive polarity, while codata types are related to the opponent side (seeking to refute) and hence to negative polarity. Both data and codata types can be seen as specifying the valid moves the respective side starts with (which the other side then has to react to). This dialogical perspective is an interesting starting point for a better understanding of data and codata in the Curry-Howard sense. However, it is only rather superficially represented by data and codata languages (including $\mathcal{PF}$); the outlook in Chapter 6 contains some thoughts on developing a session type system which subsumes $\mathcal{PF}$ and which more fully embodies the dialogical view. While this work is not intended to travel the full road from De Morgan duality to dialogical systems, the author still thinks that having this mental picture relating the two can be instructive when considering data

[6] Downen (2017) also enhances his system with polarity and a central aspect of his work is also about this relation, plus the connection to evaluation strategies (see also the paper specifically about this topic that Downen coauthored (Downen and Ariola, 2014)). We turn to the system of Zeilberger (2008b) for the reasons given in this paragraph; it should also be noted that Zeilberger's system is *inherently* about polarity rather than adding polarity in a later step.

and codata within the greater picture of the extended Curry-Howard program, in particular when thinking about polarity.

Let us now climb into the central ideas leading to **CU**, beginning with the proof-theoretical background of polarity.

3.3.1 Polarity

Consider the following left and right rules for conjunction (left rules are from Gentzen's LK, right rule is modified from the one given in LK by disallowing merging of different premise contexts).

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge L_1 \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge R$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge L_2$$

An important property of the right rule that the left rules lack is *invertibility*. That is to say, whenever a sequent of the form given in its conclusion is derivable, the respective sequents obtained by "inverting" the rule and applying it to that sequent are also derivable using the rules of the system. The same does not hold for any of the two left rules, and this would be semantically invalid anyway (or one could also say the connective specified would not be what we semantically perceive of as conjunction), since when a conjunction $A \wedge B$ implies some Δ that does not mean A or B alone implies Δ. The alternative way to specify conjunction by left and right rules, shown below, is to use the same right rule as before but a different left rule.

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge L \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge R$$

This left rule *is* invertible (which semantically is obvious considering the conjunctive interpretation of the left-hand side of the sequent). We will now consider the two set of rules to specify *distinct* connectives, despite their semantic, in the sense of truth tables, equivalence. The former connective we will write $\wedge^-$ and say that it has *negative polarity* due to its left rule being non-invertible, and the latter connective we will write $\wedge^+$ and say that it has *positive polarity* due to its left rule being invertible. The same approach applies to other connectives as well, as we will see shortly.

What is the benefit of having the full diversity of rules in the system that gives rise to two "artificial" connectives per "semantic" connective? First of all, whether a rule is invertible or not has profound practical implications in proof search. When one wants to prove some sequent that is an instance of the conclusion of some invertible rule, one can just go "backwards" over the rule and prove the respective instances of the premises. This is guaranteed to eventually lead to a proof tree if one exists, since the set of premises is logically equivalent to the conclusion. Non-invertible rules, on the other hand, potentially require backtracking and choosing a different rule to go "backwards" over. Combined with the idea of focussing on a formula, and that of shifts which allow to change the focus and embed one polarity into the other, *polarizing* a formula by picking the polarity of connectives that appear in it and inserting shifts appropriately can help to control the search space. A discussion of this is beyond the scope of this work, but we will soon again consider focussing and shifts in so far as these are relevant for the analysis of programming languages.

It is also worth mentioning that the concept of polarity originally came from the study of linear logic, in which, due to the lack of certain structural rules, a connective

and its counterpart of opposite polarity are indeed *not* equivalent logically. This is in itself an interesting area of study, involving the resource and dialogical interpretations of linear logic, and we will briefly discuss its relation to the present work in the outlook in Chapter 6.

However, while all that may be indicating that polarity is potentially relevant, there does not seem to be fundamental, perhaps philosophical, explanation of its place in logic outside of linear logic, does there? Actually, there is such an explanation that will even help us resolve the tension between intuitionistic natural deduction and classical sequent calculus hinted at earlier and pave the road for the fundamental system $\mathcal{PF}$. To get there, we now turn back to natural deduction in the style without sequents, i.e. hypotheses are explicitly positioned in the proof trees, and consider what the notion of polarity means in this context and what this has to do with the *justification of logical rules* by an analysis of their *meaning*.

3.3.2 Meaning theories

First, remember that in natural deduction there are only right rules, but other than in sequent calculus these may not only be introduction rules, but also elimination rules. For example, conjunction introduction (I) and elimination (E) in natural deduction look like this (presented without sequents):

$$\frac{A \wedge B}{A}\wedge E_1 \qquad\qquad \frac{A \wedge B}{B}\wedge E_2 \qquad\qquad \frac{A \qquad B}{A \wedge B}\wedge I$$

What we now want to consider is how to *justify* that these rules make sense, and more specifically how to achieve that within proof theory, without appealing to some external semantic argument. The following is a summary of Dummett's account of *verificationist* and *pragmatist meaning theories* (Dummett, 1991) following Zeilberger (2008b) who aptly worked out its essential points.

In the verificationist meaning theory, a proposition is considered to be given a meaning by its *canonical proofs*, which Dummett (1991) defines to be proofs that *end in a sequence of introduction rules*. We can then justify some arbitrary logical rule, including elimination rules, by showing that, for any proposition that has a proof ending in a use of this rule where the premises have canonical proofs, one can find a canonical proof for that proposition. For example, to justify the first conjunction elimination rule shown above we first consider some proof ending in a use of this rule applied to an arbitrary canonical proof $\mathcal{D}$ of the premiss $A \wedge B$.

$$\frac{\begin{matrix}\mathcal{D}\\ A \wedge B\end{matrix}}{A}\wedge E_1$$

By definition, we know that the canonical proof $\mathcal{D}$ must end in a sequence of introduction rules, the final one of which must be the conjunction introduction rule.

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_1\\ A\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ B\end{matrix}}{A \wedge B}\wedge I}{A}\wedge E_1$$

We also know that the proofs $\mathcal{D}_1$ and $\mathcal{D}_2$ to which the introduction rule is applied must themselves be canonical. Further, $\mathcal{D}_1$ must be a proof for A, and so we have found a canonical proof for the proposition A proven in the original proof:

$$\begin{array}{c}\mathcal{D}_1\\ A\end{array}$$

Thus, according to the verificationist meaning theory, we have justified the first conjunction elimination rule; the second rule is similarly justified.

The reader has hopefully acquired some familiarity with looking for dualities by now. If so, it will not be too surprising that the pragmatist meaning theory, which is intended to be dual to the verificationist one, is concerned with canonical uses of elimination rules. More precisely, Dummett (1991) defines a *canonically-obtained consequence* as being a sequence of elimination rules. The idea, dually to the verificationist side, is then that justifying some rule requires to transform any proof that begins with that rule applied to some propositions, followed by a canonically-obtained consequence, into a proof with one of the same initial propositions to which that consequence is immediately applied. For example, to justify the conjunction introduction rule, we consider its possible canonically-obtained consequences. There are two possibilities: The first elimination rule used could be $\wedge E_1$, or it could be $\wedge E_2$. After that, some other elimination rules follow (the corresponding trees here are named $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively).

$$\dfrac{\dfrac{A \quad B}{A \wedge B}\wedge I}{\begin{array}{c}A\\ \mathcal{D}_1\end{array}}\wedge E_1 \quad \rightsquigarrow \quad \begin{array}{c}A\\ \mathcal{D}_1\end{array} \qquad\qquad \dfrac{\dfrac{A \quad B}{A \wedge B}\wedge I}{\begin{array}{c}B\\ \mathcal{D}_2\end{array}}\wedge E_2 \quad \rightsquigarrow \quad \begin{array}{c}B\\ \mathcal{D}_2\end{array}$$

The proof tree on the left can obviously be transformed into one that starts with the initial proposition A and ends with the canonically-obtained consequence $\mathcal{D}_1$ (just take its tail); the same goes for the proof tree on the right with the proposition B and consequence $\mathcal{D}_2$.

In summary, both theories are about reducing proofs, which in the examples considered amounted to simply cancelling out introduction and elimination, to a form in which only a certain kind of rules is used, i.e. a *canonical form*. What is crucial, according to Dummett (1991), is that both the verificationist approach, in which canonicity comes from building a proof in the most elementary way, and the pragmatist approach, in which canonicity comes from using a given proposition in the most elementary way, are *equally convincing*. Zeilberger (2008b) goes one step further and argues that these two ways of assigning meaning to a connective actually give rise to *different* connectives. He states that from the point of view of the meaning theories, these are really semantically distinguishable and do not require to be "harmonized" as Dummett (1991) demanded.

And with this we have actually found a more fundamental way to justify (the relevance of) allowing positive polarity and negative polarity connectives to exist side-by-side. Conceiving of positive polarity connectives to be those which are justified verificationally, and of negative polarity connectives to be those which are justified pragmatically, we can keep them apart even without appealing to linearity restrictions. The characterization of the positive connective in terms of the ability to turn any proof into a canonically-obtained proof, which consist only of introduction rules, in sequent calculus corresponds to the specification of the connective with an invertible left rule, and a similar correspondence can be established for negative connectives, canonically-obtained consequences, and invertible right rules. The idea of the existence of proof (consequence) trees in canonical form from natural deduction in sequent calculus translates into the logical equivalence between a left (right) rule's conclusion sequent and the set of sequents in its premises.

Observe how building a proof for some proposition in a canonical way and using some proposition for a proof in a likewise canonical way are reminiscent of constructing a value using the constructors of some data type, and observing a value of some codata type by means of its destructors, respectively. This was the reason for why Binder et al. (2019) were interested in the work of Zeilberger (2008b) for improving the symmetry of their (co)data language and obtaining what is perhaps an ideal system for the extensibility duality. Zeilberger builds on Dummett's ideas, developing two *isomorphic* system fragments, one with canonical proofs, representing the verificationist side, and the other with canonical refutations, representing the pragmatist side; for the latter, the basic idea relating this to the pragmatic meaning theory could be summarized by refutation being the most elementary way of using a proposition. The system formed from these fragments, the Calculus of Unity (**CU**), in turn forms the basis for the foundational system $\mathcal{PF}$ presented in the next chapter. In $\mathcal{PF}$, the data and codata types of Binder et al. (2019) are replaced by more symmetric counterparts, called positive and negative data types, which are structurally identical. These are variations of the types from the two isomorphic fragments of Zeilberger (2008b) just mentioned. The data and codata languages presented in the previous chapter can also be recovered as surface languages as presented in chapter 5, demonstrating that one can view $\mathcal{PF}$ as a proper generalization of these.

Overall, the negation duality studied in this chapter and the extensibility duality studied in the previous one both quite naturally arise from **CU** (and, by extension, $\mathcal{PF}$). The end of the present chapter will also sketch the basic idea behind employing shifts that go between the positive and negative fragments, for recovering the call-by-value and call-by-name evaluation strategies; the details of achieving that in $\mathcal{PF}$ are found in chapter 5. The essence of this approach, due to Zeilberger (2008b), is related to how the negation duality underlies the pair call-by-value and call-by-name. Due to the polarized setting, where a type is either positive or negative, each type inherently lends itself to be used to model either call-by-value or call-by-name, rather than being agnostic in this regard. That is, in the non-polarized setting, the evaluation order is only controlled by constructs like μ / $\tilde{\mu}$, as mentioned in Section 3.2.3. This allows to make use of some type for either call-by-value or call-by-name, and, as Wadler (2003) showed explicitly, a term reduces under call-by-value to some other term when there is a call-by-name reduction between the respective dual terms, where these dual terms have types which are De Morgan dual to the original terms, i.e. product (conjunction) is dual to sum (disjunction). In the polarized setting, polarity records the evaluation strategy that befits the type, e.g. a negative product delays evaluation until a projection is chosen (the evaluation behavior known from lazy records). Because of this association between types and evaluation, the shifts are needed if one wants to, e.g., delay evaluation of a positive term, in which case the respective positive type needs to be explicitly wrapped in a shift to produce a negative type, making the evaluation strategy explicit at the type level; more on this later.

3.3.3 The Calculus of Unity

This section and the following summarize the key aspects of **CU** in so far as they are relevant for the following chapter. In the system one can prove all valid propositions of classical logic and refute all of its invalid propositions, provided that one chooses the appropriate connectives of the polarity related to proving or refuting, respectively. More specifically, one can understand the system as a kind of a refinement of classical sequent calculus with polarization and focussing, and with canonical

$$\dfrac{\begin{matrix}[A] \\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C\end{matrix} \quad A \vee B}{C} \vee E$$

FIGURE 3.6: Disjunction elimination rule in natural deduction.

proof and refutation judgments for intuitionistically valid propositions and dual-intuitionistically refutable propositions, respectively, and judgments for refuting the possibility of a canonical proof and for refuting the possibility of a canonical refutation; the latter two judgments are exactly classical proof and refutation when erasing the polarity of the connectives.

It should be noted that using the sequent calculus formalism, with its symmetries discussed previously, as a starting point for the refinement is important, even though the inspiration for the meaning theories came from natural deduction. There are connectives not considered in the examples above, like disjunction in the case of the pragmatist meaning theory, which, in the natural deduction setting, require considerable bureaucracy that complicates the definition of canonical forms and hinders symmetric treatment of positive and negative polarity.

The natural deduction disjunction elimination rule is shown in Fig. 3.6. It has as a premiss the proposition with the disjunction to be eliminated ($A \vee B$), plus two premises with the proposition C in its conclusion, which both depend on some hypothesis which is the left (A) or the right (B) subformula of the disjunction, respectively. As explained in chapter 2, in general, rules in natural deduction require a non-local manipulation of the proof tree, and this is the case for disjunction elimination: In order to conclude C from $A \vee B$, C must depend on both a hypothesis A and a hypothesis B somewhere up in the proof tree, and these hypotheses are then closed by the application of the rule. The sequent-style presentation, while semantically equivalent, more clearly shows the symmetry to conjunction rules: To conclude $\Gamma, A \vee B \vdash C$ one needs $\Gamma, A \vdash C$ and $\Gamma, B \vdash C$; when generalizing the right-hand side to multiple propositions as in sequent calculus, this is just like the right rule for conjunction shown above, but with the left-hand and right-hand sides of each sequent flipped.

As we will see, in the canonical forms of proofs and refutations in **CU** based on the sequent calculus right and left rules, the duality of conjunction and disjunction is even more readily apparent: Both the evidence for refuting a negative disjunction and that for proving a positive conjunction are structurally pairs (of proofs and refutations, respectively), just as the evidence for proving a positive disjunction and that for refuting a negative conjunction are structurally injections into a disjoint union (cf. the structural similarity of injections for sum types and projections for product types); more details on this follow below.

However, before we delve into the details of **CU**, note that though it is similar to the sequent calculus system of Downen (2017), the non-polarized variant of which we discussed above in this chapter, the design of **CU** is *inherently* about polarity, due to it conceptually building on the meaning theories of Dummett (1991), whereas Downen (2017) starts from a core system to which polarity is later added. Another key difference between **CU** and the system of Downen (2017) is that the former lacks anything that corresponds to the latter's μ and $\tilde{\mu}$ constructs, which however form the core of that sequent calculus language. It is possible to add these to **CU**, however, as outlined above, there are reasons, like the lack of typing precision, to decide against

that. As announced, the central goal is to recover a diverse variety of surface languages by macro embedding, which is possible in the **CU**-based $\mathcal{PF}$ by utilizing shifts, and does not require anything that corresponds to $\mu/\tilde{\mu}$. A key point will indeed be how the shifts enable the typing precision which $\mu/\tilde{\mu}$ would actually help circumvent. We will thus not consider adding this feature in this work.[7]

The following considers the positive polarity fragment, then briefly the structurally isomorphic negative polarity fragment, and finally the shifts that combine these to form **CU**.

3.3.4 The positive and negative fragments

The positive fragment is made up of a judgment for *direct* proofs, i.e. the canonical proofs which cover intuitionistically true statements, and a judgment for *justified* refutations, which cover classically false statements. When thinking about the system in a more programming language-like way, direct proofs are called *values*, and justified refutations are called *continuations*; the idea is that the latter take the former as inputs, and, using this input value, provide evidence for a contradiction, thereby refuting the proposition that the possible inputs are proofs of. An evidence for a contradiction is formed by combining a value and a continuation (written with a juxtaposition $K\ v$) for the same proposition; this combination is called a *command*[8] (this aspect is very similar to the system of Downen (2017)).

$$\frac{\Gamma \vdash K \overset{\text{cnt}}{:} P \qquad \Gamma \vdash v \overset{\text{val}}{:} P}{\Gamma \vdash K\ v \overset{\text{cmd}}{:} \#}$$

The three judgments for values, continuations, and commands, each in some context Γ (see below), are written $\Gamma \vdash v \overset{\text{val}}{:} P$, $\Gamma \vdash K \overset{\text{cnt}}{:} P$, and $\Gamma \vdash C \overset{\text{cmd}}{:} \#$, respectively. The first judges a value v to be a direct proof for proposition P, the second judges a continuation K to be a justified refutation for proposition P, and the third judges a command C to be evidence for a contradiction. Just as with the system of Downen (2017), a command is the syntactic entity that represents computation: a command combining some value and continuation reduces to another command by substituting parts of the value, according to a pattern match, into the body of the continuation; more details on this follow below.

The context Γ is a list of variables together with a proposition, as usual; however, each of the variables stands for *either* a proof *or* a refutation, with the judgment being explicitly annotated, e.g. $x \overset{\text{val}}{:} X$ or $y \overset{\text{cnt}}{:} P$. In **CU**, value variables are restricted to atoms, the reason for which we will consider soon. Rearranging the value variables and the continuation variables to be to the left of the turnstile and to its right, respectively, one obtains the presentation of contextual proofs and refutations used by the sequent calculus. The value judged to be proof for some atom or the continuation judged to be a refutation are then in the place of the proposition that the current *focus* of the proof is on, as in the sequent calculus system of Downen (2017); just as in that system, the command judgment has no such focussed-upon proposition as it is for

[7]The author still thinks that $\mu/\tilde{\mu}$ does have its place, though, whenever programming convenience in the core system itself becomes relevant. Here, a potentially interesting research topic is making $\mu/\tilde{\mu}$ compatible with program transposition.

[8]Zeilberger (2008b) uses the terminology *statement*, but the author decided to use *command* (following, e.g., Downen (2017)) instead (also for $\mathcal{PF}$), due to it sounding slightly less general to his ears.

evidence of a contradiction, which has a special place in sequent calculus (specifically, the empty sequent being a non-contingent contradiction and hence impossible to derive).

The propositional connectives of the positive fragment are conjunction, written $\otimes$, disjunction, written $\oplus$, and negation, written $\overset{v}{\neg}$.[9] The underlying inspiration for **CU** were the meaning theories of Dummett (1991) and their association to polarity, as discussed above. Especially, as summarized above, positive polarity is linked to canonical proofs which Dummett, in natural deduction, defined to be ending in a sequence of introduction rules. Therefore, rules for the value judgment mirror the introduction rules. As a first attempt, one could hence give the following rules for conjunction and disjunction, where P and Q are positive propositions (i.e. propositions formed from $\otimes$, $\oplus$, and $\overset{v}{\neg}$):

$$\frac{\Gamma \vdash v_1 \overset{\text{val}}{:} P \qquad \Gamma \vdash v_2 \overset{\text{val}}{:} Q}{\Gamma \vdash (v_1, v_2) \overset{\text{val}}{:} P \otimes Q} \qquad \frac{\Gamma \vdash v \overset{\text{val}}{:} P}{\Gamma \vdash \mathrm{inl}(v) \overset{\text{val}}{:} P \oplus Q} \qquad \frac{\Gamma \vdash v \overset{\text{val}}{:} Q}{\Gamma \vdash \mathrm{inr}(v) \overset{\text{val}}{:} P \oplus Q}$$

Indeed, these rules are *derivable* in **CU**; however, the actual rules specified for the positive fragment are a bit different. In effect, they are refactored from such rules such that there is only a single rule for the value judgment that makes use of an auxiliary *pattern* judgment $\Delta \Rightarrow p \overset{\text{val}}{:} P$ and a substitution judgment $\Gamma \vdash \sigma : \Delta$.

$$\frac{\Delta \Rightarrow p \overset{\text{val}}{:} P \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash p\sigma \overset{\text{val}}{:} P}$$

A value is thus defined as a pattern p with some substitution σ applied to it. Here *pattern* has a meaning similar to that in functional programming: a tree of constructors, corresponding to introduction rules, with variables at the leaves. These variables may only be *continuation variables* or variables for logical atoms; we will see shortly what the idea behind that is. The pattern judgment says that any substitution σ that fills the variables in p with continuations which are, in order, refutations for the propositions in Δ, results in a value $p\sigma$ which is a proof for P. The rules for the substitution judgment merely check each continuation against its associated proposition. The rules for the pattern judgment omit the context Γ since there are no variables in the patterns themselves. For conjunction and disjunction, the pattern rules look like this:

$$\frac{\Delta_1 \Rightarrow p_1 \overset{\text{val}}{:} P \qquad \Delta_2 \Rightarrow p_2 \overset{\text{val}}{:} Q}{\Delta_1, \Delta_2 \Rightarrow (p_1, p_2) \overset{\text{val}}{:} P \otimes Q} \qquad \frac{\Delta \Rightarrow p \overset{\text{val}}{:} P}{\Delta \Rightarrow \mathrm{inl}(p) \overset{\text{val}}{:} P \oplus Q} \qquad \frac{\Delta \Rightarrow p \overset{\text{val}}{:} Q}{\Delta \Rightarrow \mathrm{inr}(p) \overset{\text{val}}{:} P \oplus Q}$$

[9] In the latter, the v stands for call-by-value. Zeilberger (2008b) calls positive polarity negation *call-by-value negation* due to how one can model call-by-value using it; we return to this in chapter 5.

$$\begin{aligned}&\texttt{neg}(K \overset{\text{cnt}}{:} 1 \oplus 1) \overset{\text{cnt}}{:} 1 \oplus 1 \;:=\; \lambda\{\\ &\quad \mathrm{inl}(()) \;\mapsto\; K\,\mathrm{inr}(())\\ &\quad \mathrm{inr}(()) \;\mapsto\; K\,\mathrm{inl}(())\\ &\}\end{aligned}$$

FIGURE 3.7: Continuation example.

The decomposition into pattern and substitution judgments reduces the bureaucracy when adding new user-defined connectives, for which it suffices to give such pattern rules.[10] For completeness sake, let us also list the rather trivial pattern rules for logical atoms X and for the nullary connective 1 (constant truth), where $\cdot$ is the empty context.

$$\frac{}{x \overset{\text{val}}{:} X \Rightarrow x \overset{\text{val}}{:} X} \qquad\qquad \frac{}{\cdot \Rightarrow () \overset{\text{val}}{:} 1}$$

Unsurprisingly, there is no rule that allows one to obtain a pattern for the constant falsehood (nullary connective 0).

Now, what is still missing among the pattern rules is that for negation, which is shown below.

$$\frac{}{x \overset{\text{cnt}}{:} P \Rightarrow x \overset{\text{val}}{:} \overset{\text{v}}{\neg} P}$$

A pattern for a negation cannot be further discriminated, it is only a placeholder for the continuation to be substituted for it. Thus, obtaining a proof of a negation means embedding a continuation for the negated proposition.

So how do we obtain a continuation? The idea is that a continuation considers all patterns from which a value for the proposition under consideration could be formed, according to the pattern judgment. In constructing the evidence for the contradiction, i.e. a command, in each pattern case, it can make use of the continuation (and atom) variables, and only those, since value variables do not exist. The idea behind the lack of value variables (of non-atomic propositions) is that this aligns with focussing, which also does not allow to liberally switch to deconstructing a different subformula. This property is interesting since it demonstrates a correspondence between focussing and pattern matching, but is not relevant for the foundational system $\mathcal{PF}$ considered in the next chapter. We will therefore lift this restriction in $\mathcal{PF}$ and allow value variables for any proposition/type.

Now, the actual rule for the continuation judgment given in Zeilberger (2008b) is not particularly syntactic:

$$\frac{\forall(\Delta \Rightarrow p \overset{\text{val}}{:} P) : \qquad \Gamma, \Delta \vdash \phi(p) \overset{\text{cmd}}{:} \#}{\Gamma \vdash (\lambda\phi) \overset{\text{cnt}}{:} P}$$

Just as described above, a continuation $\lambda\phi$ consists of a map ϕ from patterns to commands, the details of how to implement it are not specified. Fig. 3.7 shows an example of a continuation called neg, which consumes a $1 \oplus 1$ (which encodes the boolean type) value, matches on it and sends (i.e. forms a command from the value and the

[10]This approach is not directly described in Zeilberger (2008b), but Zeilberger (2008a) employs this approach, albeit with a less symmetric variant of **CU**.

continuation) the opposite injection (as in a boolean negation) to some other continuation K assumed to be available in the context (indicated by writing it in braces behind the name neg). Generally, the rule for the continuation judgment requires, for all possible patterns p for the relevant proposition P (that is to be refuted) and all possible substitutions $\sigma : \Delta$ that fit the respective pattern, that $\phi(p)$ is a valid command according to the command judgment, when extending the context Γ with Δ. How to concretely implement this $\forall$-check is also not specified. In the case of the original **CU** (Zeilberger, 2008b), there are no recursive types, thus there are only finitely many patterns for each proposition/type, which means that the check can of course be implemented. But specifying pattern rules for recursive types is easily possible (see Zeilberger (2008a)), and for these cases we end up with *exotic terms*, like infinite maps from natural number patterns, which are not generally implementable. For Zeilberger (2008a), questions of how to concretely realize his system to make it applicable as a programming language are not his main focus, thus leaving this aspect abstract is arguably comprehensible enough.[11]

As mentioned above, reducing a command amounts to substituting into the body of the continuation according to the pattern match. As an example, using the continuation neg as defined in Fig. 3.7 and assuming some concrete continuation K, we can write the command $\texttt{neg}(K)\ \mathrm{inr}(())$, which reduces, in one step, to $K\ \mathrm{inl}(())$. The next reduction steps depend on K. Why not give a concrete example for K here? We could try to, but this would just lead us to needing another continuation, and so on. As a matter of fact, there is no way to obtain a closed (i.e. without free variables) command, and this is in line with our logical interpretation: a command is evidence for a contradiction, and a closed command would be evidence for a non-contingent contradiction; if such a thing were allowed by the typing rules, the system would be unsound. What we can do is deliberately add special commands that allow us to get computation started. We saw this approach already when "challenging" the proof of the *tertium non datur*; as pointed out, the idea behind this, based on the notion of *daimon*, goes back to Girard (2001). Zeilberger (2008b) enhances **CU** with a single daimonic command **Done**; e.g. we could define K such that its patterns always map to **Done** and hence $K\ \mathrm{inl}(())$ would reduce to **Done**. When considering logical properties of **CU**, Zeilberger only considers the *pure* system without **Done**. In summary, daimonic commands allow to introduce a "harmless" inconsistency in a controlled way; the next chapter makes further use of daimonic commands, including for structuring the soundness proofs for $\mathcal{PF}$.

Now that we have seen how reduction in **CU** works, let us consider if our quest for symmetry was successful: How does the extensibility duality concretely benefit from moving to the **CU** setting? Section 3.2.3 explained how, in the systems of Downen (2017) and Herbelin (2005), getting rid of a use of the CUT rule is a local version of cut elimination, relating consistency with the reduction relation in a direct way. In these systems, the command is the term assignment for the CUT rule. One of its components can be thought of as an abstraction, and the other as the entity that instantiates that abstraction (resulting in evidence for a contradiction). In the positive fragment of **CU**, a command is structured in the same way, however, the "abstraction" is now necessarily a continuation and the entity that "instantiates" it is now necessarily a value. That is, a command is still a meeting of an abstraction and some entity to "plug into", but this entity now has a canonical form and the abstraction (continuation) only matches on such canonical forms. This is reminiscent

[11]Since, as mentioned above, value variables are allowed in $\mathcal{PF}$, and, to support program transposition, actually only shallow matches are allowed, a concrete implementation of the type system supporting recursive types is ensured for $\mathcal{PF}$.

of first-order functions pattern matching on data type values, except that the result is another such explicit meeting of two forces rather than a term which may contain computation in a rather arbitrary way. There is a dual abstraction to continuations in the negative fragment (summarized in the next paragraph), which likewise results in a command when instantiated. As will be demonstrated in more detail in the next chapter, this *uniformity* simplifies the presentation of the extensibility duality in the **CU**-based $\mathcal{PF}$, as opposed to the natural deduction setting. As pointed out in Section 2.3.4, recognizing the symmetry there requires quite some lateral thinking and is certainly not directly apparent from the structure; this is inherently different in the **CU** setting.

The negative polarity fragment is completely *structurally isomorphic* to the positive fragment, with the structural properties of proofs and refutations exactly reversed: Refutations, called *covalues*, are now canonical, consisting of a pattern and a substitution just like values, where the pattern is a tree of observations. Proofs $\mu\phi$, called *expressions*, pattern match on such covalues and again provide evidence for a constradiction, i.e. a command (all specified in the map from patterns to commands ϕ, as with continuations). Commands for the negative fragment are combinations of expressions and covalues (again written with a juxtaposition); when combining the fragments to full **CU**, the two kinds of commands are simply merged into one syntactic entity and may be used wherever a rule from either fragment expects a command. The connectives for negative conjunction, disjunction, and negation are written &, $\parr$, and $\overset{n}{\neg}$[12], respectively. The pattern rules for negative conjunction are structurally the same as that for positive disjunction, and the same goes for negative disjunction and positive conjunction; negative and positive negation pattern rules are isomorphic to each other. The negative falsehood constant is written $\bot$ and has a pattern rule structurally identical to that of the positive truth constant 1; finally, there is no pattern rule for the negative truth constant $\top$ as that would mean that it is possible to refute constant truth (just as it is impossible to prove constant falsehood 0).

In Section 3.1, it was announced that in the foundational system we were seeking, implication would lose its special status to the more symmetric negation. Indeed, a function from A to B, which is a proof term for an implication, can be emulated in the negative fragment by an expression of type $\overset{n}{\neg}A \parr B$. The idea is that an input is itself an expression and not a covalue and hence its type must be embedded into a covalue type using the negative fragment negation $\overset{n}{\neg}$ to be able to combine it with the output type to a covalue type using the negative fragment disjunction ($\parr$).[13] Note that the covalue type judgment is a refutation judgment and this conceptually corresponds to the typing of the result of destructing codata, as mentioned earlier, and thus the covalue type judgment types an output. To further combine the implication expression to form more complex covalues (and from there, expressions), it has to be embedded into a covalue itself, which on the type level again employs negation, obtaining covalue type $\overset{n}{\neg}(\overset{n}{\neg}A \parr B)$. The high-level view of this is that negation, compared to implication, is conceptually more focused, giving rise to a greater economy of concepts: Where terms assigned for implication under a Curry-Howard interpretation, i.e. functions, conceptually involve an input and an output, negation *only has an input, or, perhaps more neutrally, only a single "port"*; implication is recovered by requiring what is connected to this port to be able to be split in two parts, or two

[12]Called *call-by-name negation* dually to positive call-by-value negation; again, refer to chapter 5 for details on recovering call-by-value and call-by-name.

[13]Erasing polarity, we get the usual definition $A \to B :\equiv \neg A \vee B$.

sources if you will, which are on the type level combined using $\parr$.[14] The part of the input of covalue type $\overset{\text{n}}{\neg}A$ is what is conceptually the input of the function modeled by the expression; it is of a negated form since to view an input as an output (of covalue type), it has to be negated (inverting the direction of the flow, if you will). The other part of the input of type B models the output of the function (and thus is not negated). Generalizing codata types to negative data types following the negative types of **CU** to make them entirely symmetric to positive data types, as is done in the next chapter, will allow us to define the function type exactly like this, as a combination of input and output.

3.3.5 Shifts

The final ingredient for **CU** are *shifts*, which allow to embed a positive, justified refutation (a continuation) into a negative refutation (a covalue) and conversely a negative, justified proof (an expression) into a positive proof (a value). Full **CU** consists of the rules for the positive and the negative fragment taken together plus the pattern rules for shifts shown below:

$$\frac{}{u \overset{\text{exp}}{:} N \Rightarrow u \overset{\text{val}}{:} \downarrow N} \qquad \frac{}{u \overset{\text{cnt}}{:} P \Rightarrow u \overset{\text{cov}}{:} \uparrow P}$$

Shifts can be used for the purpose of improving proof search performance, or more generally tackling the proof search space, as mentioned above. More relevant for our purposes, as we will see, being able to combine positive and negative types like this is essential for embedding different evaluation strategies of surface languages (like the data and codata languages considered in the previous chapter) and reflecting surface behavior by precise types in the core system.

In particular, one can describe data like natural numbers by some positive type $\mathbb{N}$, as one is used to from functional programming, and then model surface some term that types as a natural number, and that contains computation, by a negative expression of type $\uparrow \mathbb{N}$. As an initial example, assume that there is some surface-level function `add` with two $\mathbb{N}$ parameters and that this is translated to a function in the core system of the same name but with an additional continuation parameter of continuation type $\mathbb{N}$. The surface-level term $\texttt{add}(4, 2)$ is then translated to the following core term:

$$\mu\{k \mapsto \texttt{add}(4, 2, k)\}$$

The variable k has continuation type $\mathbb{N}$, and hence the expression that matches on it has expression type $\uparrow \mathbb{N}$. One can think of k as a "return" continuation that the expression passes control to when it is finished with the computation it contains.

In the foundational system $\mathcal{PF}$ presented next (Chapter 4), it is possible to *define* shifts as user-defined types (see Section 5.1.1). They can then be used, as mentioned, to recover surface languages in $\mathcal{PF}$; this approach will be studied in detail in Chapter 5.

[14]Closely related to this is the conception of the negative disjunction $\parr$ of linear logic as being related to parallelism computationally, which was originally pointed out by Girard (1987) and recently studied by Aschieri and Genco (2019).

Chapter 4

Polarized Flow: A Framework Consolidating both Symmetries

Disclaimer: The system $\mathcal{PF}$ presented in this chapter was jointly developed with David Binder and Ingo Skupin, with support from our advisor Klaus Ostermann. A version of $\mathcal{PF}$ forms the formal core of a manuscript currently under review for publication (as of December 2020). The specific formalization and systematic development of the soundness proofs in this chapter (Section 4.2, Section 4.4) are novel and have not been submitted for review for any publication. The same goes for the pragmatic extensions in Section 4.3 (these are ported from the language of Binder et al. (2019)).

This chapter introduces the foundational system *Polarized Flow* (or $\mathcal{PF}$ for short), which is in particular intended as a design framework for programming languages.[1] $\mathcal{PF}$ is a variation of the Calculus of Unity (**CU**) discussed at the end of the previous chapter, made to fit the matrix formalism for the extensibility duality discussed in chapter 2.

Chapter overview This chapter is structured as follows:

- Section 4.1 gives an informal introduction to $\mathcal{PF}$ and its central aspects.
- Section 4.2 formally defines $\mathcal{PF}$ and shows its soundness, following the logical intuitions behind sequent-calculus-inspired systems discussed in the previous chapter.
- Section 4.3 considers some extensions to make $\mathcal{PF}$ more practical, in particular demonstrating with some examples how these increase its potential to be leveraged for automated tooling, based upon ideas discussed in Section 2.4.1.
- Section 4.4 defines the extension of $\mathcal{PF}$ with parametric polymophism, following the approach discussed in Section 2.4.2.

Contributions

- The main contribution is the system $\mathcal{PF}$ itself, along with its formal details. (The applicability of $\mathcal{PF}$ as a framework for studying programming language design is discussed in the following chapters, chiefly by giving macro embeddings of surface languages into $\mathcal{PF}$.)

[1] The name *Polarized Flow* was chosen to indicate the polarized type setting and that computation in the system can be conceptually understood as the "flow" between producers and consumers; the concrete manifestation of this "flow" depends upon whether the producer or the consumer is of canonical form, and hence on polarity.

```
consumer function Nat.pred() := {        | function+ pred(k : ¬Nat) : ¬Nat := {
    zero() ⇒ zero()                      |     zero() ⇒ zero() ≫ k
    succ(m) ⇒ m                          |     succ(m) ⇒ m ≫ k
}                                        | }
```

FIGURE 4.1: Example data fragment program (left) and a corresponding $\mathcal{PF}$ program (right).

- In particular, that consumers and producers are structurally identical allows for an easy formalization of program transposition as one single, parametric, direction-agnostic transformation, with no lurking technical complications.
- A major part of the formal details in Section 4.2 is devoted to demonstrating how elegantly the soundness proof for $\mathcal{PF}$ can be carried out and how an intuitive understanding of this soundness is gleaned directly from the logical intuitions via Curry-Howard; in particular, there is arguably no technical difficulty involved in the core part of the proof, and the rest is some also rather straightforward plumbing. Therefore, the author also considers $\mathcal{PF}$ as a useful tool that allows to *decompose* a soundness proof for some surface language into a demonstration of the correspondence between the surface language into $\mathcal{PF}$ via a macro embedding and the elegant soundness proof for $\mathcal{PF}$; the latter can be reused while the former really carves out the specifics of the surface language as opposed to the common traits many languages share and which are captured by $\mathcal{PF}$.

4.1 Informal Introduction to Polarized Flow

4.1.1 Computation in $\mathcal{PF}$

To understand the central ideas behind the design of $\mathcal{PF}$, it is instructive to go back to the intuition behind computation initially considered in chapter 2, as a meeting of the two opposing forces of production and consumption. In the data and codata languages discussed in chapter 2, computation is triggered by a meeting of a constructor or producer function call and a destructor or consumer function call, e.g. `succ(zero()).pred()`. Such a term reduces in one step to the term (body) looked up in the producer or consumer function, with appropriate instantiation of the variables bound by the relevant pattern and those which are arguments of the function; when `pred` is a consumer function, it could be defined as shown on the left-hand side of Fig. 4.1 (returning the predecessor of the given number if it exists, or zero otherwise; the result of reducing the example term is thus `zero()`). We will abstractly refer to the terms that in our system represent the two opposing forces as producers and consumers; the following paragraphs will lead to a clear definition of these in $\mathcal{PF}$ which makes producers and consumers isomorphic. This is guided by the correspondence between producers and proofs and consumers and refutations hinted at in chapter 3, and the fact that the sequent calculus-inspired systems discussed in that chapter are not biased towards either proof or refutation.

As mentioned in chapter 2, constructors and destructors are structurally different (as are producer and consumer function signatures). Borrowing ideas discussed in chapter 3, particularly from **CU**, this difference is eliminated in $\mathcal{PF}$ by making the result of a destructor call (or consumer function call) *without* the consumed argument a first-class entity. In the example, `pred()` could now perhaps be such a first-class

entity (but see below), called a *negative value* if it is a destructor call, dual to *positive values* like succ(zero()) (when built using constructors). While a positive value is constructed using the constructors specified in the relevant data type, as usual, the negative value can be seen as a *first-class observation* which has the same structure as a data value, but is built up using the destructors specified in the relevant *negative data type*; we will from now on refer to the usual data types as *positive data types* and generally refer to either as a data type where the polarity (positive or negative) does not matter. Negative data types are not quite the same as the codata types we saw in chapter 2. A destructor signature now lacks a return type, like the second Nat in Nat.pred() : Nat. This makes constructor and destructor signatures structurally isomorphic: both consist of the type that is produced or consumed, together with a (possibly empty) list of argument types.

This leads us to the question: If, seemingly, there is no result of a destructor call (there is no result type given in the signature!), how does it even make sense to speak of computation? Well, first of all, observe how the situation is reminiscent of *continuation-passing style* (CPS), in which we also would not return results. Rather, an additional argument has to be provided, a *continuation*, into which the result is to be fed. Keeping this idea in the back of our minds, let us now turn to the syntactic entity of $\mathcal{PF}$ which actually directly represents a computation and over which the reduction relation is thus defined in $\mathcal{PF}$.

In $\mathcal{PF}$, a meeting of a producer and a consumer, which triggers computation, is of a separate syntactic form called a *command*. Logically, as explained in chapter 3, a producer corresponds to a proof of some statement/type A while a consumer corresponds to a refutation of some statement/type B, and thus a command, which is only well-formed when $A = B$, corresponds to a logical contradiction. Different from the data and codata languages we saw before, both producers and consumers are now first-class entities. There are two possible well-formed commands: In the positive variant, a positive value meets with a consumer function call, which is of a syntactic category called *positive continuation* and, like destructor calls, results in a first-class entity logically corresponding to a refutation. In the negative variant, a negative value meets a *negative continuation*, which is a producer function call which, like constructor calls, results in a first-class entity logically corresponding to a proof. Consequently, commands reduce, to other commands, by looking up the relevant case in the (producer or consumer) function definition for the constructor or destructor call. An example of a consumer function definition is shown on the right-hand side of Fig. 4.1; read on for an explanation of this.

Getting back to the problem of arriving at a "result", observe that consumers, i.e. positive continuations and negative values, are first-class entities (just like producers), thus they can also be passed as arguments to calls. And this is exactly how the result (type) is emulated: the destructor signature pred($\overline{\texttt{Nat}}$) takes the place of the original destructor signature from the codata type, where the bar over the type says that the expected argument is a consumer. Generally, every argument type in signatures in $\mathcal{PF}$ is specified to either be a producer or a consumer, which is distinguished notationally by putting a bar on top of the latter. And similarly to CPS, "emitting" a result requires to make a producer for that result meet with a consumer that was made available via passing it as an argument.[2]

[2]It should be noted that there is an important high-level conceptual difference between continuation-passing *style* and $\mathcal{PF}$ (note the emphasis): The former is only a particular way (a choice) of writing programs in languages in which one may also write them differently, while the latter gives the programmer no other choice but to employ continuations (or rather, consumers) in this way (not because of some static check that rules out other styles, but because $\mathcal{PF}$ *inherently* requires this).

The example term from above is emulated by the command $\texttt{succ}(\texttt{zero}()) \gg \texttt{pred}(k)$ (written with infix $\gg$ with the producer to its left and the consumer to its right), where k is a consumer assumed to be available in that context. The bodies of a producer or consumer functions (negative and positive continuations) are also commands (they have to be since commands reduce to commands), and k could thus for instance be an argument of such a function. For example, on the right-hand side of Fig. 4.1, there is the definition of the consumer function `pred` (in $\mathcal{PF}$), which consumes a `Nat` and pattern matches on this input, and which additionally has a continuation argument of consumer type $\overline{\texttt{Nat}}$ to pass the result to, e.g., the command for the first case of `pred` combines the producer $\texttt{zero}()$ (the "result") with the consumer argument variable k.

4.1.2 Daimons

In $\mathcal{PF}$, there is no implicitly assumed "outer" consumer like, the user who "looks at" the result; rather, it must always be explicitly specified what should "happen next" with the result (which is again similar to the CPS perspective). This raises another question, however: Is it at all possible to write a closed program, i.e. one containing no free variables? The answer to this question is simply "no"; this is again similar to CPS, where we always need an "outer continuation" if we want to be able to run the program somehow.

Now, while one might be content with this situation and e.g. use a form of symbolic execution that stops when the result is the command $v \gg k$ where v is a concrete value and k is the open symbol standing for the "outer" consumer, in practice it would be nice to have a less non-standard way of running a program. For this purpose we can use *daimonic commands*, based on the concept of *daimon* (Girard, 2001), which we already saw in action when "challenging" the *tertium non datur* proof/program in Section 3.2.5. A very simple daimonic command is **Result**, which is intended for the purpose of simulating the situation known from the usual declarative languages in which the end result is implicitly assumed to be "looked at" by the user; when this final point of the computation has been reached, instead of triggering more computation by having producer and consumer meet, this "look-at-me" daimon is triggered by $\textbf{Result}(v)$ (where v is a concrete positive value that is this "end result"). We will also employ daimons for other extra-logical aspects, where reducing daimonic commands "passes control to the daimon" (instead of making producer and consumer meet) in order to, e.g., handle I/O, mutate state, or make "native" calls,[3] but also to structure our system into a core part which has no recursion facilities and a recursion daimon.

The purpose of modelling extra-logical aspects deserves some illustration, for which we consider a simple I/O daimon specification. For our purposes, a specification of a daimon requires at least the syntax of its daimonic commands and reduction rules for these. The I/O daimon has two daimonic commands for reading from some source and writing to some target (that we assume to be specified somehow): **Read** and **Write**. These come equipped with the following reduction rules, from

[3] This idea is similar to the concept of a "central administrator" as introduced by Cartwright and Felleisen (1994) for their semantic framework for extensible denotational language specifications. The administrator takes care of effects that the interpreter propagates to it ("the meaning of a program is defined as the composition of the interpreter and the administrator" (Cartwright and Felleisen, 1994)), i.e., of effects that can be considered extra-logical.

Console			Evaluation
`> 39`	⏎		$\mathbf{Read}(\tilde{\mu}\{n \Rightarrow \mathbf{Read}(...)\})$
		$\triangleright$	$39 \gg \tilde{\mu}\{n \Rightarrow \mathbf{Read}(\tilde{\mu}\{m \Rightarrow n \gg \mathtt{add}(m, ...)\})\}$
`> 3`	⏎	$\triangleright$	$\mathbf{Read}(\tilde{\mu}\{m \Rightarrow 39 \gg \mathtt{add}(m, ...)\})$
		$\triangleright$	$3 \gg \tilde{\mu}\{m \Rightarrow 39 \gg \mathtt{add}(m, ...)\}$
		$\triangleright$	$39 \gg \mathtt{add}(3, \tilde{\mu}\{r \Rightarrow \mathbf{Write}(r, ...)\})$
		$\triangleright^*$	$\mathbf{Write}(42, \tilde{\mu}\{_ \Rightarrow \mathbf{Done}\})$
`42`		$\triangleright$	$42 \gg \tilde{\mu}\{_ \Rightarrow \mathbf{Done}\}$
		$\triangleright$	$\mathbf{Done}$
`Process ended.`			

FIGURE 4.2: Console interaction example for I/O daimon.

$\mathtt{Nat}$	$\mathtt{zero}()$	$\mathtt{succ}(m : \mathtt{Nat})$
$\mathtt{pred}(k : \overline{\mathtt{Nat}})$	$\mathtt{zero}() \gg k$	$m \gg k$
$\mathtt{out_cns}()$	$\mathbf{Result}(\mathtt{zero}())$	$\mathbf{Result}(\mathtt{succ}(m))$

FIGURE 4.3: Example $\mathcal{PF}$ program in matrix form (only one matrix, for `Nat`).

which one can also see which type the arguments of the daimonic commands have:

$$\mathbf{Read}(k : \overline{\mathtt{Nat}}) \triangleright n \gg k, \quad n \text{ read from the input}$$
$$\mathbf{Write}(n : \mathtt{Nat}, k : \overline{\mathtt{Nat}}) \triangleright n \gg k, \quad n \text{ written to the output}$$

For simplicity, we let the commands read and write natural numbers `Nat` (which can be specified by the usual Peano data type). **Read** reads from the input source and passes the result to the given continuation. **Write** writes the given number n to the output target, then passes n to the given continuation.

For instance, we can implement the "daimonic" input source and output target by the standard input and output streams of some console. The command[4]

$$\mathbf{Read}(\tilde{\mu}\{n \Rightarrow \mathbf{Read}(\tilde{\mu}\{m \Rightarrow n \gg \mathtt{add}(m, \tilde{\mu}\{r \Rightarrow \mathbf{Write}(r, \tilde{\mu}\{_ \Rightarrow \mathbf{Done}\})\})\})\})$$

can then be interactively executed as shown in Fig. 4.2.

Section 4.2.3 considers formal details that generally apply to daimons and that are relevant for the soundness of $\mathcal{PF}$, and also gives another example of a daimon, for mutable state.

4.1.3 Program transposition for $\mathcal{PF}$

Before turning to the formal details, let us look at how our system fares symmetry-wise. We already saw that producers and consumers, which logically correspond to proofs and refutations, respectively, are structurally identical. It turns out that this also simplifies the presentation of the other duality considered in this work, the extensibility duality, in particular making program transposition in both directions, i.e. from data to codata and vice versa, one single, parametric transformation.

Following the ideas from Chapter 2, we can view programs as matrices where (for instance) the columns are labelled with producer signatures and the rows are labelled with consumer signatures. In the next section, we will use this formalism for $\mathcal{PF}$. An example $\mathcal{PF}$ program in this matrix form is shown in Fig. 4.3.[5] Generally, programs consist of multiple matrices, one per type; in this simple example we have only one matrix, for type `Nat`. Linearizing this matrix by reading it row-for-row

[4] The $\tilde{\mu}$ construct is used for a local version of consumer functions (details in Section 4.3).

[5] This matrix is for the program in Fig. 4.1, with one trivial consumer `out_cns` added.

$$
\begin{array}{l}
\mathbf{data}_{+}\ \texttt{Nat}\ \{\ \texttt{zero}()\ |\ \texttt{succ}(\texttt{Nat})\ \}\\
\\
\mathbf{function}_{+}\ \texttt{pred}(k : \overline{\texttt{Nat}}) : \overline{\texttt{Nat}}\ := \{\\
\quad \texttt{zero}() \ \Rightarrow\ \texttt{zero}() \gg k\\
\quad \texttt{succ}(m) \ \Rightarrow\ m \gg k\\
\}\\
\\
\mathbf{function}_{+}\ \texttt{out_cns}() : \overline{\texttt{Nat}}\ := \{\\
\quad \texttt{zero}() \ \Rightarrow\ \mathbf{Result}(\texttt{zero}())\\
\quad \texttt{succ}(m) \ \Rightarrow\ \mathbf{Result}(\texttt{succ}(m))\\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{data}_{-}\ \texttt{Nat}\ \{\ \texttt{pred}(\overline{\texttt{Nat}})\ |\ \texttt{out_cns}()\ \}\\
\\
\mathbf{function}_{-}\ \texttt{zero}() : \overline{\texttt{Nat}}\ := \{\\
\quad \texttt{pred}(k) \ \Rightarrow\ \texttt{zero}() \gg k\\
\quad \texttt{out_cns}() \ \Rightarrow\ \mathbf{Result}(\texttt{zero}())\\
\}\\
\\
\mathbf{function}_{-}\ \texttt{succ}(m : \texttt{Nat}) : \texttt{Nat}\ := \{\\
\quad \texttt{pred}(k) \ \Rightarrow\ m \gg k\\
\quad \texttt{out_cns}(m) \ \Rightarrow\ \mathbf{Result}(\texttt{succ}(m))\\
\}
\end{array}
$$

FIGURE 4.4: The two possible linearizations for the example $\mathcal{PF}$ program from Fig. 4.3 (top: positive polarity, bottom: negative polarity).

gives us the program with the positive data type and consumer functions for this type, in Fig. 4.4 (top); a column-by-column linearization yields the program with the negative data type and producer functions of this type (Fig. 4.4, bottom).

Producer and consumer signatures are structurally identical in $\mathcal{PF}$, which streamlines the formal presentation and in particular the definition of transposition. For this definition we consider programs as maps from type names to matrices which are labelled by signatures and the cells of which are filled with commands; the program is well-formed if all of its commands are, which in turn means that the constructor/destructor and function calls within them comply with the signatures. Each matrix is to be understood as representing (consumer or producer) functions, read off row-by-row (we could also choose column-by-column, for the sake of the following it is only important that this is fixed globally). To each matrix it is annotated whether it is of positive polarity (for a positive data type) or of negative polarity (for a negative data type). Depending on this, we read the rows as consumer or as producer functions. Transposing with respect to some type T is now just flipping the polarity annotation for the matrix for T and transposing that matrix.

In particular, no technical tinkering is necessary, other than with the program transposition for the data and codata languages of Chapter 2, where the structural difference between consumer and producer signatures required one to be careful in the precise formalization of the transformations, distinguishing the direction of the transformation, i.e. from data to codata or from codata to data. In the $\mathcal{PF}$ setting, the simple description from above translates easily into a single, direction-agnostic formal definition of transposition parametric in the polarity; there is no need to inspect the polarity other than to flip the polarity annotations. Also, with this simple definition of transposition the proofs that operational semantics and typing (program well-formedness) are preserved by it are trivial. More precisely, such proofs actually become unnecessary since we can just view programs as matrices in the first place (the matrix being the ground truth, if you will), with data and codata programs merely being linearizations of matrices; it is clear that transposition as described above is equivalent to switching between linearizations and hence inherently preserves our desired properties. Following this idea, the next section formalizes programs as collections of matrices.

4.2 Core Polarized Flow

We now first present a practical version of $\mathcal{PF}$ which allows unrestricted recursion. It is this pragmatic form of the system that we will use in later sections and the next chapter; in particular, the practical version is the one we employ when we consider transposition, since restricting recursion would also complicate transposition. We then present a logical core form of the system, which fixes the polarity (and hence linearization) of individual matrices and restricts recursive references to only functions "below" the call site. Next, we present how to daimonically extend the logical core, and prove soundness (including progress and preservation) in these settings (first for the logical core, then extending this result to daimonic extensions). Finally, we translate the practical version into a certain daimonic extension (for unrestricted recursion) of the logical core, thereby demonstrating progress and preservation for the practical version. Figure 4.5 gives an overview of the soundness proofs.

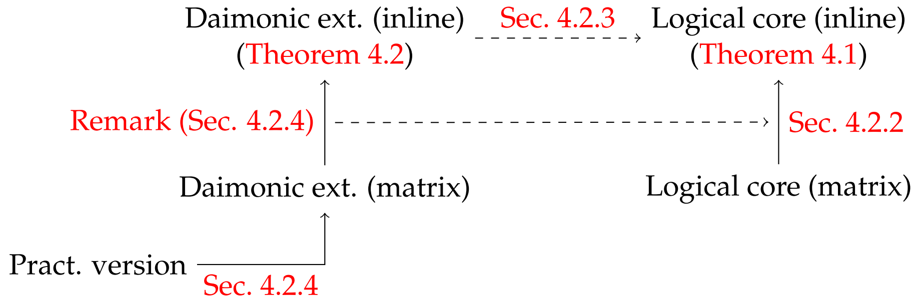

FIGURE 4.5: Overview of soundness proofs.

4.2.1 Practical version

We start with the definition of a *program* as a collection (more precisely, a map with types as its domain) of program matrices, as shown in Fig. 4.6b, plus the term syntax, shown in Fig. 4.6a.[6] The typing rules for the practical version are shown in Fig. 4.7 (with term typing rules in Fig. 4.7a and program wellformedness rules in Fig. 4.7c), and its reduction rules in Fig. 4.8. Reduction of commands happens with respect to a program $\mathcal{P}$, identified by a subscript $\mathcal{P}$ when not clear from the context. Also, when not clear from context, we will distinguish the reduction of the practical version by a superscript 0, i.e. writing it as $\triangleright^0$.

The term syntax and typing rules formally capture the intuitions conveyed in the informal overview above. Especially, commands C consist of a producer p and a consumer c (of the same type, i.e. one of producer type T and one of consumer type $\overline{T}$; cf. rule T-CMD), which syntactically both have the same form $\mathsf{X}\sigma$ (the identifier t is used in all places where the distinction between producer and consumer does not matter[7]). Here X is some generic *label name*, i.e. the name of some signature $\mathsf{X}\Xi$ that appears as a label in a matrix (i.e. it is in the program's global signature context Σ, collected via rule M-SIG, denoted as being available to the term typing judgment by a subscript; cf. rule P-OK), which means that it refers to some constructor, destructor, or (producer or consumer) function in a linearized version of the program. In order for a (producer or consumer) term $\mathsf{X}\sigma$ to typecheck (rule T-X, parametric in judgment $\mathcal{J}$, generically covers this), the substitution σ (a map from variables to terms) has to fit with the relevant signature's argument context Ξ (rule T-SUBST; an argument context maps variables to types). Where they appear in a term, producer and consumer variables have to be bound in the relevant matrix field, by the argument context (Ξ_p or Ξ_c) of the row or column label (rule M-OK checks the command in the matrix field in the context combining Ξ_p and Ξ_c and rule T-VAR looks up the variable in the context).

[6] The vector notation $\overrightarrow{X_i}^i$ is used, throughout the rest of this work, for a sequence $X_1, ..., X_i, ..., X_n$ (following Downen and Ariola (2014)). When clear from the context the index i is left implicit (as in $\overrightarrow{X}$). In this formalization all collections are ordered sequences (order does not always matter, but is always safe to assume), written using set notation ($\{...\}$); set operators like $\times$ apply to these sequences in the obvious way (preserving the order).

[7] The identifiers p, c, and t are merely distinguished to indicate where a producer or consumer is expected or where this does not matter. Whether some term is a producer or consumer is *not* actually determined by the syntax, but rather by the signatures in the global signature context Σ of the program (by looking up X in Σ).

$T \in$ type names, $\mathsf{X} \in$ label names, $x \in$ variable names *(Names)*
(var. naming convention: plain (e.g. x, y) for prd, overbar (e.g. $\bar{x}$, $\bar{y}$) for cns)

$$\begin{array}{lll} \mathcal{J} & ::= & \mathrm{prd} \mid \mathrm{cns} \\ \Xi & ::= & \overrightarrow{x \overset{\mathcal{J}}{:} T} \end{array}$$

(Judgments)
(Argument context)

(shorthands: $x : T$ for $x \overset{prd}{:} T$, $\bar{x} : \overline{T}$ for $\bar{x} \overset{cns}{:} T$)

$$\begin{array}{lll} \Sigma^{\mathrm{elem}} & ::= & \{\mathrm{prd} \mapsto \overrightarrow{\mathsf{X}\Xi}, \mathrm{cns} \mapsto \overrightarrow{\mathsf{X}\Xi}\} \\ \Sigma & ::= & \overrightarrow{T \mapsto \Sigma^{\mathrm{elem}}} \end{array}$$

(Signature element)
(Signature)

$$\begin{array}{lll} \sigma & ::= & \overrightarrow{x \mapsto t} \\ p, c, t & ::= & \mathsf{X}\sigma \mid x \\ C & ::= & p \gg c \mid \mathfrak{D} \end{array}$$

(Substitutions)
(Producers / Consumers)
(Commands)

(A) Terms (common). Daimonic commands $\mathfrak{D}$ customizable.

$$\begin{array}{lll} \mathcal{M}^{\mathrm{cells}} & ::= & \overrightarrow{(\mathsf{X}\Xi, \mathsf{X}\Xi) \mapsto C} \\ \mathcal{M} & ::= & (\mathcal{M}^{\mathrm{cells}}, \overrightarrow{\mathsf{X}\Xi}, \overrightarrow{\mathsf{X}\Xi}) \\ \mathcal{P} & ::= & \overrightarrow{T \mapsto \mathcal{M}} \end{array}$$

(Program matrix, unlabeled)
(Program matrix, labeled)
(Program)

(B) Programs (practical version).

$$\mathcal{M}^{\mathrm{cells}} \;::=\; \overrightarrow{\mathsf{X}\Xi \mapsto \overrightarrow{\mathsf{X}\Xi \mapsto C}}$$ *(Program matrix, unlab.)*

(If the outer X_i are for producers, $\mathcal{M}^{cells}$ is of negative polarity.
If the outer X_i are for consumers, $\mathcal{M}^{cells}$ is of positive polarity.)

$$\begin{array}{lll} \mathcal{M} & ::= & (\mathcal{M}^{\mathrm{cells}}, \overrightarrow{\mathsf{X}\Xi}, \overrightarrow{\mathsf{X}\Xi}) \\ \mathcal{P} & ::= & \overrightarrow{T \mapsto \mathcal{M}} \end{array}$$

(Program matrix, labeled)
(Program)

(C) Programs (logical core).

FIGURE 4.6: Syntax of $\mathcal{PF}$.

Figure 4.9 shows an example program[8] together with a reduction sequence relative to that program. Here we assume the simple daimonic command former **Result** (the same as informally used above), which does not have any custom reduction rules, i.e. **Result** simply ends the reduction, simulating the behavior of reaching a value known from natural deduction. The typing rule for **Result** allows to typecheck any command of the form **Result**(v) (in any context), with the only premise that v typechecks in the empty context.

4.2.2 Logical core

Next we present the syntax of the logical core, shown in Fig. 4.6c for programs and matrices; the term syntax is reused from Fig. 4.6a (though daimonic commands are excluded for now and added in the next subsection). The program/matrix syntax is also almost identical to that of the practical version except, importantly, for an explicit orientation (polarity) of the matrices which distinguishes function definitions from xtors. Unlike the practical version of $\mathcal{PF}$, the program wellformedness rules of the logical core system, shown in Fig. 4.7d (term typing rules are identical to those for the practical version, see Fig. 4.7a), prohibit unrestricted recursion, by actually

[8] In left-hand sides of matrix entries, types are omitted from argument types (they are redundant since the label lists, i.e. lists of $\mathsf{X}\Xi$, are also separately specified); also, these left-hand sides are written in the usual first-order function application style, e.g. `succ`(x) instead of `succ`$\{x : \texttt{Nat}\}$.

$$\frac{\mathsf{X}\Xi \in \Sigma(T)(\mathcal{J}) \qquad \Gamma \vdash_\Sigma \sigma : \Xi}{\Gamma \vdash_\Sigma \mathsf{X}\sigma \overset{\mathcal{J}}{:} T}\ \text{T-X} \qquad \frac{x \overset{\mathcal{J}}{:} T \in \Gamma}{\Gamma \vdash_\Sigma x \overset{\mathcal{J}}{:} T}\ \text{T-VAR}$$

$$\frac{\Gamma \vdash_\Sigma p \overset{\text{prd}}{:} T \qquad \Gamma \vdash_\Sigma c \overset{\text{cns}}{:} T}{\Gamma \vdash_\Sigma p \gg c \overset{\text{cmd}}{:} \#}\ \text{T-CMD} \qquad \frac{\overrightarrow{\Gamma \vdash_\Sigma t \overset{\mathcal{J}}{:} T}}{\Gamma \vdash_\Sigma \overrightarrow{x \mapsto t} : \overrightarrow{x \overset{\mathcal{J}}{:} T}}\ \text{T-SUBST}$$

(A) Term typing (common). For daimonic commands $\mathfrak{D}$ custom rules are added.

$$\frac{\mathcal{M} = (\mathcal{M}^{\text{cells}}, \overrightarrow{\mathsf{X}_p\Xi_p}, \overrightarrow{\mathsf{X}_c\Xi_c}) \quad L = \overrightarrow{\mathsf{X}_p\Xi_p} \times \overrightarrow{\mathsf{X}_c\Xi_c} \qquad \text{dom}(\text{uncurry}(\mathcal{M}^{\text{cells}})) \in \{L, L^\top\}}{\vdash \mathcal{M}\ \text{SIG}(\{\text{prd} \mapsto \overrightarrow{\mathsf{X}_p\Xi_p}, \text{cns} \mapsto \overrightarrow{\mathsf{X}_c\Xi_c}\})}\ \text{M-SIG}$$

(B) Signature judgment. Gray-text parts only apply to logical core.

$$\frac{\overrightarrow{\Xi_p, \Xi_c \vdash_\Sigma C \overset{\text{cmd}}{:} \#}}{\vdash_\Sigma ((\overrightarrow{\mathsf{X}_p\Xi_p, \mathsf{X}_c\Xi_c}) \mapsto C, \ldots, \ldots)\ \text{OK}}\ \text{M-OK}$$

$$\frac{\overrightarrow{\vdash \mathcal{M}\ \text{SIG}(\Sigma^{\text{elem}})} \qquad \Sigma = \overrightarrow{T \mapsto \Sigma^{\text{elem}}} \qquad \overrightarrow{\vdash_\Sigma \mathcal{M}\ \text{OK}}}{\vdash \overrightarrow{T \mapsto \mathcal{M}}\ \text{OK}}\ \text{P-OK}$$

(C) Well-formedness (practical version).

$$\frac{\vdash \mathcal{P}\ \text{OK}}{\vdash \{T \mapsto \varnothing\} \cup \mathcal{P}\ \text{OK}}\ \text{P-MEMPTY-OK}$$

$$\frac{\mathcal{P}\ \text{DECOMPOSE}(\mathsf{X}\Xi \mapsto \overrightarrow{\mathsf{X}'_i\Xi'_i \mapsto C_i}^i; \overrightarrow{T \mapsto \mathcal{M}}) \qquad \overrightarrow{\vdash \mathcal{M}\ \text{SIG}(\Sigma^{\text{elem}})} \quad \vdash \overrightarrow{T \mapsto \mathcal{M}}\ \text{OK} \qquad \Sigma = \overrightarrow{T \mapsto \Sigma^{\text{elem}}} \quad \overrightarrow{\mathsf{X}\Xi, \mathsf{X}'_i\Xi'_i \vdash_\Sigma C_i \overset{\text{cmd}}{:} \#}^i}{\vdash \mathcal{P}\ \text{OK}}\ \text{P-OK}$$

(D) Well-formedness (logical core).

$$\frac{\mathcal{M} = (\{\mathsf{X}\Xi \mapsto m\} \cup \mathcal{M}^{\text{cells}}, \overrightarrow{\mathsf{X}_p\Xi_p}, \overrightarrow{\mathsf{X}_c\Xi_c}) \qquad \mathcal{M}_0 = (\mathcal{M}^{\text{cells}}, \overrightarrow{\mathsf{X}_p\Xi_p} \setminus \{\mathsf{X}\Xi\}, \overrightarrow{\mathsf{X}_p\Xi_c} \setminus \{\mathsf{X}\Xi\})}{\mathcal{M}\ \text{DECOMPOSE}(\mathsf{X}\Xi \mapsto m; \mathcal{M}_0)}\ \text{M-DCP}$$

$$\frac{\mathcal{P} = \{T \mapsto \mathcal{M}\} \cup \mathcal{P}_0 \qquad \mathcal{P}_0 = \{T \mapsto \mathcal{M}_0\} \cup \mathcal{P} \qquad \mathcal{M}\ \text{DECOMPOSE}(\mathsf{X}\Xi \mapsto m; \mathcal{M}_0)}{\mathcal{P}\ \text{DECOMPOSE}(\mathsf{X}\Xi \mapsto m; \mathcal{P}_0)}\ \text{P-DCP}$$

(E) Decomposition auxiliary (logical core).

FIGURE 4.7: Typing rules for $\mathcal{PF}$.

not allowing recursion at all, which requires this distinction (since the recursion issue does not apply to xtors). Specifically, the typing rules demand there to be an order among the matrices and, within each matrix, among the rows, which each comprise a function definition. A field in a row may only refer to functions defined below its row, thus excluding any cyclic references; cf. rule P-OK which typechecks this "top row" (containing the C_i) given the signatures (Σ) computed (via rule M-SIG) for the rest of the program without that row[9]), and which separately checks ($\vdash -$ OK) that rest of the program recursively.[10]

When we now consider commands relative to some given program, as we did for the practical version, we can straightforwardly simplify the presentation by transitively inlining all function definitions as local abstractions. This is possible since we

[9]Obtained via auxiliary judgment DECOMPOSE (see Fig. 4.7e) which decomposes a matrix into the "top row" m (containing the C_i) and the remainder.

[10]Rule P-MEMPTY-OK is for the base case, i.e. an empty "top" matrix, where the program is well-formed whenever the program without that empty matrix is.

$$\begin{array}{lll} \mathsf{X}_p\sigma_p \gg \mathsf{X}_c\sigma_c & \triangleright_{\mathcal{P}} & C\sigma_p\sigma_c, \\ \quad \text{if there exist } T, C \text{ such that } \mathcal{P}(T)(\mathsf{X}_p,\mathsf{X}_c) = C & & \\ \mathfrak{D} & \triangleright_{\mathcal{P}} & \textit{user-defined} \end{array}$$

FIGURE 4.8: Reduction rules for the practical version of $\mathcal{PF}$ (identified by superscript 0 where necessary).

$$\begin{array}{lll} \mathcal{M}^{\text{cells}}_{\texttt{Nat}} & := & \{ \; (\texttt{zero}, \texttt{add}(y,\bar{y})) \mapsto y \gg \bar{y}, \\ & & \quad (\texttt{succ}(x), \texttt{add}(y,\bar{y})) \mapsto x \gg \texttt{add}(y, \texttt{succCns}(\bar{y})), \\ & & \quad (\texttt{zero}, \texttt{succCns}(\bar{y})) \mapsto \texttt{succ}(\texttt{zero}) \gg \bar{y}, \\ & & \quad (\texttt{succ}(x), \texttt{succCns}(\bar{y})) \mapsto \texttt{succ}(\texttt{succ}(x)) \gg \bar{y}, \\ & & \quad (\texttt{zero}, \texttt{doneCns}) \mapsto \textbf{Result}(\texttt{zero}), \\ & & \quad (\texttt{succ}(x), \texttt{doneCns}) \mapsto \textbf{Result}(\texttt{succ}(x)) \\ & & \} \\ \Sigma^{\text{elem}}(\text{prd}) & := & \{\texttt{zero}\{\}, \texttt{succ}\{x : \texttt{Nat}\}\} \\ \Sigma^{\text{elem}}(\text{cns}) & := & \{\texttt{add}\{y : \texttt{Nat}, \bar{y} : \overline{\texttt{Nat}}\}, \texttt{succCns}\{\bar{y} : \overline{\texttt{Nat}}\}, \texttt{doneCns}\{\}\} \\ \mathcal{P}_{\text{ex}} & := & \{\texttt{Nat} \mapsto (\mathcal{M}^{\text{cells}}_{\texttt{Nat}}, \Sigma^{\text{elem}}(\text{prd}), \Sigma^{\text{elem}}(\text{cns}))\} \end{array}$$

Reduction steps:

$$\begin{array}{ll} & \texttt{succ}(\texttt{succ}(\texttt{zero})) \gg \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{doneCns}) \\ \triangleright & \texttt{succ}(\texttt{zero}) \gg \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{succCns}(\texttt{doneCns})) \\ \triangleright & \texttt{zero} \gg \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{succCns}(\texttt{succCns}(\texttt{doneCns}))) \\ \triangleright & \texttt{succ}(\texttt{zero}) \gg \texttt{succCns}(\texttt{succCns}(\texttt{doneCns})) \\ \triangleright & \texttt{succ}(\texttt{succ}(\texttt{zero})) \gg \texttt{succCns}(\texttt{doneCns}) \\ \triangleright & \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero}))) \gg \texttt{doneCns} \\ \triangleright & \textbf{Result}(\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero})))) \end{array}$$

FIGURE 4.9: Addition example.

prohibited recursive references. So, instead of a command C which contains a call to some function $f(\overrightarrow{t})$ specified in program $\mathcal{P}$, we consider C with the call to f replaced by an inlined function definition $\mu(\overrightarrow{x \mapsto t})\{\overrightarrow{\mathsf{X}\Xi \Rightarrow C}\}$. The C in that local abstraction are taken from the matrix row for f; the concrete arguments $\overrightarrow{t}$ of f at the call site are annotated to the μ as an explicit substitution (to be substituted into the C). We use μ for producer abstractions (negative) and $\bar{\mu}$ for consumer abstractions (positive).

It is clear that this presentation of the logical core is equivalent to the matrix-based one, but that it looks much more like a conventional logical calculus. Therefore we now use it to demonstrate the soundness of this system. Especially, instead of requiring the entire signature of the implicitly assumed program for typechecking the considered command, it suffices to have an implicitly assumed set of xtor signatures (for each positive and negative type in the program).

The reduction rules for the reduction $\triangleright$ for the system are shown in Fig. 4.10, but let us first motivate them from a logical perspective. We can directly express the soundness of the system:

Theorem 4.1 (Soundness)**.** *There is no command C such that $\vdash C \overset{cmd}{:} \#$.*

In other words, there is no way to obtain evidence for a contradiction (as $\vdash C \overset{\text{cmd}}{:} \#$ says that C is evidence for a contradiction). Now, how do we prove the soundness theorem? Basically, we shrink the evidence until it disappears, and our reduction steps are an internalization of the shrinking steps we use to demonstrate this. More precisely, we prove the following lemma, from which the soundness theorem immediately follows, since it means that the number of commands appearing in the

$$\mu(\sigma_p)\{\overrightarrow{\mathsf{X}_i\Xi \Rightarrow C_i}^i\} \gg \mathsf{X}_i\sigma_c \quad \triangleright \quad C_i\sigma_c\sigma_p$$
$$\mathsf{X}_i\sigma_p \gg \tilde{\mu}(\sigma_c)\{\overrightarrow{\mathsf{X}_i\Xi \Rightarrow C_i}^i\} \quad \triangleright \quad C_i\sigma_c\sigma_p$$

FIGURE 4.10: Reduction rules for the logical core of $\mathcal{PF}$.

substructure can be iteratively made arbitrary small, even smaller than 1, which is impossible.

Lemma 4.1 (Evidence shrinking)**.** *Denote the number of* syntactically different *commands appearing in the substructure of some command C as* $\#(C)$*. Now, consider any command C with* $\vdash C \overset{cmd}{:} \#$*. We can find another command* C' *with* $\vdash C' \overset{cmd}{:} \#$ *and* $\#(C') < \#(C)$*.*

To prove the lemma, we do two things. First, we define the reduction rules for commands (Fig. 4.10), which gives us exactly the first part of what we need: A way to obtain a command C' containing at least one fewer command, up to syntactic equality, in its substructure than any given command C.[11] This can be rather straightforwardly seen in the right-hand sides by calculating their amount of commands relative to that of the left-hand sides.[12] Second, we need to show that the newly obtained command C' actually is evidence for a contradiction, for which we use the knowledge that C is such evidence. That is, we need to show that reduction preserves the command typing.

Lemma 4.2 (Preservation)**.** *For all commands* C, C'*, if* $\vdash C \overset{cmd}{:} \#$ *and* $C \triangleright C'$*, then* $\vdash C' \overset{cmd}{:} \#$*.*

Proof. We know C' is a right-hand side C_i inside a μ or $\tilde{\mu}$ with some substitution σ applied to it, i.e. $C' = C_i\sigma$. Since we assume the relevant program to be well-formed, C_i (which comes from a matrix cell in the matrix form of the program) typechecks in some argument context Ξ. By the command substitution lemma, which we leave for the appendix (Lemma A.2), we can conclude that $C_i\sigma$ typechecks if $\vdash \sigma : \Xi$. Since C typechecks and σ consists of two parts which are respectively applied to an xtor call and used as an explicit substitution for a μ- or $\tilde{\mu}$-abstraction, both part of C, by inversion we know that $\vdash \sigma : \Xi'$ for some argument context Ξ'. But this is necessarily the same as Ξ, since it is the cell for this xtor and this abstraction from which we got the C_i. □

[11] To be precise, a reduction rule can only be applied when the lookup of the xtor in the $\mu/\tilde{\mu}$ abstraction succeeds. Since we assume the command we want to reduce to typecheck relative to the original matrix program $\mathcal{P}$, we know this to be the case, i.e. *progress* holds, if the cases listed in the abstraction are complete for the signatures from $\mathcal{P}$. And this we also know to hold since we only consider commands created by inlining function definitions from these matrices.

[12] We demonstrate this for the first reduction rule (proceed similarly for the second rule). We have (where we use the command count #(...) for other syntactic domains as well, defined in the expected way):

$$\#(C_i\sigma_c\sigma_p) \leq \#(C_i\sigma_p) + \#(\mathsf{X}_i\sigma_c) - O_i$$
$$\leq \#(\mu(\sigma_p)\{\overrightarrow{\mathsf{X}_i\Xi \Rightarrow C_i}^i\}) + \#(\mathsf{X}_i\sigma_c) - O < \#(\mu(\sigma_p)\{\overrightarrow{\mathsf{X}_i\Xi \Rightarrow C_i}^i\}) + \#(\mathsf{X}_i\sigma_c) - O + 1 = \#(C'),$$

where C' is the left-hand side, O is the overlap within C' (i.e., the number of commands appearing on both sides of the top-most $\gg$ in C'), and O_i is the overlap limited to comparing σ_c with only $C_i\sigma_p$. In particular, the first inequality holds because (1) while a variable in $\mathrm{dom}(\sigma_c)$ may appear more than once in $C_i\sigma_p$, the number of syntactically different commands in $C_i\sigma_p\sigma_c$ is still the same as if the variable only appeared once, and (2) variable bindings do not cross binders (due to the explicit substitution syntax), so the terms substituted by σ_c and σ_p do not potentially increase the count of different commands by now appearing deeper inside the substructure.

Note that the *goal* of the process of eliminating commands until none are left is analogous to that of the complete elimination of uses of cut rules, or cut elimination, of Gentzen (1935) (with commands and cut rules being the only ways of obtaining evidence for a contradiction in their respective systems). However, there is no judgmental organization in the system LK of Gentzen (1935) that enforces cuts to be only obtainable from a meeting of a producer and a consumer (logically, of a left and a right rule for the same connective; cf. Footnote 2 in Section 3.2.3). Thus, LK inherently requires a quite more elaborate cut elimination proof: Gentzen (1935) does not always just completely remove a cut in each of his proof steps. In some cases he instead, e.g., transforms the proof tree such that a subproof leading to a cut becomes smaller (iteratively leading to the cut disappearing eventually). But in our setting it suffices to show that there is no evidence for any possible closed instance of a command, which could only be obtained as a meeting of a producer and a consumer, and hence we were able to simply employ the shrinking argument.

In using the reduction relation to demonstrate consistency, we follow Zeilberger (2008b), who however does not separately prove consistency for the "vacuous" (Zeilberger, 2008b) purely logical part of **CU** without daimonic commands. Methodologically, he is interested in demonstrating the practically relevant progress and preservation properties for **CU** with the "Done" daimon (with a "runnable" reduction), and does not approach consistency as fundamentally (conceptually starting from the meta-logical property of non-derivability for the judgment) as we did above (with progress and preservation being "tools" to achieve this conceptually "prior" property). Therefore shrinking towards disappearance does not explicitly occur in his proof: He does not explicitly prove that there is no closed command in the pure system (though he does mention this property and it is rather straightforward to prove for **CU**, similarly to the proof for $\mathcal{PF}$). We, however, will utilize the rather simple proof of this section, which demonstrates just that for our logical core, as a starting point which we systematically enhance to arrive at the consistency proof for the daimonic setting, thereby highlighting how exactly this setting differs from the purely logical core.

4.2.3 Daimonic extension

We now consider extending the logical core by adding daimonic commands. Let the accompanying collection of typing and reduction rules be T-Rules$^{\mathfrak{D}}$ and R-Rules$^{\mathfrak{D}}$[13], respectively. The added typing rules are only for typing commands and these commands are all daimonic. Further, we require substitutions to be applicable to daimonic commands in such a way that the command substitution lemma (Lemma A.2 in the appendix), as used in the proof of preservation above, is preserved. We refer to a reduction that happens via a rule in R-Rules$^{\mathfrak{D}}$ by a superscript $\mathfrak{D}$, e.g. $C \triangleright^{\mathfrak{D}} C'$.

For a daimonically extended system, we can prove a modified version of soundness, which says that if one has evidence for a contradiction, one can also always obtain a daimonic command that already serves as evidence for a contradiction. This makes it clear that there is no unsoundness introduced into the system beyond that introduced by the daimons.

Theorem 4.2 (Modified soundness)**.** *If there is a command C with $\vdash C \overset{cmd}{:} \#$, there is a daimonic command D such that $\vdash D \overset{cmd}{:} \#$.*

[13]We assume the overall set of reduction rules to be deterministic unless specified otherwise.

This theorem actually follows from another which says that D can be found by reduction:

Theorem 4.3 (Evidence shrinking to daimon)**.** *For all commands C with $\vdash C \overset{cmd}{:} \#$, there is a daimonic command D such that $C \rhd^* D$ and $\vdash D \overset{cmd}{:} \#$.*

We can compare this to the situation in the logical core system, where we shrinked away the evidence for the contradiction by using the reduction steps. In a daimonic extension, the evidence does not shrink away completely, but the only thing remaining is daimonic.

To see why the daimonic shrinking theorem holds, consider the following: We can use the reduction of non-daimonic commands to reduce C (if it is not already a daimonic command, in which case we are done).[14] And as in the shrinking for the logical core, if we do not encounter a daimonic command, we could make it structurally smaller than any size, including 1, which is impossible, hence we *must* encounter a daimonic command at some point in the reduction sequence. All that remains now is to show that each reduction step preserves typing, and we have proven the theorem.

Lemma 4.3 (Preservation (daimonic extension))**.** *For all commands C, C', if C is non-daimonic and $\vdash C \overset{cmd}{:} \#$ and $C \rhd C'$, then $\vdash C' \overset{cmd}{:} \#$.*

Proof. Reuse the proof of the preservation lemma for the logical core, which, if C is non-daimonic, does not depend on the absence of daimonic extension (especially it does not depend on C' being non-daimonic). □

It may be desirable that even daimonic commands do not destroy well-typedness when reduced, in which case we call the daimonic extension *good-natured*.

Definition 4.1. *The daimonic extension is* good-natured *iff for all daimonic commands D and commands C, when $\vdash D \overset{cmd}{:} \#$ and $D \rhd^{\mathfrak{D}} C$, it follows that $\vdash C \overset{cmd}{:} \#$.*

Note that we do not *require* daimonic commands to reduce at all, irrespective of whether the extension is good-natured or not. Overall we can conclude that while allowing to derive what corresponds to evidence for a contradiction, a good-natured daimonic extension does not break the practically relevant properties of progress and preservation (and even if the extension is not good-natured, preservation is preserved for non-daimonic commands).

An example of a good-natured daimonic extension is the simple I/O daimon from Section 4.1.2. Another such example, that we will make use of in the next chapter, is a daimonic extension for mutable state. To model mutable state, we consider the reduction of pairs of a command C and a current storage state $\mathcal{S}$, written using a superscript as $C^{\mathcal{S}}$. We will omit the superscript whenever $\mathcal{S} = \emptyset$. For this generalized form of reduction we give rules for the three daimonic commands for mutable state, **NewBox**, **OpenBox**, and **SetBox**, as shown in Fig. 4.11. For each the relevant type T is annotated as a subscript. **NewBox** picks a fresh storage cell address α and writes the given value $a : T$ to that cell, passing *addr* to the given continuation. Addresses that contain a value of type T have the primitive positive type $\texttt{Ref}\langle T \rangle$. **OpenBox** looks up the given address in the current store and passes the resulting value to the given continuation. **SetBox** updates the store at the given address with

[14] Same as in the logical core, *progress* is straightforward.

$$
\begin{array}{lll}
\mathbf{NewBox}_T(a : T, k : \overline{\texttt{Ref}}\langle T\rangle)^{\mathcal{S}} & \triangleright & (\alpha \gg k)^{\mathcal{S}\cup\{\alpha\mapsto a\}} \ (\alpha : \texttt{Ref}\langle T\rangle \textit{ fresh}) \\
\mathbf{OpenBox}_T(\alpha : \texttt{Ref}\langle T\rangle, k : \overline{T})^{\mathcal{S}} & \triangleright & (\mathcal{S}(\alpha) \gg k)^{\mathcal{S}} \\
\mathbf{SetBox}_T(\alpha : \texttt{Ref}\langle T\rangle, a : T, C)^{\mathcal{S}} & \triangleright & C^{\mathcal{S}[\alpha\mapsto a]} \\
(v \gg k)^{\mathcal{S}} & \triangleright & C^{\mathcal{S}}
\end{array}
$$

(C is the result of ordinary one-step reduction of $v \gg k$)

FIGURE 4.11: Mutable state daimon reduction rules.

the given value and reduces to the given command in that new storage state. Logical commands do not touch the store when one-step reducing.

This description already suggests which types the arguments of the daimonic commands are allowed to have; these are also annotated in Fig. 4.11. It easy to see that, assuming these types, the command part of each reduction result is well-typed again; that is, as claimed, the mutable state daimon is good-natured, thus progress and preservation hold when extending $\mathcal{PF}$ with it.

There is one (easily resolved) caveat: We do not actually know that the address passed to **OpenBox** or **SetBox** exists in the current store and that it has the correct type. However, assuming that the address can only be obtained from a use of **NewBox**, these properties are automatically guaranteed. We thus impose the restriction that it is not possible to access address values directly (i.e. other than via **NewBox**). Reduction *does* lead to concrete address values being arguments of **OpenBox** or **SetBox**, but these are not accessible to the programmer.

4.2.4 Soundness of the practical version

We can now prove preservation[15] for the practical version by translating a program into a logical core program with a certain daimonic extension for unrestricted recursion. It will be instructive to see how exactly the practical version presentation is daimonic, i.e. how its unrestricted recursion is *extra-logical*.

We translate a cell entry C (a command) to a cell entry in the (daimonically extended) logical core program, as follows: We collect all names F of functions called that are specified in a row or matrix above or equal to the current row. If F is empty, we do not need to change C. Otherwise, we replace each such occurrence of an $f \in F$ by a new kind of variable ϕ_f; call the resulting command C_0. Finally, the cell entry in the logical core program is:[16]

$$\mathbf{Rec}\,\overrightarrow{\phi_f}^{f\in F}.C_0$$

For such a daimonic command we add the following reduction rule:

$$\mathbf{Rec}\,\overrightarrow{\phi_f}^{f\in F}.C \quad \triangleright^{\mathfrak{D}} \quad C\,\overrightarrow{[\phi_f \mapsto f]}^{f\in F}$$

And we add the following typing rule:

$$\frac{\Gamma \vdash_{\Sigma\cup\overrightarrow{f}^f} C\,\overrightarrow{[\phi_f \mapsto f]}^{f} \overset{\text{cmd}}{:} \#}{\Gamma \vdash_{\Sigma} \mathbf{Rec}\,\overrightarrow{\phi_f}^{f}.C \overset{\text{cmd}}{:} \#}\ \text{T-CMD}$$

On its own, this daimonic extension is not good-natured, since there is no guarantee that all function names f actually appear in the signature of the program. However, for the image of the translation from practical version programs to logical

[15] *Progress* is again easy, similarly to the logical core.

[16] For the proofs of the previous sections to be applicable, we need a way to deal with inlining function definitions in this daimonic command; see the remark below on how to do that.

$$
\begin{array}{lll}
\mathcal{M}^{\text{cells}}_{\texttt{Nat}} & := & \{ \ \texttt{add}(y, \bar{y}) \mapsto \{ \\
& & \quad \texttt{zero} \ \mapsto \ y \gg \bar{y}, \\
& & \quad \texttt{succ}(x) \ \mapsto \ \mathbf{Rec}\,\phi_{\texttt{add}}.\ x \gg \phi_{\texttt{add}}(y, \texttt{succCns}(\bar{y})) \ \}, \\
& & \texttt{succCns}(\bar{y}) \mapsto \{ \\
& & \quad \texttt{zero} \ \mapsto \ \texttt{succ}(\texttt{zero}) \gg \bar{y}, \\
& & \quad \texttt{succ}(x) \ \mapsto \ \texttt{succ}(\texttt{succ}(x)) \gg \bar{y} \ \}, \\
& & \texttt{doneCns} \mapsto \{ \\
& & \quad \texttt{zero} \ \mapsto \ \mathbf{Result}(\texttt{zero}), \\
& & \quad \texttt{succ}(x) \ \mapsto \ \mathbf{Result}(\texttt{succ}(x)) \ \} \\
& & \} \\
\Sigma^{\text{elem}}(\text{prd}) & := & \{\texttt{zero}\{\}, \texttt{succ}\{x : \texttt{Nat}\}\} \\
\Sigma^{\text{elem}}(\text{cns}) & := & \{\texttt{add}\{y : \texttt{Nat}, \bar{y} : \overline{\texttt{Nat}}\}, \texttt{succCns}\{\bar{y} : \overline{\texttt{Nat}}\}, \texttt{doneCns}\{\}\}
\end{array}
$$

(note that $\mathcal{M}^{cells}_{Nat}$ is a positive polarity matrix)

$$
\mathcal{P}_{\text{ex}} := \{\texttt{Nat} \mapsto (\mathcal{M}^{\text{cells}}_{\texttt{Nat}}, \Sigma^{\text{elem}}(\text{prd}), \Sigma^{\text{elem}}(\text{cns}))\}
$$

Reduction steps:

$$
\begin{array}{ll}
& \texttt{succ}(\texttt{succ}(\texttt{zero})) \gg \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{doneCns}) \\
\triangleright & \mathbf{Rec}\,\phi_{\texttt{add}}.\ \texttt{succ}(\texttt{zero}) \gg \phi_{\texttt{add}}(\texttt{succ}(\texttt{zero}), \texttt{succCns}(\texttt{doneCns})) \\
\triangleright^{\mathfrak{D}} & \texttt{succ}(\texttt{zero}) \gg \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{succCns}(\texttt{doneCns})) \\
\triangleright & \mathbf{Rec}\,\phi_{\texttt{add}}.\ \texttt{zero} \gg \phi_{\texttt{add}}(\texttt{succ}(\texttt{zero}), \texttt{succCns}(\texttt{succCns}(\texttt{doneCns}))) \\
\triangleright^{\mathfrak{D}} & \texttt{zero} \gg \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{succCns}(\texttt{succCns}(\texttt{doneCns}))) \\
\triangleright & \texttt{succ}(\texttt{zero}) \gg \texttt{succCns}(\texttt{succCns}(\texttt{doneCns})) \\
\triangleright & \texttt{succ}(\texttt{succ}(\texttt{zero})) \gg \texttt{succCns}(\texttt{doneCns}) \\
\triangleright & \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero}))) \gg \texttt{doneCns} \\
\triangleright & \mathbf{Result}(\texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zero}))))
\end{array}
$$

FIGURE 4.12: Addition example, translated to the logical core.

core programs, this is the case, allowing us to conclude that progress and preservation hold (in the logical core) relative to all programs in the image of the translation. As an example of this translation from programs to programs, consider the result of the translation of the addition program (Figure 4.9), shown in Figure 4.12. Compared to the practical version presentation, two daimonic steps ($\triangleright^{\mathfrak{D}}$) are now interspersed, exactly in the spots where a recursive addition call occurs.

Of course, we would like the practical version reduction $\triangleright^0$ to *always* be in close correspondence with the reduction for the translated commands. We will now demonstrate this to be the case and immediately have preservation for the practical version of $\mathcal{PF}$ from this fact. The translation only changes commands in matrix cells; the closed commands where we consider their reduction relative to a program of the practical version stay the same. Now, such a command C reduces to a command instantiated from a cell entry, i.e. $C \triangleright^0 C'$ with $C'\sigma$ being a cell entry. We also have $C \triangleright C''\sigma$ in the logical core, where C'' is the translation of C'. If $C' = C''$ (because it does not contain a recursive reference), the reduction steps in the practical version and in the logical core agree. Otherwise, it is, for some C_0:

$$
C''\sigma = (\mathbf{Rec}\,\overrightarrow{\phi_f}^{f \in F}.C_0)\sigma \ \triangleright^{\mathfrak{D}} \ (C_0\,\overrightarrow{[\phi_f \mapsto f]}^{f \in F})\sigma = C'\sigma.
$$

Thus, overall we have $C \triangleright C''\sigma \triangleright^{\mathfrak{D}} C'\sigma$, so the reduction in the logical core takes at most one extra step, and this step is guaranteed to be daimonic. Moreover, since typechecking of commands (individually) does not differ between the practical version and the logical core, from this reduction correspondence we immediately have preservation for the practical version as well.

Remark: Inlining for Rec. We have not yet considered how to inline function definitions in a daimonic **Rec** command, but we used this inlined presentation in the proofs of the previous sections, so we need to make sure this inlining is possible. We

take the recursion command

$$\mathbf{Rec}\,\overrightarrow{\phi_f}^{f\in F}.C,$$

as created above, and turn it into

$$\mathbf{Rec}\,\overrightarrow{\phi_f : t_f}^{f\in F}.C,$$

where each t_f is the inline form of f, i.e. a μ- (or similarly a $\tilde{\mu}$-)abstraction

$$\mu\{\,\overrightarrow{\mathsf{X}_i\Xi \Rightarrow C_i}^i\},$$

where the variables $\overrightarrow{x_j}^j$ annotated in the matrix row label for f are left open. The C_i themselves undergo inlining, but only for functions not in F; the other function names f simply become their respective ϕ_f. Finally, we accordingly adapt the reduction rule for **Rec**, replacing recursive function calls by such inline forms of the functions, with the call arguments substituted for the x_j:

$$\mathbf{Rec}\,\overrightarrow{\phi_f : t_f}^{f\in F}.C \quad \triangleright^{\mathfrak{D}} \quad C\,[\overrightarrow{\phi_f(\overrightarrow{t_{f,j}}^j) \mapsto t_f\,[\overrightarrow{x_j \mapsto t_{f,j}}^j]}^{f\in F}]$$

As intended, a reduction step now does not lead to global references to functions anymore.

4.3 Pragmatic Extensions

In this section we consider two rather small extensions of the practical version, which will however be quite handy when practically working with the system, as evidenced by some initial examples. These will also serve as a first demonstration of how $\mathcal{PF}$ is indeed able to serve as a foundation for automated tooling. From now on, we will exclusively use the practical version as our base system, unless otherwise noted.

4.3.1 Local annotations

As a first little aid, we can allow the X identifiers appearing in the program to be annotated as *local*. Now, consider some concrete linearization. If the respective identifier becomes a function name (and not an xtor name), then this annotation indicates that it should be inlined using a μ- or $\tilde{\mu}$-abstraction (as used in the logical core proofs, but without the explicit substitutions). We only allow local annotations in the simple case that the relevant function is only called once in the entire program. We also require calls to local functions to have only variables as their arguments, which will enable us to recover the matrix form from the linearized form, as we will see below.

As a first example, consider again the addition program from the previous section. We do not really want an extra top-level definition for `succCns`, since its purpose is limited to the place where it appears in the matrix cell for `add`. For simplicity we will place a local annotation by prefixing the identifier with an underscore _ (assuming identifiers never start with that symbol). When we have `_succCns`, linearizing `Nat` can produce one of the two programs shown in Figure 4.13. The program on the top, where `Nat` is positively polarized, has a $\tilde{\mu}$ abstraction for the successor consumer. Note that in order to be able to unambiguously recover the name information for the matrix from the linearized form, the name `_succCns` is also annotated to the $\tilde{\mu}$. The program on the bottom, where `Nat` is negatively polarized, has `_succCns` as a destructor. We can read this program back into matrix form, together with the local annotation. From there we can decide to linearize the other way, and recognize `_succCns` to be inlined by means of the annotation. Put in another way, in the

```
data₊ Nat { zero | succ(Nat) }

function₊ add(y, ȳ) := {
    zero        ⇒ y ≫ ȳ
    succ(x)     ⇒
      x ≫ add(y, μ̃_succCns{zero ⇒ succ(zero) ≫ ȳ | succ(x) ⇒ succ(succ(x)) ≫ ȳ})
}
```

```
data₋ Nat { add(Nat, N̄at) | _succCns(N̄at) }

function₋ zero := {
    add(y, ȳ)       ⇒ y ≫ ȳ
    _succCns(ȳ)     ⇒ succ(zero) ≫ ȳ
}

function₋ succ(x) := {
    add(y, ȳ)       ⇒ x ≫ add(y, _succCns(ȳ))
    _succCns(ȳ)     ⇒ succ(succ(x)) ≫ ȳ
}
```

FIGURE 4.13: Addition example with local annotation.

negatively polarized program there is no inlining happening in this example, but the local annotation on the destructor makes it clear what to inline in the other linearization, and is necessary so we do not lose the information. That is, with these annotations, there is no information loss when switching between the linearization, thus these switches are full inverses of each other (i.e. going back and forth we will always come back to the same program we started with).

Going from a linearized program with local abstractions back to the matrix form also requires to collect the variables, and their types that appear free in each abstraction's body. As indicated above, here we assume that the arguments of the call to the function annotated as local are variables only, allowing to uniquely recover the matrix form from the linearized form. In the addition example, there is the free variable $\bar{y}$ within the body of the $\tilde{\mu}$-abstraction for `_succCns`. In the other linearization, this free variable corresponds to the argument of the `_succCns` destructor. In the slightly more involved example of Figure 4.14, which is (close to) Reynold's *meta-circular interpreter* in the linearization for the negatively polarized `Val` (top of the figure), this automatic process gives us an interpreter with *closures* in the opposite linearization (bottom). More precisely, the local abstraction in the `Abs` case of `eval` that encodes a first-class function is turned into the `_closure` constructor. That is, the transposition combined with the proper treatment of local abstractions sort of "invents" closures for us.

This is another example in the vein of uses of defunctionalization to interderive semantic artifacts, as elaborated on in Section 2.2.4; this can be leveraged for automated tooling similarly to the tools build upon the system of Binder et al. (2019) (see Section 2.4.1). Starting from the high-level code with first-class function abstractions

that is *meta-circular* in the sense that it uses first-class functions to implement first-class functions, we arrive at code closer to code runnable on a machine (with the code itself resembling an abstract machine). In particular, there is less "cleverness" required to write the high-level code, as it almost only amounts to a straightforward realization of the rather self-referential description: "An abstraction evaluates to something that can be applied to a value." Practically the only thing one has to take care of is variable binding, which in this example was dealt with by using environments (in a De Bruijn-style).[17] And with program transposition ready as a tool, there is no need to be clever if one wants to obtain a "real" implementation of first-class function, since transposition gives that for free.

Note that, as in the previous example, we gave the abstraction corresponding to the closure constructor a name, `_closure`. In an interactive programming environment, it should also be possible to create such names automatically, perhaps with some help from the user and/or taking contextual information into consideration. Finally, note that in the example there is another constructor corresponding to a local abstraction, named `_aux`. This one is a bit annoying, as it is only an artifact of how we encoded sequential evaluation in our system. This artifact can be avoided by a more careful encoding, but it is also related to a somewhat intricate asymmetry between data and codata; we will come back to this topic in the next chapter when we consider macros for surface languages.

4.3.2 Whole-value patterns

In each of the examples of the previous section, we saw an abstraction which does not actually inspect the values it is pattern matching on: Both cases of the successor consumer are identical relative to the whole pattern for the respective case; the `_aux` producer has only one case, but this likewise only inserted the whole pattern into the right-hand side, and not the pattern variables individually. Our system was deliberately designed to facilitate program transposition, and thus there is only one case per constructor, i.e. all matches are shallow. However, allowing a *whole-value pattern* (like the as-pattern from Haskell, but only for the entire pattern), with which one can abbreviate the entire constructor call, does not break the ability to transpose. Also, adding this feature allows us to write a simple macro that just repeats the same command for all cases, so that we have the ability to (on the macro level) write μ- and $\tilde{\mu}$-abstractions that simply map from a variable to a command where this variable appears freely (like in a λ-abstraction, for instance).

We use a new symbol W for a whole-value pattern reference in a matrix cell. We refer to the respective pattern from the row label as W_r and to that from the column label as W_c. We straightforwardly enhance the reduction rules to first replace the W with the respective pattern, i.e. a $\mathsf{X}\sigma^{\mathrm{id}}_{\Xi}$, where the $\sigma^{\mathrm{id}}_{\Xi}$ simply map the variables from the respective label's argument context Ξ back to themselves, before instantiating the variables as usual. Once the program is linearized, if the respective X refers to a function, we will write that pattern out in place of the W to make it clear that this is a recursive reference. If it is an xtor pattern, there is thus no need for a r or c subscript, so we simply write W; we also suggestively write $W@\mathsf{X}\Xi \Rightarrow C$, like in a Haskell-style as-pattern.[18] For example, the `_aux` abstraction from the meta-circular interpreter example above becomes:

$$\mu_{\texttt{_aux}}\{W@\texttt{apply}(v,\bar{v}) \Rightarrow e_2 \gg \texttt{eval}(l,W)\}$$

[17]We have omitted the straightforward definition of `lookup` (used in the `var` case of `eval`).

[18]We may sometimes use a different symbol than W since we can recognize it as a whole-value reference via the as-pattern, i.e. the relevant symbol is what stands before the @.

```
data− Val { apply(Val, V̅a̅l̅) }
data+ ValList { nil | cons(Val, ValList) }
data+ Exp { var(Nat) | app(Exp, Exp) | abs(Exp) }
data+ Nat { zero | succ(Nat) }

function+ eval(l, v̄) := {
    var(n)          ⇒ n ≫ lookup(l, v̄)
    app(e1, e2)     ⇒ e1 ≫ eval(l, apply(μ_aux{apply(v, v̄) ⇒ e2 ≫ eval(l, apply(v, v̄))}, v̄))
    abs(e)          ⇒ μ_closure{apply(v, v̄) ⇒ e ≫ eval(cons(v, l), v̄)} ≫ v̄
}
```

```
data+ Val { _closure(Exp, ValList) | _aux(Exp, ValList) }
data+ ValList { nil | cons(Val, ValList) }
data+ Exp { var(Nat) | app(Exp, Exp) | abs(Exp) }
data+ Nat { zero | succ(Nat) }

function+ apply(v, v̄) := {
    _closure(e, l)  ⇒ e ≫ eval(cons(v, l), v̄)
    _aux(e, l)      ⇒ e ≫ eval(l, apply(v, v̄))
}

function+ eval(l, v̄) := {
    var(n)          ⇒ n ≫ lookup(l, v̄)
    app(e1, e2)     ⇒ e1 ≫ eval(l, apply(_aux(e2, l), v̄))
    abs(e)          ⇒ _closure(e, l) ≫ v̄
}
```

FIGURE 4.14: Meta-circular interpreter (top) and interpreter with closures (bottom).

For readability, we will also sometimes leave out the variables in the pattern if they are unused (on the surface), like v or $\bar{v}$ in the `_aux` abstraction, replacing them by underscores (as known from, e.g., Haskell or Scala).

Now, for abstractions where all cases are identical when utilizing such a W, as is the case for the `succCns` abstraction, as announced we use a simple macro that turns ($\tilde{\mu}$ analogously) $\mu\{W \Rightarrow C\}$, an abstraction for some type T, to $\mu\{\overrightarrow{W@\mathsf{X}_i\Xi_i \Rightarrow C}^i\}$, where $\overrightarrow{\mathsf{X}_i\Xi_i}^i$ are the xtor signatures of T; we may use a different symbol than W (see footnote above). The `succCns` abstraction we can now write (on the macro level) as:

$$\tilde{\mu}_{\texttt{_succCns}}\{n \Rightarrow \texttt{succ}(n) \gg \bar{y}\}$$

Overall, with a few small extensions we have gotten rid of some practically important awkwardnesses exhibited by our parsimonious, transposition-friendly core system. Next we equip it with parametric polymorphism, and then are ready to put it to use by recovering existing languages in the next chapter.

4.4 Polymorphic Extension

This section presents the extension of $\mathcal{PF}$ with parametric polymorphism. Just like with the polymorphic natural deduction-based system (Ostermann and Jabs, 2018) summarized in Section 2.4.2, this requires Generalized Algebraic Data Types and their dual, which we called Generalized Algebraic Codata Types in the natural deduction setting. Actually, the modification applied to the syntax of types is exactly the same as for natural deduction, and the key change to the type system, which involves computing the most general unifier (Hindley, 1969; Milner, 1978; Robinson, 1965) of producer and consumer type parameters, also carries over from there. Thus, for the intuition underlying the polymorphic extension and unification, and more examples, refer to Section 2.4.2.

In this section, we first consider a system which extends the practical version of $\mathcal{PF}$, with built-in recursion. As with the non-polymorphic system, we then consider the logical core, from which recursion and hence the practical version extension can be recovered by a daimon. Daimonic extensions in general and the recursion daimon, as well as showing how consistency of the practical version follows from that of the logical core via the daimon, are all straightforward adaptations of what is presented above, so we will not consider them in detail again. Instead, we focus on proving the polymorphic logical core consistent, for which it turns out to be sufficient to adapt the *preservation* lemma to take unification into account. Compared to the natural deduction based GA(Co)DT language of Ostermann and Jabs (2018) and its Coq proof, the proof presented here is streamlined quite a bit thanks to the fully symmetric setting.[19]

4.4.1 Practical version

Fig. 4.15 shows the syntax for the polymorphic system extending the practical version of $\mathcal{PF}$ (in Fig. 4.15a and Fig. 4.15b; Fig. 4.15c is for the logical core considered below). Types are now either data types with an instantiation of type parameters $\langle\overrightarrow{T}\rangle$, or variables A. X-calls now also have type parameters instantiated, with the

[19] The non-polymorphic data and codata language can be embedded into the non-polymorphic version of $\mathcal{PF}$ (see Section 5.2), and the same goes for the GA(Co)DT language and the polymorphic version of $\mathcal{PF}$; in the polymorphic embedding type parameters simply carry over, otherwise the embedding is identical to that for the non-polymorphic version (Section 5.2.5 presents an example embedding of a GA(Co)DT program into $\mathcal{PF}$).

$D \in$ *type names*, $\mathsf{X} \in$ *label names*, $x \in$ *variable names* *(Names)*
$A \in$ *type variable names*
(var. naming convention: plain (e.g. x, y) for prd, overbar (e.g. $\bar{x}$, $\bar{y}$) for cns)

$$T ::= D\langle \overrightarrow{T} \rangle \mid A \quad \textit{(Types)}$$
$$\mathcal{J} ::= \mathsf{prd} \mid \mathsf{cns} \quad \textit{(Judgments)}$$
$$\Xi ::= \overrightarrow{x \overset{\mathcal{J}}{:} T} \quad \textit{(Argument context)}$$

(shorthands: $x : T$ for $x \overset{prd}{:} T$, $\bar{x} : \overline{T}$ for $\bar{x} \overset{cns}{:} T$)

$$\Sigma^{\text{elem}} ::= \{\mathsf{prd} \mapsto \overrightarrow{\mathsf{X}\Xi}, \mathsf{cns} \mapsto \overrightarrow{\mathsf{X}\Xi}\} \quad \textit{(Signature element)}$$
$$\Sigma ::= \overrightarrow{T \mapsto \Sigma^{\text{elem}}} \quad \textit{(Signature)}$$
$$\Sigma^{\text{elem}}_{\mathrm{T}} ::= \overrightarrow{\mathsf{X} \mapsto \langle \overrightarrow{T} \rangle}, \overrightarrow{\mathsf{X} \mapsto \langle \overrightarrow{T} \rangle} \quad \textit{(Type parameters element)}$$
$$\Sigma_{\mathrm{T}} ::= \overrightarrow{T \mapsto \Sigma^{\text{elem}}_{\mathrm{T}}} \quad \textit{(Type parameters)}$$

$$\sigma ::= \overrightarrow{x \mapsto t} \quad \textit{(Substitutions)}$$
$$p, c, t ::= \mathsf{X}\langle \overrightarrow{T} \rangle \sigma \mid x \quad \textit{(Producers / Consumers)}$$
$$C ::= p \gg c \mid \mathfrak{D} \quad \textit{(Commands)}$$

(A) Terms (common).

$$\mathcal{M}^{\text{cells}} ::= \overrightarrow{(\mathsf{X}\langle \overrightarrow{A} \rangle \Xi, \mathsf{X}\langle \overrightarrow{A} \rangle \Xi) \mapsto C} \quad \textit{(Program matrix, unlabeled)}$$
$$\mathcal{M} ::= (\mathcal{M}^{\text{cells}}, \overrightarrow{\mathsf{X}\langle \overrightarrow{A} \rangle \Xi}, \overrightarrow{\mathsf{X}\langle \overrightarrow{A} \rangle \Xi}, \Sigma^{\text{elem}}_{\mathrm{T}}) \quad \textit{(Program matrix, labeled)}$$
$$\mathcal{P} ::= \overrightarrow{D \mapsto \mathcal{M}} \quad \textit{(Program)}$$

(B) Programs (practical version).

$$\mathcal{M}^{\text{cells}} ::= \overrightarrow{\mathsf{X}\langle \overrightarrow{A} \rangle \Xi \mapsto \overrightarrow{\mathsf{X}\langle \overrightarrow{A} \rangle \Xi \mapsto C}} \quad \textit{(Program matrix, unlab.)}$$

(If the outer X_i are for producers, $\mathcal{M}^{cells}$ is of negative polarity.
If the outer X_i are for consumers, $\mathcal{M}^{cells}$ is of positive polarity.)

$$\mathcal{M} ::= (\mathcal{M}^{\text{cells}}, \overrightarrow{\mathsf{X}\langle \overrightarrow{A} \rangle \Xi}, \overrightarrow{\mathsf{X}\langle \overrightarrow{A} \rangle \Xi}, \Sigma^{\text{elem}}_{\mathrm{T}}) \quad \textit{(Program matrix, labeled)}$$
$$\mathcal{P} ::= \overrightarrow{T \mapsto \mathcal{M}} \quad \textit{(Program)}$$

(C) Programs (logical core).

FIGURE 4.15: Syntax for the polymorphic system. Differences to the syntax of the non-polymorphic system (Fig. 4.6) are highlighted.

signatures of the X (in the program matrices) now being of the form $\mathsf{X}\langle \overrightarrow{A} \rangle \Xi$, where $\overrightarrow{A}$ are the type variables available in Ξ and in the bodies (i.e., the cells of the matrix). For each X, it must also be specified how the type parameters of their respective data type should be instantiated; the type variables $\overrightarrow{A}$ bound in the signatures are available here.

Fig. 4.16 presents the typing rules, extending Fig. 4.7 (for the non-polymorphic system). The differences to the non-polymorphic system are (1) that we carry around more static signature information, namely the type parameter instantiations just mentioned, and properly make use of these in T-X, and (2) the unification of the type parameter instantiations of the relevant data type for X employed in M-OK. Note that, just as with the system of Section 2.4.2, when unification of this instantiation for some X_p and X_c fails, the matrix cell need not actually be filled with any command since when considering only well-formed programs this code can never

$$\frac{\Gamma \vdash_{\Sigma;\Sigma_T} \sigma : \Xi[\overrightarrow{A \mapsto T'}] \quad \mathsf{X}\langle\overrightarrow{A}\rangle\Xi \in \Sigma(D)(\mathcal{J}) \quad \langle\overrightarrow{T}\rangle = \Sigma_T(D)(\mathsf{X})}{\Gamma \vdash_{\Sigma;\Sigma_T} \mathsf{X}\sigma \overset{\mathcal{J}}{:} T}\ \text{T-X} \qquad \frac{x \overset{\mathcal{J}}{:} T \in \Gamma}{\Gamma \vdash_{\Sigma;\Sigma_T} x \overset{\mathcal{J}}{:} T}\ \text{T-VAR}$$

$$\frac{\Gamma \vdash_{\Sigma;\Sigma_T} p \overset{\text{prd}}{:} T \quad \Gamma \vdash_{\Sigma;\Sigma_T} c \overset{\text{cns}}{:} T}{\Gamma \vdash_{\Sigma;\Sigma_T} p \gg c \overset{\text{cmd}}{:} \#}\ \text{T-CMD} \qquad \frac{\overrightarrow{\Gamma \vdash_{\Sigma;\Sigma_T} t \overset{\mathcal{J}}{:} T}}{\Gamma \vdash_{\Sigma;\Sigma_T} \overrightarrow{x \mapsto t} : \overrightarrow{x \overset{\mathcal{J}}{:} T}}\ \text{T-SUBST}$$

(A) Term typing (common). For daimonic commands $\mathfrak{D}$ custom rules are added.

$$\frac{\mathcal{M} = (\mathcal{M}^{\text{cells}}, \overrightarrow{\mathsf{X}_p\Xi_p}, \overrightarrow{\mathsf{X}_c\Xi_c}, \Sigma_T^{\text{elem}}) \quad L = \overrightarrow{\mathsf{X}_p\Xi_p} \times \overrightarrow{\mathsf{X}_c\Xi_c} \quad \text{dom}(\text{uncurry}(\mathcal{M}^{\text{cells}})) \in \{L, L^\top\}}{\vdash \mathcal{M}\ \text{SIG}(\{\text{prd} \mapsto \overrightarrow{\mathsf{X}_p\Xi_p}, \text{cns} \mapsto \overrightarrow{\mathsf{X}_c\Xi_c}\}; \Sigma_T^{\text{elem}})}\ \text{M-SIG}$$

(B) Signature judgment. Gray-text parts only apply to logical core.

$$\frac{\Sigma_T^{\text{elem}} = \overrightarrow{\mathsf{X}_{p,i} \mapsto \langle\overrightarrow{T_{p,i}}\rangle}^i, \overrightarrow{\mathsf{X}_{c,j} \mapsto \langle\overrightarrow{T_{c,j}}\rangle}^j \quad \overrightarrow{\tau_{i,j} = \text{mgu}(\overrightarrow{T_{p,i} = T_{c,j}}) \implies \Xi_{p,i}\tau_{i,j}, \Xi_{c,j}\tau_{i,j} \vdash_{\Sigma;\Sigma_T} C_{i,j}\tau_{i,j} \overset{\text{cmd}}{:} \#}^{i,j}}{\vdash_\Sigma (\overrightarrow{(\mathsf{X}_{p,i}\Xi_{p,i}, \mathsf{X}_{c,j}\Xi_{c,j}) \mapsto C_{i,j}}^{i,j}, \ldots, \ldots, \Sigma_T^{\text{elem}})\ \text{OK}}\ \text{M-OK}$$

$$\frac{\overrightarrow{\vdash \mathcal{M}\ \text{SIG}(\Sigma^{\text{elem}}; \Sigma_T^{\text{elem}})} \quad \Sigma; \Sigma_T = \overrightarrow{T \mapsto \Sigma^{\text{elem}}}; \overrightarrow{T \mapsto \Sigma_T^{\text{elem}}} \quad \overrightarrow{\vdash_{\Sigma;\Sigma_T} \mathcal{M}\ \text{OK}}}{\vdash \overrightarrow{T \mapsto \mathcal{M}}\ \text{OK}}\ \text{P-OK}$$

(C) Well-formedness (practical version).

$$\frac{\vdash \mathcal{P}\ \text{OK}}{\vdash \{T \mapsto \emptyset\} \cup \mathcal{P}\ \text{OK}}\ \text{P-MEMPTY-OK}$$

$$\frac{\begin{gathered}\mathcal{P}\ \text{DECOMP.}((\mathsf{X}\Xi \mapsto \overrightarrow{\mathsf{X}'_i\Xi'_i \mapsto C_i}^i, \langle\overrightarrow{T'}\rangle); \overrightarrow{T \mapsto \mathcal{M}}) \\ \overrightarrow{\vdash \mathcal{M}\ \text{SIG}(\Sigma^{\text{elem}}, \Sigma_T^{\text{elem}})} \quad \vdash \overrightarrow{T \mapsto \mathcal{M}}\ \text{OK} \\ \Sigma; \Sigma_T = \overrightarrow{T \mapsto \Sigma^{\text{elem}}}; \overrightarrow{T \mapsto \Sigma_T^{\text{elem}}} \\ \overrightarrow{\tau_i = \text{mgu}(\overrightarrow{T'} = \Sigma_T(T_1)(\check{\mathsf{X}}'_i)) \implies \Xi\tau_i, \Xi'_i\tau_i \vdash_{\Sigma;\Sigma_T} C_i\tau_i \overset{\text{cmd}}{:} \#}^i\end{gathered}}{\vdash \mathcal{P}\ \text{OK}}\ \text{P-OK}$$

(D) Well-formedness (logical core).

FIGURE 4.16: Typing rules for $\mathcal{PF}$ with parametric polymorphism. Type parameter bindings $\langle\overrightarrow{A}\rangle$ are omitted whenever they are not directly used by the respective rule. Differences to the non-polymorphic system (Fig. 4.7) are highlighted (except $\Sigma; \Sigma_T$ in the term judgments). The M-DCP rule for DECOMPOSE is straightforwardly extended to cover Σ_T^{elem}.

be reached. In examples we will leave these cases away or grayed out when in matrix form; technically, in order to keep the rule concise, M-OK allows for an arbitrary command to be in such a place (in the premiss of M-OK, typechecking the command is only necessary when mgu computation is successful), which is fine since it is unreachable anyway.

For computing the mgu, we have the simple algorithm shown in Fig. 4.17, essentially following Robinson (1965).

Lemma 4.4 (mgu computes the most general unifier). *If* $\tau = mgu(\overrightarrow{T = T'})$ *for two lists of types* $\overrightarrow{T}, \overrightarrow{T'}$, *then* τ *is the most general unifier for these lists. That is, for any map*

$$
\begin{array}{lll}
\text{mgu}(\emptyset) & := & \emptyset \\
\text{mgu}(\{A = T\} \cup \mathcal{C}) & := & \text{mgu}(\mathcal{C}[A \mapsto T]) \circ [A \mapsto T] \text{ if } A \notin \text{FV}(T) \\
\text{mgu}(\{T = A\} \cup \mathcal{C}) & := & \text{mgu}(\{A = T\} \cup \mathcal{C}) \\
\text{mgu}(\{D\langle \overrightarrow{T} \rangle = D\langle \overrightarrow{T'} \rangle\} \cup \mathcal{C}) & := & \text{mgu}(\overrightarrow{T = T'} \cup \mathcal{C}) \\
\text{mgu} & & \text{fails otherwise}
\end{array}
$$

FIGURE 4.17: Computing the most general unifier (mgu).

from type variables to types τ' that is a unifier of these two lists (i.e. $\overrightarrow{T}\tau' = \overrightarrow{T'}\tau'$) there exists a map from type variables to types τ'' such that $\tau' = \tau\tau''$.

Proof. The algorithm of mgu is virtually the same as that given by Pierce (2002, Def. 22.4.4), allowing us to reuse his proof (Pierce, 2002, Th. 22.4.5). The only difference is that $D\langle T_1, ..., T_n\rangle$ takes the place of the function type $T_1 \to T_2$ in the substructure case; we thus need to generalize the relevant parts of the proof to take into account the arbitrary number (instead of constant 2) of arguments T_i, but this is straightforward. ☐

Continuing the example from the previous sections, Fig. 4.18 shows the metacircular interpreter, now with a proper user-defined generic function type `Fun`$\langle *, * \rangle$ (top), and its transposition result, the interpreter with closures (bottom). In linearized forms of programs such as these, we indicate the number of type parameters of some data type in its declaration, right next to the name of the type, with a list of asterisks $\langle *, ..., * \rangle$. In constructor or destructor signatures, the respective instantiation of the data type parameters is given behind a colon, e.g.

$$\texttt{apply}\langle A_1, A_2\rangle(A_1, \overline{A}_2) : \texttt{Fun}\langle A_1, A_2\rangle.$$

Note that in the transposition result (the interpreter with closures) the type parameters of `Fun` are only used in a limited fashion, namely being always instantiated with `Val` and `Val`. In such cases the post-processing of the transposition result described below may be useful.

Safe monomorphization. In the example there is only one constructor for `Fun`, and this constructor specifies the type parameters to be `Val`. In this situation and of course also in the case that all type parameter instantiations of all constructors of the considered type agree, as long as they are concrete types (without type variables), the type parameters can be safely removed (but the fact that `Fun` was originally parametric should ideally be recorded to facilitate going back to the other linearization).

Recursive type elimination. After safe monomorphization, it becomes apparent that there is no recursive reference to `Val` within its constructor signatures and that `Val` is now merely a wrapper around `Fun`. This is a case where a type (like `Val`) is only needed to tie a recursive knot for some other type (like for `Fun`), and where this purpose disappears after transposition. We can thus collapse the two types into one. This also allows for auxiliary wrapper functions like `inCns` to be removed. Possibly it is desirable to add a linguistic construct that makes it explicit that some type (like `Val`) has exactly the recursive knot-tying purpose, in order to automate such an elimination step, and, in particular, to ease discovery of auxiliary wrapper functions.

$\textbf{data}_-$ `Fun`$\langle *,* \rangle$ { `apply`$\langle A_1, A_2\rangle(A_1, \overline{A}_2)$: `Fun`$\langle A_1, A_2\rangle$ }
$\textbf{data}_+$ `Val` { `in(Fun`$\langle$`Val, Val`$\rangle$`)` }
$\textbf{data}_+$ `ValList` { `nil` | `cons(Val, ValList)` }
$\textbf{data}_+$ `Exp` { `var(Nat)` | `app(Exp, Exp)` | `abs(Exp)` }
$\textbf{data}_+$ `Nat` { `zero` | `succ(Nat)` }

$\textbf{function}_+$ `inCns`$(\bar{f})$:= {
    `in`(f) $\Rightarrow f \gg \bar{f}$
}

$\textbf{function}_+$ `eval`$(l, \bar{v})$:= {
    `var`(n) $\Rightarrow n \gg$ `lookup`$(l, \bar{v})$
    `app`(e_1, e_2) $\Rightarrow$
      $e_1 \gg$ `eval`$(l,$ `inCns(`
        `apply(in(`μ`_aux`$\{W \Rightarrow e_2 \gg$ `eval`$(l,$ `inCns`$(W))\}), \bar{v})))$
    `abs`(e) $\Rightarrow$
      `in(`μ`_closure`$\{$`apply`$(v, \bar{v}) \Rightarrow e \gg$ `eval(cons`$(v, l), \bar{v})\}) \gg \bar{v}$
}

$\textbf{data}_+$ `Fun`$\langle *,* \rangle$ { `_closure(Exp, ValList)` : `Fun`$\langle$`Val, Val`$\rangle$ | `_aux(Exp, ValList)` : `Fun`$\langle$`Val, Val`$\rangle$ }
$\textbf{data}_+$ `Val` { `in(Fun`$\langle$`Val, Val`$\rangle$`)` }
$\textbf{data}_+$ `ValList` { `nil` | `cons(Val, ValList)` }
$\textbf{data}_+$ `Exp` { `var(Nat)` | `app(Exp, Exp)` | `abs(Exp)` }
$\textbf{data}_+$ `Nat` { `zero` | `succ(Nat)` }

$\textbf{function}_+$ `apply`$(v, \bar{v})$:= {
    `_closure`(e, l) $\Rightarrow e \gg$ `eval(cons`$(v, l), \bar{v})$
    `_aux`(e, l) $\Rightarrow e \gg$ `eval`$(l,$ `inCns(apply`$(v, \bar{v})))$
}

$\textbf{function}_+$ `eval`$(l, \bar{v})$:= {
    `var`(n) $\Rightarrow n \gg$ `lookup`$(l, \bar{v})$
    `app`(e_1, e_2) $\Rightarrow e_1 \gg$ `eval`$(l,$ `inCns(apply(in(_aux`$(e_2, l)), \bar{v})))$
    `abs`(e) $\Rightarrow$ `in(_closure`$(e, l)) \gg \bar{v}$
}

FIGURE 4.18: Meta-circular interpreter with user-defined generic function type (top) and interpreter with closures (bottom).

4.4.2 Logical core

As announced, to prove the system to be sound we now turn to the polymorphic logical core, i.e. the extension of the logical core of $\mathcal{PF}$ with parametric polymorphism, the consistency of which implies that of the system with built-in recursion in the same way as shown in Section 4.2. The syntax of the logical core is shown in Fig. 4.15 (Fig. 4.15a and Fig. 4.15c). The set of changes between the practical polymorphic system with built-in recursion and the polymorphic logical core is the same as that between the respective non-polymorphic systems: in both cases, going from one to the other requires to impose a linearization per matrix, depending on whether the type is specified to be of positive or of negative polarity. The typing rules, shown in Fig. 4.16d, are thus enhanced with polymorphism starting from the non-polymorphic logical core rules (in Fig. 4.7d, which rule out recursion via a decomposition into "top row" and the rest of the program) in the same way as the typing rules for the practical version are enhanced starting from the non-polymorphic practical version.

Proving the logical core to be sound means proving the following theorem:

Theorem 4.4 (Soundness)**.** *There is no command C such that $\vdash C \overset{cmd}{:} \#$.*

To show this, we can easily extend the *evidence shrinking* argument from Section 4.2.2. The only difference is that the reduction now also instantiates the type parameters, i.e. reduction is now defined like this:

$$\begin{array}{lcl} \mu\langle\tau\rangle(\sigma_p)\{\overrightarrow{\mathsf{X}_i\langle\overrightarrow{A}\rangle\Xi \Rightarrow C_i}^i\} \gg \mathsf{X}_i\langle\overrightarrow{T}\rangle\sigma_c & \triangleright & C_i\tau\overrightarrow{[A \mapsto T]}\sigma_p\sigma_c \\ \mathsf{X}_i\langle\overrightarrow{T}\rangle\sigma_p \gg \bar{\mu}\langle\tau\rangle(\sigma_c)\{\overrightarrow{\mathsf{X}_i\langle\overrightarrow{A}\rangle\Xi \Rightarrow C_i}^i\} & \triangleright & C_i\tau\overrightarrow{[A \mapsto T]}\sigma_p\sigma_c \end{array}$$

It is clear that this does not affect the validity of the shrinking argument.[20] So we only need to take care of the preservation lemma:

Lemma 4.5 (Preservation)**.** *For all commands C, C', if $\vdash C \overset{cmd}{:} \#$ and $C \triangleright C'$, then $\vdash C' \overset{cmd}{:} \#$.*

Proof. We approach this proof similarly to that for the non-polymorphic logical core. We now just need to also take into account the instantiation of type variables, which will crucially hinge on program cell entries being typechecked using the most general unifier of the relevant type parameters.

We know C' is a right-hand side C_i inside a μ or $\bar{\mu}$ with some type variable instantiations τ_p, τ_c and some substitution σ applied to it, i.e. $C' = C_i\tau_p\tau_c\sigma$. Since we assume the relevant program to be well-formed, and C_i comes from a matrix cell in the matrix form of the program, $C_i\tau'$ typechecks in some argument context $\Xi\tau'$, where $\tau' = \mathrm{mgu}(\overrightarrow{T_p = T_c})$ and $\overrightarrow{T_p}$, $\overrightarrow{T_c}$ are the instantiations of the type parameters of the data type relevant for the matrix cell.

Since C typechecks and σ consists of two parts which are respectively applied to an xtor call and used as an explicit substitution for a μ- or $\bar{\mu}$-abstraction, both part of C, by inversion (rules T-CMD and T-X) we know that $\vdash \sigma : \Xi'\tau_p\tau_c$ for some argument context Ξ'. This Ξ' is necessarily the same as Ξ, since it is the cell for this xtor and this abstraction from which we got the C_i. By the same inversion we also gather that $\overrightarrow{T_p}\tau_p = \overrightarrow{T_c}\tau_c$, i.e. $\tau_p\tau_c$ is a unifier for $\overrightarrow{T_p = T_c}$.

[20] In particular the lookup in the cases is not affected (i.e. *progress* still holds).

It remains to show that from $\Xi\tau' \vdash C_i\tau' \overset{\text{cmd}}{:} \#$ and $\vdash \sigma : \Xi\tau_p\tau_c$ it follows that $\vdash C_i\tau_p\tau_c\sigma \overset{\text{cmd}}{:} \#$. If we can somehow obtain $\Xi\tau_p\tau_c \vdash C_i\tau_p\tau_c \overset{\text{cmd}}{:} \#$, by the command substitution lemma (polymorphism variant), which we leave for the appendix (Lemma A.4), we have the desired conclusion. Since $\tau_p\tau_c$ is a unifier for $\overrightarrow{T_p = T_c}$, and τ' is the most general unifier for these constraints (see Lemma 4.4 above that states that mgu indeed computes the most general unifier), we know that there is a type instantiation τ'' such that $\tau_p\tau_c = \tau'\tau''$. Finally, from $\Xi\tau' \vdash C_i\tau' \overset{\text{cmd}}{:} \#$ we get $\Xi\tau'\tau'' \vdash C_i\tau'\tau'' \overset{\text{cmd}}{:} \#$ by the type instantiation invariance lemma (Lemma A.5 in the appendix). $\square$

Chapter 5

Recovering Surface Languages in Polarized Flow

The foundational system $\mathcal{PF}$ allows to encode a wide variety of programming languages. This chapter demonstrates this by giving appropriate macro embeddings into $\mathcal{PF}$; thus, all translations from the surface languages that we consider are highly local. More precisely, the constructs of a surface language arise from *syntactic abstractions*, in the sense of Felleisen (1991), over $\mathcal{PF}$; in practical programming, the term *macro* is used for the language feature that facilitates such syntactic abstractions. (Felleisen's work enabled the creation of systematic macro systems for, e.g., Racket, which can be used as a tool for language specification.)

Notably, Felleisen (1991) directly adapts ideas from Kleene et al. (1952) and Troelstra (1973) on conservative extensions of logic with *eliminable* syntactic symbols; thus, arguably, his work also occupies a spot in the "wider" Curry-Howard program. That is to say, this chapter does not leave the realm of studying programming languages by finding corresponding concepts in logic, or generally, the theory of formal systems. Rather, we stay on track, just considering a different aspect that fits well with our "reverse engineering" goal inspired by "reverse mathematics" (Stillwell, 2019): the correspondence of the constructs of a surface language to symbols of a conservative extension of a formal system.

Following Felleisen (1991), we will write a syntactic abstraction for such a construct $\mathfrak{C}$ as follows:

$$A_{\mathfrak{C}}(t_1, ..., t_n) = ...\ t_1\ ...\ t_n\ ...$$

Here $t_1, ..., t_n$ are terms in the surface language which are to be recursively translated to $\mathcal{PF}$ and inserted into an $\mathcal{PF}$ term as specified by the right-hand side of the equation for $A_{\mathfrak{C}}$. For example, for calling a function with some value the following syntactic abstraction could be used:

$$A_{f(\cdot)}(v) = \mu\{\bar{k} \Rightarrow v \gg f(\bar{k})\}$$

(The syntactic abstraction actually used for such calls is a bit different; see the section below.) Assuming some other abstractions for values, together the abstractions give rise to a translation into $\mathcal{PF}$, say ϕ. Translating, e.g., $f(v)$ requires to use ϕ recursively on v and then unfold the syntactic abstraction, i.e. $\phi(f(v)) = \mu\{\bar{k} \Rightarrow \phi(v) \gg f(\bar{k})\}$.

The following sections give embeddings for first-order functions, with call-by-value and call-by-name evaluation strategies, for the data and codata languages considered in the first chapter (Rendel, Trieflinger, and Ostermann, 2015; Binder et al., 2019), as well as embeddings involving daimons (for I/O, mutable state, call-by-need) and simple control effects. The final section walks through embeddings for subsets of real languages (Java and Haskell) and how to combine these with program transposition in $\mathcal{PF}$, illustrated by an example of Lämmel and Rypacek (2008)

(taken from their work on the "expression lemma"; see Section 5.4).

Chapter overview More precisely, this chapter is structured as follows:

- Section 5.1 first generally illustrates the basic principles for embeddings for languages based on intuitionistic natural deduction by means of a simple first-order call-by-value language, then discusses how the embedding has to be adapted for call-by-name.
- Section 5.2 generalizes the first-order languages and its embedding to the full data fragment and dualizes the development to also recover the codata fragment; it also considers how program transposition in these surface languages can be made compatible with transposition in $\mathcal{PF}$.
- Section 5.3 considers embedding languages with control effect constructs which allow to realize non-linear control flow, enhancing the macros from Section 5.1.
- Section 5.4 considers an example given by Lämmel and Rypacek (2008) involving two semantically equivalent programs, using what was established in this chapter to re-explore this development.

Contributions

- Based upon some ideas of Zeilberger (2008b), this chapter demonstrates that $\mathcal{PF}$ is indeed able to serve as a foundational system for various kinds of (idealized) programming languages, via macro embeddings. Note that this is not intended to claim that this is the only or even necessarily the best such system which is capable of that. Rather, the intention is to show that $\mathcal{PF}$, which has been deliberately designed to consolidate the extensibility duality and the De Morgan duality, is still conceptually closely connected with previously existing languages, and not merely an elegant framework. This also supports that, as announced in the introduction, $\mathcal{PF}$ can potentially serve as a systematic foundation for tools that allow to automatically compare and convert between semantically equivalent programs in different languages; this aspect will be fleshed out in Section 5.4 (see below), providing some initial information on how $\mathcal{PF}$ can strategically support the development of such tools as a kind of backbone.
- In particular, program transposition for $\mathcal{PF}$ is compatible with transposition for (co)data surface languages (this just requires some care when specifiying the embeddings, see Section 5.2.3 of why this is in a certain sense a likely unavoidable technical complication).
- The final Section 5.4 brings the framework $\mathcal{PF}$ to bear on programming languages existing in the real world, giving a first impression of how it enables principled, fully formal analysis of the relationship between the linguistic features, semantics, and dualities involved. In particular, utilizing $\mathcal{PF}$ in this way is compatible with the categorical semantics framework of Lämmel and Rypacek (2008) which allows to state and prove the equivalence between algebraic and coalgebraic programs in a formally precise way. Centrally, with $\mathcal{PF}$, one can rather straightforwardly close the formal gap left by Lämmel and Rypacek, 2008 between the categorical framework and the concrete languages, which they only relate in an informal manner.

$$
\begin{array}{lll}
f & \in & \mathbb{F} \qquad \textit{(function names)} \\
t & ::= & t_v \mid t_c \\
t_v & ::= & Z \mid S(t_v) \mid x_S \mid y \\
t_c & ::= & f@(t,t) \mid \mathit{return}(t_v) \\
\\
\mathcal{E} & ::= & f@(\mathcal{E},t) \mid f@(v,\mathcal{E}) \mid [] \\
v & ::= & Z \mid S(v)
\end{array}
$$

$$
\begin{array}{lll}
f@(Z,v) & \blacktriangleright_{\mathrm{v}} & \pi_1(\mathcal{D}(f))[y \mapsto v] \\
f@(S(v'),v) & \blacktriangleright_{\mathrm{v}} & \pi_2(\mathcal{D}(f))[x_S \mapsto v', y \mapsto v] \\
\mathcal{E}[t] & \triangleright_{\mathrm{v}} & \mathcal{E}[t'] \quad \text{if } t \blacktriangleright_{\mathrm{v}} t' \\
\mathcal{E}[\mathit{return}(t_v)] & \triangleright_{\mathrm{v}} & \mathcal{E}[t_v] \quad \text{if } \mathcal{E} \neq []
\end{array}
$$

FIGURE 5.1: Syntax, contraction and reduction rules of a simple first-order CBV language.

$$
\begin{array}{l}
\mathtt{add} \mapsto (\mathit{return}(y), \mathtt{suc}@(\mathtt{add}@(x_S, y))) \\
\mathtt{suc} \mapsto (\mathit{return}(S(Z)), \mathit{return}(S(S(x_S))))
\end{array}
$$

FIGURE 5.2: Addition function example.

5.1 First-Order Functions

We start our exploration with a simple declarative language with first-order functions. This is used to illustrate the basic principles for recovering all languages based on intuitionistic natural deduction. We will especially see how macro embeddings for various evaluation strategies look like.

5.1.1 Embedding call-by-value

We first show how to macro-translate a first-order language with a call-by-value semantics. The language consists of Peano natural numbers, with their usual zero and successor constructors, as well as first-order functions from pairs of natural numbers to natural numbers with shallow matching on the first argument (zero or successor) integrated. The following definitions globally assume the presence of function definitions, which are formally a map $\mathcal{D}$ from the set of function names $\mathbb{F}$ to pairs of computation terms (see below); for each pair, the first element is the body of the function for the zero case, which is projected out by π_1, and the second element is that for the successor case, projected by π_2. Fig. 5.1 shows the syntax, contraction and reduction rules; this presentation takes the well-known approach with evaluation contexts (Felleisen and Hieb, 1992). The variable x_S is used to access the predecessor of the first argument in the successor case (cf. the second contraction rule). Fig. 5.2 shows the addition function as an example (see the remark on *return* below).[1] The type system is trivial, as all terms have type $\mathbb{N}$.

A central aspect of this language's macro embedding into $\mathcal{PF}$ is that the resulting terms have different types depending on whether the original terms are fully evaluated or still contain computation. For instance, the translation of the term $\mathtt{add}@(S(S(Z)), S(Z))$, which has computational content, has type $\mathtt{Shift}\langle\mathtt{Nat}\rangle$, while

[1] We only allow values and variables as arguments of S, but can easily lift S to a function `suc`, as shown in the example, to circumvent this restriction. Also, as in the call to `suc` in the example, we will omit the second argument (think of it as an arbitrary value) when it is ignored by the function, thereby encoding unary functions.

Value terms (trivial):
$A_Z() = \mathtt{z} \qquad A_{S(\cdot)}(t_v) = \mathtt{s}(t_v) \qquad A_{x_S}() = x_S \qquad A_y() = y$

Computation terms:

$$\begin{aligned}
A'_{f@(\cdot,\cdot)}(t_v^1, t_v^2) &= t_v^1 \gg f(t_v^2, \bar{k}) \\
A'_{f@(\cdot,\cdot)}(t_c^1, t_v^2) &= t_c^1 \gg \mathtt{sh}(f(t_v^2, \bar{k})) \\
A'_{f@(\cdot,\cdot)}(t_v^1, t_c^2) &= t_c^2 \gg \mathtt{sh}(\bar{\mu}\{n \Rightarrow t_v^1 \gg f(n, \bar{k})\}) \\
A'_{f@(\cdot,\cdot)}(t_c^1, t_c^2) &= t_c^1 \gg \mathtt{sh}(\bar{\mu}\{n_1 \Rightarrow t_c^2 \gg \mathtt{sh}(\bar{\mu}\{n_2 \Rightarrow n_1 \gg f(n_2, \bar{k})\})\}) \\
A'_{return(\cdot)}(t_v) &= t_v \gg \bar{k}
\end{aligned}$$

$$A_{\mathfrak{C}}(\overrightarrow{t}) = \mu\{\mathtt{sh}(\bar{k}) \Rightarrow A'_{\mathfrak{C}}(\overrightarrow{t})\}$$
for constructs $\mathfrak{C}$ forming computation terms

FIGURE 5.3: Syntactic abstractions for the embedding into $\mathcal{PF}$.

the translation of the fully evaluated term $S(S(S(Z)))$ has type `Nat`. The type `Shift` is defined as follows[2]:

$$\mathbf{data}_{-}\ \mathtt{Shift}\langle A\rangle\ \{\ \mathtt{sh}\langle A\rangle(\overline{A}) : \mathtt{Shift}\langle A\rangle\ \}$$

The idea is that a consumer $\bar{k}$ of type $\overline{A}$ (e.g. $\overline{\mathtt{Nat}}$) is available in the body of a producer for $\mathtt{Shift}\langle A\rangle$ (e.g. $\mathtt{Shift}\langle\mathtt{Nat}\rangle$), and to this $\bar{k}$ the result of the computation gets passed. For example, $\mathtt{add}@(S(S(Z)), S(Z))$ is translated to:

$$\mu\{\mathtt{sh}(\bar{k}) \Rightarrow \mathtt{s}(\mathtt{s}(\mathtt{z})) \gg \mathtt{add}(\mathtt{s}(\mathtt{z}), \bar{k})\}$$

Function definitions are translated such that a consumer argument for the "return" is introduced, and here $\bar{k}$ is passed as this argument to `add`. If we compose our example term with, e.g., a call to `suc`, such that the resulting term is $\mathtt{suc}(\mathtt{add}@(S(S(Z)), S(Z)))$, this is translated by combining the μ-abstraction above with a `sh` destructor applied to the `suc` consumer, to form a command that is in turn wrapped in a μ abstraction.

$$\mu\{\mathtt{sh}(\bar{k}) \Rightarrow \mu\{\mathtt{sh}(\bar{k}) \Rightarrow \mathtt{s}(\mathtt{s}(\mathtt{z})) \gg \mathtt{add}(\mathtt{s}(\mathtt{z}), \bar{k})\} \gg \mathtt{sh}(\mathtt{suc}(\bar{k}))\}$$

The consumer argument for `suc` comes from the outer μ-abstraction. Importantly, the local structure is kept intact as expected from a macro; i.e. in this example, the call to add is a constituent subterm before the translation, and so is its translation result. In the syntax given in Fig. 5.1 we have already anticipated the split between computation terms t_c and fully evaluated terms t_v (see also the remark on *return* below).[3] The macros for each kind of t_c and t_v are given in Fig. 5.3 in the style of *syntactic abstractions*, following Felleisen (1991) (see the exposition at the beginning of this chapter); we refer to the compositional macro translation induced by the syntactic abstractions as $\mathcal{M}$ and to the translation of computation terms with the outer μ-abstraction stripped off as $\mathcal{M}'$ (corresponding to the auxiliary syntactic abstractions A'). The macros come bundled with the usual Peano data type definition $\mathbf{data}_{+}\ \mathtt{Nat}\ \{\ \mathtt{z}\ |\ \mathtt{s}(\mathtt{Nat})\ \}$.

[2]This definition, assuming parameter A is only used for positive types, in **CU** corresponds to the shift $\uparrow$ from positive to negative; `Shift` can actually also be used for negative instantiations of A, since type parameters in $\mathcal{PF}$ are polarity-agnostic, but a proof of a negative type already contains computation, so wrapping this type in a `Shift` is superfluous. It should be noted that the negative-to-positive shift $\downarrow$ can be defined similarly. But due to the polarity-agnostic type parameters of $\mathcal{PF}$, polymorphic type formers resulting in positive types, e.g., sums, can directly use a negative type, leaving the shift implicit. Such (implicit) negative-to-positive shifts can be used to model more interesting variations of evaluation strategies; we discuss one such variation in Section 5.4.4.

[3]Thus, strictly speaking, the macro embedding requires a preprocessing that identifies which terms are fully evaluated, but this is easy to recognize.

$$
\begin{array}{ll}
\textbf{function}_{+}\ \texttt{add}(y, \bar{k})\ := \{ & \\
\quad \texttt{z} & \Rightarrow y \gg \bar{k} \\
\quad \texttt{s}(x_S) & \Rightarrow \mu\{\texttt{sh}(\bar{k}) \Rightarrow x_S \gg \texttt{add}(y, \bar{k})\} \gg \texttt{sh}(\texttt{suc}(\bar{k})) \\
\} &
\end{array}
$$

$$
\begin{array}{ll}
\textbf{function}_{+}\ \texttt{suc}(\bar{k})\ := \{ & \\
\quad \texttt{z} & \Rightarrow \texttt{s}(\texttt{z}) \gg \bar{k} \\
\quad \texttt{s}(x_S) & \Rightarrow \texttt{s}(\texttt{s}(x_S)) \gg \bar{k} \\
\} &
\end{array}
$$

FIGURE 5.4: Addition function example from Fig. 5.2, embedded into $\mathcal{PF}$.

Remark: *return*(t_v)**.** We clearly delineate when the evaluation of a subterm reaches a value by requiring the extra reduction step specified by the second reduction rule. Contraction formally may always only result in computation terms, and thus the definition of a function must use *return* if the body is directly a value or a variable, as in the zero case of the addition function (Fig. 5.2). Making the transition from one contraction sequence to another, with a new redex being picked (if any), explicit, will make it simpler to demonstrate the correspondence of the reduction of the translated term with a machine for the macro-level reduction.

What remains to define is the embedding of function definitions. To do so, we simply turn

$$f \mapsto (t_Z, t_S)$$

into:

$$
\begin{array}{ll}
\textbf{function}_{+}\ f(y, \bar{k})\ := \{ & \\
\quad \texttt{z} & \Rightarrow \mathcal{M}'(t_Z) \\
\quad \texttt{s}(x_S) & \Rightarrow \mathcal{M}'(t_S) \\
\} &
\end{array}
$$

That is, we add a consumer argument $\bar{k}$ to the function signature, and macro-translate the body without wrapping the result in a μ-abstraction, such that $\bar{k}$ is bound by the function signature. Fig. 5.4 shows the result of embedding the functions from Fig. 5.2. Especially, here we can see again how the term translation kept the local structure intact, as expected; e.g., $\texttt{add@}(x_S, y)$ was a constituent subterm (within the successor case of `add`) before the translation, and so is its embedding $\mu\{\texttt{sh}(\bar{k}) \Rightarrow x_S \gg \texttt{add}(y)\}$.

5.1.2 CBV reduction correspondence

We now turn to the question of how the reduction of our surface CBV language corresponds to the reduction of the translated terms in $\mathcal{PF}$. The short answer is that there is no one-to-one correspondence, but that there is a one-to-one correspondence between an abstract evaluation machine, which in turn is in a certain correspondence with surface reduction, and reduction in $\mathcal{PF}$. The underlying reason is that $\mathcal{PF}$ makes all control flow explicit while there are some implicit steps in $\triangleright_{\text{v}}$, particularly the search for the next redex within the currently considered term that can be contracted with $\blacktriangleright_{\text{v}}$. Also, the result of a macro translation for some term t does not

actually reduce, not even in multiple steps ($\triangleright^*$), to the result of the macro translation of t' with $t \triangleright_v t'$. Rather, it reduces to the result of macro translating the machine configuration for the evaluation machine that makes searching for redexes explicit. We now consider the formal details of the correspondence between surface (macro) level reduction and reduction in $\mathcal{PF}$, via the abstract evaluation machine.

Contraction correspondence. In the next paragraph we introduce the evaluation machine, but first we want to show that while reduction is not in one-to-one correspondence, each *contraction* step corresponds to a single $\mathcal{PF}$ reduction step via the macro translation; the machine is built upon this principle.[4] Intuitively, our contraction correspondence lemma should say that when a term t_1 contracts to a term t_2, then the result of macro translating t_1, a negative continuation k_1, should reduce to k_2, the result of macro translating t_2. While this does not work out directly (continuations do not reduce, commands do), there is a specific way how we can conceive of such a continuation reducing to another. For this we think of the matching abstraction for the shifted positive continuation k as a universal quantification. Stripping this abstraction off we obtain a command with the free variable k, and we think of a negative continuation as reducing to another when their unwrapped commands reduce when substituting an arbitrary, but the same for both commands, positive continuation for k (hence universal quantification):

Lemma 5.1. *When $t_1 \blacktriangleright_v t_2$, then for all positive continuations k_0: $\mathcal{M}'(t_1)[k_0] \triangleright \mathcal{M}'(t_2)[k_0]$.*

Proof. By distinguishing the two possible rules for deriving the contraction, obtaining the possible redexes by inversion; the rest is just unfolding of definitions. (Details in Appendix B.) □

As defined in Section 5.1.1, we write $\mathcal{M}'$ for $\mathcal{M}$[5] followed by stripping off the outer negative match, freeing the continuation variable (k), and we use the shorthand notation $c[v]$ for the substitution of v in a command c with only a single free variable. More generally, we will write $k^{'}$ for the command that results from taking the continuation k and stripping off the outer abstraction.

Evaluation machine. A *machine configuration* is a pair of an evaluation context and a computation term. We define the machine step relation $\rightsquigarrow$ as follows:

$$\begin{array}{llll} (1) & (\mathcal{E}, t) & \rightsquigarrow & (\mathcal{E}, t') \quad \text{if } t \blacktriangleright_v t' \\ (2) & (\mathcal{E}, \mathit{return}(v)) & \rightsquigarrow & (\mathcal{E}_0, \mathcal{E}'[v]) \quad \text{if } \mathcal{E} = \mathcal{E}_0[\mathcal{E}'], \mathcal{E}' \text{ has depth 1} \\ (3) & (\mathcal{E}, \mathcal{E}'[t_c]) & \rightsquigarrow & (\mathcal{E}[\mathcal{E}'], t_c) \quad \text{if } \mathcal{E}' \text{ has depth 1} \end{array}$$

The first rule captures reduction via contraction of a found redex, the second captures recomposition with the context once a value[6] is reached, while the third rule captures the search for the next redex, with accumulation of the context. We write $\rightsquigarrow_{(i)}$ when a step is derived by rule (i); observe that there is no recursive occurrence of $\rightsquigarrow$ in the definition of $\rightsquigarrow$, which means that the relation is simply the union $\bigcup_{i \in \{1,2,3\}} \rightsquigarrow_{(i)}$. This abstract machine can be seen as a variant of the CEK machine (Felleisen and Friedman, 1987), albeit without the environment (unnecessary in the

[4] We will later employ this fact when showing the one-to-one correspondence between machine steps and $\mathcal{PF}$ steps.

[5] The macro translation induced by the syntactic abstractions of Section 5.1.1.

[6] As pointed out in the remark above, our contraction relation is purposefully defined in such a way that only computation terms are reachable. It is this machine rule's task to unpack the value from a return clause and then deal with it in the appropriate way.

$$
\begin{array}{lll}
\mathcal{M}^{\text{ctx}}(f@(v,[])) & := & \bar{\mu}\{n \Rightarrow v \gg f(n,\bar{k})\} \\
\mathcal{M}^{\text{ctx}}(f@([],t)) & := & \bar{\mu}\{n \Rightarrow t \gg \mathtt{sh}(\bar{\mu}\{m \Rightarrow n \gg f(m,\bar{k})\})\} \\
\mathcal{T}([]) & := & \bar{\mu}\{n \Rightarrow \mathbf{Result}(n)\} \\
\mathcal{T}(\mathcal{E}[\mathcal{E}']) & := & \mathcal{M}^{\text{ctx}}(\mathcal{E}')[\mathcal{T}(\mathcal{E})],\text{ if } \mathcal{E}' \text{ has depth } 1 \\
\mathcal{T}((\mathcal{E},t)) & := & \mathcal{M}'(t)[\mathcal{T}(\mathcal{E})]
\end{array}
$$

FIGURE 5.5: Translating context frames and machine configurations

first-order setting) and presented with evaluation contexts (Felleisen and Hieb, 1992) taking the place of continuations.[7]

Correspondence of term reduction to machine steps. We can associate a semantically equivalent term with each machine configuration: $(\mathcal{E}, t) \risingdotseq \mathcal{E}[t]$; in particular, we refer to $([], t)$ as the *initial configuration for t*. Intuitively, machine steps are "only" more fine-grained than term reduction steps, making more aspects of the evaluation explicit but never semantically "moving away" from the reduction, i.e., the configurations stay associated with only the terms appearing in the reduction sequence and in the order they appear in within the sequence. Formally, we can prove the following lemma about this "tightness" of the correspondence.

Lemma 5.2. *For any two terms t_1, t_2 with $t_1 \rhd_v t_2$ and any machine configuration $c_1 \risingdotseq t_1$, there are configurations $c'_1, ..., c'_k \risingdotseq t_1$ $(k \geq 0)$ and $c_2 \risingdotseq t_2$ such that: $c_1 \leadsto c'_1 \leadsto ... \leadsto c'_k \leadsto c_2$.*

Proof. By distinguishing the two possible rules by which the reduction could have been derived. Then, for each case, the desired reduction sequence follows by a simple induction on the context entangled with the term in the configuration. (Details in Appendix B.) □

Translating machine configurations. Let us now consider whether our definition of the machine steps is sufficient for one-to-one correspondence with reduction in $\mathcal{PF}$. We need a non-compositional translation $\mathcal{T}$ from evaluation contexts to continuations and extend that to machine configurations. This translation $\mathcal{T}$ is similar to a CPS transformation; this is because its source are evaluation contexts which in our machine configurations play the role of containing "already finished" parts of the computation. Correspondingly, in Fig. 5.5 we define a translation $\mathcal{M}^{\text{ctx}}$ of single frames, i.e. contexts of depth 1, to positive continuations, and then use this to define $\mathcal{T}$. The translation $\mathcal{M}^{\text{ctx}}$ fits with term translation:

Lemma 5.3.

1. $\mathcal{M}'(\mathcal{E}[t_c]) = \mathcal{M}(t_c) \gg sh(\mathcal{M}^{ctx}(\mathcal{E}))$ *if $\mathcal{E}$ has depth* 1.
2. $\mathcal{M}'(\mathcal{E}[t_v])[k] = (\mathcal{M}^{ctx}(\mathcal{E})[k])'[t_v]$ *if $\mathcal{E}$ has depth* 1.

Proof. Straightforward from the macro definitions. □

Using Lemma 5.3 (2) we obtain a variant of the definitional equation for $\mathcal{T}$ for instantiation with values; here we write $\mathcal{T}'$ for $\mathcal{T}$ followed by stripping off the outer value abstraction (freeing the variable n).

[7] The same kind of presentation with tuples of terms and evaluation contexts, albeit using a different approach for the step rules and an additional label, is used in lecture notes by Ronald Garcia (`https://www.cs.ubc.ca/~rxg/cpsc509/05-abstract-machines.pdf`; accessed Nov. 6, 2020).

$$
\begin{array}{lcll}
f & \in & \mathbb{F} & \textit{(function names)} \\
t & ::= & t_v \mid t_c & \\
t_v & ::= & Z \mid S(t_v) \mid x_S & \\
t_c & ::= & f@(t, \boxed{t_c}) \mid return(t_v) \mid \boxed{y} & \\
\\
\mathcal{E} & ::= & \boxed{f@(\mathcal{E}, t) \mid []} & \\
v & ::= & Z \mid S(v) &
\end{array}
$$

$$
\begin{array}{lll}
f@(Z, \boxed{t}) & \blacktriangleright_n & \pi_1(\mathcal{D}(f))[y \mapsto \boxed{t}] \\
f@(S(v), \boxed{t}) & \blacktriangleright_n & \pi_2(\mathcal{D}(f))[x_S \mapsto v, y \mapsto \boxed{t}] \\
\mathcal{E}[t] & \triangleright_n & \mathcal{E}[t'] \quad \text{if } t \blacktriangleright_n t' \\
\mathcal{E}[return(t_v)] & \triangleright_n & \mathcal{E}[t_v] \quad \text{if } \mathcal{E} \neq []
\end{array}
$$

FIGURE 5.6: Syntax, contraction and reduction rules of a simple first-order CBN language (differences to CBV highlighted).

Lemma 5.4. *$\mathcal{T}'(\mathcal{E}[\mathcal{E}'])[v] = \mathcal{M}'(\mathcal{E}'[v])[\mathcal{T}(\mathcal{E})]$ if $\mathcal{E}'$ has depth 1.*

Proof.

$$\mathcal{T}'(\mathcal{E}[\mathcal{E}'])[v] = (\mathcal{T}(\mathcal{E}[\mathcal{E}']))'[v] = (\mathcal{M}^{\text{ctx}}(\mathcal{E}')[\mathcal{T}(\mathcal{E})])'[v] \overset{\text{Lem. 5.3(2)}}{=} \mathcal{M}'(\mathcal{E}'[v])[\mathcal{T}(\mathcal{E})] \qquad \square$$

Correspondence of machine steps to $\mathcal{PF}$ reduction. Taking a look at a simple example first, our translation seems to do fine:

$$
\begin{array}{ll|ll}
 & ([], \texttt{add@}(3,2)) & & 3 \gg \texttt{add}(2, \underbrace{\tilde{\mu}\{n \Rightarrow \textbf{Result}(n)\}}_{k}) \\
\rightsquigarrow_{(1)} & ([], \texttt{suc@}(\texttt{add@}(2,2))) & \triangleright & \texttt{add(2,2)} \gg \texttt{sh}(\texttt{suc}(k)) \\
\rightsquigarrow_{(3)} & (\texttt{suc@}([]), \texttt{add@}(2,2)) & \triangleright & 2 \gg \texttt{add}(2, \texttt{suc}(k))
\end{array}
$$

We can prove that the one-to-one correspondence holds in general:

Theorem 5.1. *For any two machine configurations with $c_1 \rightsquigarrow c_2$ it is $\mathcal{T}(c_1) \triangleright \mathcal{T}(c_2)$.*

Since $\rightsquigarrow = \bigcup_{i \in \{1,2,3\}} \rightsquigarrow_{(i)}$, to verify this theorem it suffices to separately consider each of the rules for $\rightsquigarrow$. For the three rules, the desired implication is a straightforward consequence of, respectively, Lemma 5.1, Lemma 5.4, and Lemma 5.3 (1); for details we refer to Appendix B.

Using Lemma 5.2 about the association between machine configurations and terms $\fallingdotseq$ we get the following easy corollary of Theorem 5.1:

Corollary 5.1. *When $t_1 \triangleright_v t_2$ and $c_1 \fallingdotseq t_1$ (e.g. c_1 is the initial configuration for t_1), then there are $c'_1, ..., c'_k \fallingdotseq t_1$ ($k \geq 0$) and $c_2 \fallingdotseq t_2$ such that $\mathcal{T}(c_1) \triangleright \mathcal{T}(c'_1) \triangleright ... \triangleright \mathcal{T}(c'_k) \triangleright \mathcal{T}(c_2)$.*

5.1.3 Embedding call-by-name

We now change the evaluation strategy of our first-order language to call-by-name, resulting in the syntax, contraction and reduction rules shown in Fig. 5.6. We only consider a simple flavor of call-by-name here, in which the second argument of a function call is not evaluated (reflected in the definition of evaluation contexts and contraction rules) but the first is. This suffices to demonstrate how we have to change our syntactic abstractions, shown in Fig. 5.7, to account for call-by-name evaluation.

In our syntax, compared to CBV we now have the variable y for the second argument as a computation term, and as second arguments to a function call we now only allow computation terms. The syntactic abstraction for the y variable does not have to change, but we will have to take care of the fact that it is not a continuation abstraction anymore when translating function bodies in order to always get

Value terms (trivial):

$A_Z() = \mathtt{z} \qquad A_{S(\cdot)}(t_v) = \mathtt{s}(t_v) \qquad A_{x_S}() = x_S$

Computation terms:

$$\begin{array}{lcl}
A'_{f@(\cdot,\cdot)}(t_v^1, t_c^2) & = & t_v^1 \gg f(t_c^2, \bar{k}) \\
A'_{f@(\cdot,\cdot)}(t_c^1, t_c^2) & = & t_c^1 \gg \mathtt{sh}(f(t_c^2, \bar{k})) \\
A'_{return(\cdot)}(t_v) & = & t_v \gg \bar{k} \\
 & & \\
A_{\mathfrak{C}}(\overrightarrow{t}) & = & \mu\{\mathtt{sh}(\bar{k}) \Rightarrow A'_{\mathfrak{C}}(\overrightarrow{t})\} \\
\multicolumn{3}{l}{\quad \text{for constructs } \mathfrak{C} \neq y \text{ forming computation terms}} \\
A_y() & = & y
\end{array}$$

FIGURE 5.7: Syntactic abstractions for the CBN embedding.

$$\begin{array}{ccccccccc}
t_0 & \rhd^{\leq n}_{\mathrm{n,t}} & \mathcal{E}[f@(1,t)] & & \rhd_{\mathrm{n,f}} & & \mathcal{E}[t] & \rhd^{\leq m}_{\mathrm{n,t}} & t_1 \\
\Downarrow\!\Uparrow & & \Downarrow\!\Uparrow & & & & \Downarrow\!\Uparrow & & \Downarrow\!\Uparrow \\
c_0 & \rightsquigarrow^n & (\mathcal{E}, f@(1,t)) & & \not\rightsquigarrow & & (\mathcal{E}, t) & \rightsquigarrow^m & c_1 \\
 & & & & & & & & \\
\downarrow \mathcal{T} & & \downarrow \mathcal{T} & & & & \downarrow \mathcal{T} & & \downarrow \mathcal{T} \\
 & & & & & & & & \\
\mathcal{T}(c_0) & \rhd^n & 1 \gg f(\mathcal{M}(t), \mathcal{T}[\mathcal{E}]) & \rhd & \mathcal{M}(t) \gg \mathtt{sh}(\mathcal{T}(\mathcal{E})) & \rhd & \mathcal{M}'(t)[\mathcal{T}(\mathcal{E})] & \rhd^m & \mathcal{T}(c_1)
\end{array}$$

FIGURE 5.8: CBN reduction correspondence for an example with $f \mapsto (y)$. Top: Thunk forcing term reduction step in between two macro-steps. Bottom: Corresponding $\mathcal{PF}$ steps.

commands at the top-level. In the syntactic abstractions for function calls, there is no evaluation of the second argument t_c happening anymore, instead it is passed directly to the function. Translation of function bodies is the same as for CBV, with the decisive difference that the second argument of the original function is now a computation term and hence the first argument of the function in $\mathcal{PF}$ is of shifted type. Further, as hinted at, when the function bodies are translated, a top-level (and only top-level) y becomes $y \gg \mathtt{sh}(\bar{k})$ since we must create a command here.

The correspondence between surface reduction and $\mathcal{PF}$ reduction is, like in the CBV case, not a direct one, but again there is a one-to-one correspondence between an evaluation machine and $\mathcal{PF}$ reduction. The demonstration of this one-to-one correspondence and the correspondence between surface reduction and machine steps is very similar to what we saw for CBV, but we now distinguish different phases, or *macro-steps*, of the evaluation. All reduction steps up until one that leads to a function body which is a variable y that would then next be instantiated form one such macro-step. The next step (which we call *thunk forcing*) takes care of getting the evaluation of the until now unevaluated term, as substituted for y, started, which we capture by the definition following below; after this step another macro-step follows, and so on until a macro-step leads to a value. Fig. 5.8 shows an example with two macro-steps and the corresponding $\mathcal{PF}$ steps per machine step comprising the macro-steps and for the thunk forcing term reduction step.

Definition 5.1. *We call a contraction step* $t_1 \blacktriangleright_n t_2$ thunk forcing, *in symbols* $t_1 \blacktriangleright_{n,f} t_2$, *when* $t_1 = f@(n,t)$ *and* $\pi_i(\mathcal{D}(f)) = y$, *with* $i = 1$ *if* $n = Z$ *and* $i = 2$ *otherwise.*

When a contraction is not thunk forcing, we denote it with $t_1 \blacktriangleright_{\mathrm{n,t}} t_2$ (with a t for *thunked*). We lift the notions of thunk forcing and not thunk forcing to reduction: When the reduction step is derived by the first rule using a $\blacktriangleright_{\mathrm{n,t}}$ step, or by the second

rule, we denote it with $\triangleright_{\mathrm{n,t}}$. When the reduction step is derived by the first rule using a $\blacktriangleright_{\mathrm{n,f}}$ step, we denote it with $\triangleright_{\mathrm{n,f}}$. Combined, the not thunk forcing steps $\triangleright^*_{\mathrm{n,t}}$ until the next thunk forcing step $\triangleright_{\mathrm{n,f}}$ formally characterize a macro-step. Similarly to CBV, we have a direct correspondence between contraction and $\mathcal{PF}$ reduction, but in this case the correspondence is restricted to the $\blacktriangleright_{\mathrm{n,t}}$ steps.

Lemma 5.5. *When $t_1 \blacktriangleright_{n,t} t_2$, then for all positive continuations k_0: $\mathcal{M}'(t_1)[k_0] \triangleright \mathcal{M}'(t_2)[k_0]$.*

Proof. Almost identically to the corresponding Lemma 5.1 for CBV, making use of y not being the body in the relevant case of the applied function. □

Our evaluation machine is, like in the CBV case, build on top of contraction. The actual rules for the machine are identical to the CBV case, but we restrict the contractions taken into account to the not thunk forcing ones ($\blacktriangleright_{\mathrm{n,t}}$).

(1)	$(\mathcal{E}, t)$	$\rightsquigarrow$	$(\mathcal{E}, t')$	if $t \blacktriangleright_{\mathrm{n,t}} t'$
(2)	$(\mathcal{E}, return(v))$	$\rightsquigarrow$	$(\mathcal{E}_0, \mathcal{E}'[v])$	if $\mathcal{E} = \mathcal{E}_0[\mathcal{E}']$, $\mathcal{E}'$ has depth 1
(3)	$(\mathcal{E}, \mathcal{E}'[t_c])$	$\rightsquigarrow$	$(\mathcal{E}[\mathcal{E}'], t_c)$	if $\mathcal{E}'$ has depth 1

Similarly to CBV, there is a close correspondence between $\blacktriangleright_{\mathrm{n,t}}$ and $\rightsquigarrow$, where again $(\mathcal{E}, t) \leftrightharpoons \mathcal{E}[t]$:

Lemma 5.6. *For any two terms t_1, t_2 with $t_1 \triangleright_{n,t} t_2$ and any machine configuration $c_1 \leftrightharpoons t_1$, there are configurations $c'_1, ..., c'_k \leftrightharpoons t_1$ ($k \geq 0$) and $c_2 \leftrightharpoons t_2$ such that: $c_1 \rightsquigarrow c'_1 \rightsquigarrow ... \rightsquigarrow c'_k \rightsquigarrow c_2$.*

Proof. The proof of Lemma 5.2 can be reused as is, with $\triangleright_{\mathrm{n,t}}$ taking the place of $\triangleright_{\mathrm{v}}$. □

However, like the macro-steps which do not always end in a value, the transitive closure of the machine step relation $\rightsquigarrow^*$ does not for all starting configurations c contain a pair of c and a final configuration $([], return(v))$ where v is a value. Instead, in general the machine has to be run multiple times with thunk forcing interspersed. By the following lemma (plus remark) we know that there is always a one-to-one correspondence[8] for such an interspersed step and $\mathcal{PF}$ reduction that is compatible with the machine steps preceding the interspersed step. As in the CBV case, $\mathcal{T}$ refers to the translation of machine configurations, shown in Fig. 5.9.

Lemma 5.7. *Consider any two terms t_1, t_2 with $t_1 \triangleright_{n,f} t_2$, i.e., by definition of $\triangleright_{n,f}$, $t_1 = \mathcal{E}[f@(n,t)]$ for some n and t. For any machine configuration $c_1 \leftrightharpoons t_1$ with the form $c_1 = (\mathcal{E}, f@(n,t))$ (this restriction is insignificant, see the remark below):*
$\mathcal{T}(c_1) \triangleright force(t, \mathcal{E}) \triangleright \mathcal{T}(c_2)$ for some $c_2 \leftrightharpoons t_2$.
We define $force(t, \mathcal{E}) := \mathcal{M}(t) \gg \mathtt{sh}(\mathcal{T}(\mathcal{E}))$.

Proof. By the definition of $\triangleright_{\mathrm{n,f}}$ it is $t_2 = \mathcal{E}[t]$. The desired $c_2 \leftrightharpoons t_2$ is simply $c_2 = (\mathcal{E}, t)$, since:

$$\begin{aligned}\mathcal{T}(c_1) = \mathcal{M}'(f@(n,t))[\mathcal{T}(\mathcal{E})] = n \gg f(\mathcal{M}(t), \mathcal{T}(\mathcal{E}))\\ \triangleright \mathcal{M}(t) \gg \mathtt{sh}(\mathcal{T}(\mathcal{E})) \triangleright \mathcal{M}'(t)[\mathcal{T}(\mathcal{E})] = \mathcal{T}(\mathcal{E}, t).\end{aligned}$$
□

Remark 5.1. *The restriction to machine configurations c_1 of the form $(\mathcal{E}, f@(n,t))$ is insignificant for the overall correspondence, since in the general case we would just have more context frames wrapped around $f@(n,t)$ in the second component of the configuration. Thus there would just follow some more machine steps $\rightsquigarrow_{(3)}$ that move these frames to the first*

[8] It takes two steps from $\mathcal{T}(c_1)$ to $\mathcal{T}(c_2)$ in the lemma below, but the first step is for carrying out the function call (i.e. lookup and substitution) that leads to the actual thunk forcing.

$$\begin{array}{lll} \mathcal{M}^{\text{ctx}}(f@([],t)) & := & f(t,\bar{k}) \\ \mathcal{T}([]) & := & \tilde{\mu}\{n \Rightarrow \textbf{Result}(n)\} \\ \mathcal{T}(\mathcal{E}[\mathcal{E}']) & := & \mathcal{M}^{\text{ctx}}(\mathcal{E}')[\mathcal{T}(\mathcal{E})], \text{ if } \mathcal{E}' \text{ has depth 1} \\ \mathcal{T}((\mathcal{E},t)) & := & \mathcal{M}'(t)[\mathcal{T}(\mathcal{E})] \end{array}$$

FIGURE 5.9: Translating context frames and machine configurations (CBN).

component (in particular upholding $\fallingdotseq$ to the original term) until the form of machine configuration required for the lemma is reached.

Last but not least, as for CBV, we can show that the machine steps are in one-to-one correspondence to translated steps in $\mathcal{PF}$.

Theorem 5.2. *For any two machine configurations with $c_1 \rightsquigarrow c_2$, it is $\mathcal{T}(c_1) \triangleright \mathcal{T}(c_2)$.*

Just as for CBV, this is proved by separately considering the three rules for $\rightsquigarrow$, in each case straightforwardly applying the relevant lemma: Lemma 5.5 for rule (1), and direct adaptations of Lemma 5.4, and Lemma 5.3 (1) for rule (2) and (3), respectively; these lemmas are proven in the same way as their CBV counterparts. Together with Lemma 5.6 and Lemma 5.7 we obtain as a direct corollary the (up to the possibly interspersed step) same statement concerning the correspondence between surface and $\mathcal{PF}$ reduction that we also have for CBV:

Corollary 5.2. *When $t_1 \rhd_n t_2$ and $c_1 \fallingdotseq t_1$ (e.g. c_1 is the initial configuration for t_1), then there are $c'_1, ..., c'_k \fallingdotseq t_1$ ($k \geq 0$) and $c_2 \fallingdotseq t_2$ (and possibly t_f, $\mathcal{E}_f$ s.t. $\mathcal{E}_f[t_f] = t_2$) such that*

$$\mathcal{T}(c_1) \triangleright \mathcal{T}(c'_1) \triangleright ... \triangleright \mathcal{T}(c'_k) \; (\triangleright \mathit{force}(t_f, \mathcal{E}_f)) \triangleright \mathcal{T}(c_2)$$

(optional step in braces).

It is also possible to embed call-by-need as a *daimonic* modification of call-by-name with a caching (memoization) daimon, which we consider next.

5.1.4 Embedding call-by-need

So far we have seen how to translate call-by-value and call-by-name languages into $\mathcal{PF}$. Call-by-need has a bit of a different nature in that it involves an extra-logical aspect of caching intermediate computations, which should in $\mathcal{PF}$ be modelled by a daimon. We can just use the mutable state daimon from the end of Section 4.2.3 for this purpose (with one little variation in the technical details, see below), though we will only make use of it in a limited fashion.[9]

Specifically, we need **NewBox** to put a negative producer of type $\texttt{Shift}\langle T\rangle$, which encapsulates some computation, into the store, say, at address α. Once the value v that this computation produces is requested, **OpenBox** retrieves the producer and carries out the computation, followed by **SetBox** which replaces the storage cell content at α with v. In later requests to the value associated with α, the stored value v can now be reused. Conceptually, in the usual terminology of lazy evaluation, α together with the content at α is a *thunk*, i.e. an entity that caches a computation result, since the content changes from having type $\texttt{Shift}\langle T\rangle$ (i.e., it contains computation) to T (i.e. it is a value).

[9]Modifying the call-by-name embedding to make use of the daimon has some similarities to the enhancement of call-by-name operational semantics with constructs for modifying the heap in the compiler intermediate language based on sequent calculus of Downen et al. (2016).

$$
\begin{aligned}
\mathcal{M}^{\text{lazy}}(f \mapsto (t_0, t_S)) &= f(y : \texttt{Ref}\langle\texttt{Nat}\rangle, k : \overline{\texttt{Nat}}) : \overline{\texttt{Nat}} := \{ \\
&\quad \texttt{Zero} \Rightarrow \mathcal{M}'(t_0), \\
&\quad \texttt{Succ}(n) \Rightarrow \mathcal{M}'(t_S)\,\} \\
A^{\text{lazy}}_{f@(,)}(t_c^1, t_c^2) &= \mu\{\texttt{sh}(k) \Rightarrow \textbf{NewBox}(t_c^2, \tilde{\mu}\{\alpha \Rightarrow t_c^1 \gg \texttt{sh}(f(\alpha, k))\})\} \\
A^{\text{lazy}}_{y}() &= \mu\{\texttt{sh}(k) \Rightarrow \textbf{OpenBox}(y, k, \\
&\quad \tilde{\mu}\{t_c \Rightarrow \texttt{sh}(\tilde{\mu}\{v \Rightarrow \textbf{SetBox}(y, v, v \gg k)\})\})\}
\end{aligned}
$$

FIGURE 5.10: Macros for call-by-need.

Making this formally precise gives us the modification of the call-by-name embedding shown in Fig. 5.10, obtaining a translation of a call-by-need language into $\mathcal{PF}$. Compared to the call-by-name macros, there is one additional macro for reference variables: Instead of directly using the $\mathcal{PF}$ variable y in the body as for call-by-name, we now wrap each occurrence of a reference variable using this macro to use the **OpenBox** command (with one decisive alteration, see below). Correspondingly, the second argument of functions, accessed by y, is now of type $\texttt{Ref}\langle\texttt{Nat}\rangle$ (or generally $\texttt{Ref}\langle T\rangle$ for some data type T). The function application macro now uses **NewBox** to bring the thunk on the heap and make the reference to it available.

The macro for variable y uses a variant of **OpenBox** with two continuation arguments, which triggers the first continuation in case the reference pointed to a value, and the second in case the reference pointed to a producer p encapsulating a computation. As the first continuation argument, the result continuation k is passed. As the second continuation argument, a continuation is passed which first updates, using **SetBox**, the content of the cell with the result v of the computation of p, thereby caching that result, then passes v to k.

This concludes our consideration of embeddings for different evaluation strategies. In the next section we consider how we can employ the embedding technique introduced in this section for full data and codata language fragments (we only consider call-by-value, but the approach can be readily adapted for other evaluation strategies) and how the symmetry of these surface languages is reflected in $\mathcal{PF}$ via the embeddings.

5.2 Data and Codata

5.2.1 Data fragment

This section generalizes the simple call-by-value language from the previous section to arbitrary data types and functions of arbitrary arity ≥ 1; the first argument is required and the only one that is pattern matched upon. This language is, up to presentation details (the explicit computation and value term distinction, *return*, and the necessary lifting of constructors to functions), the *data fragment* of Rendel, Trieflinger, and Ostermann (2015). The next section considers the *codata fragment*, which Rendel, Trieflinger, and Ostermann showed to be the image of program transposition of data fragment programs and vice versa.

Fig. 5.11 shows the syntax and operational semantics of our presentation of the data fragment. As a simple example we can again consider the addition function, shown in Fig. 5.12.[10] A reduction example involving addition is shown in Fig. 5.13.

[10] In examples, we leave out the types of constructors (and later, similarly, that of destructors) in cases of a function, say with name f, since it can be inferred from the type T of the argument that f pattern matches upon (which is specified in its name, i.e. $f = (T, ...)$).

$\mathbb{D}$: *data type names*
$\mathbb{C} \subseteq \mathbb{D} \times \mathbb{C}'$ (*constructor names, where $\mathbb{C}'$ is some set of names*)
$\mathbb{F} \subseteq \mathbb{D} \times \mathbb{F}'$ (*function names, where $\mathbb{F}'$ is some set of names*)

$$
\begin{array}{lcl}
C & \in & \mathbb{C} \\
f & \in & \mathbb{F} \\
\\
t & ::= & t_v \mid t_c \\
t_v & ::= & C(\overline{t_v}) \mid x_i \mid y_i \quad (i \in \mathbb{N}) \\
t_c & ::= & f(\bar{t}) \mid \mathit{return}(t_v)
\end{array}
$$

Global signatures and declarations:
$\Sigma^{\mathbb{C}} : \mathbb{C} \to \mathbb{D}^*$ *(constructor signatures)*
$\Sigma^{\mathbb{F}} : \mathbb{F} \to \mathbb{D}^* \times \mathbb{D}$ *(function signatures)*
$\mathcal{F} : \mathbb{F} \to (\mathbb{C} \nrightarrow t_c)$ *(function declarations)*

Operational semantics:

$$
\begin{array}{lcl}
\mathcal{E} & ::= & f(\overline{v}, \mathcal{E}, \overline{t}) \mid [] \\
v & ::= & C(\overline{v}) \\
\\
f(C(\overline{v}), \overline{v'}) & \blacktriangleright_{\mathrm{v}} & \mathcal{F}(f)(C)[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{v'}] \\
\mathcal{E}[t] & \rhd_{\mathrm{v}} & \mathcal{E}[t'] \quad \text{if } t \blacktriangleright_{\mathrm{v}} t' \quad (1) \\
\mathcal{E}[\mathit{return}(t_v)] & \rhd_{\mathrm{v}} & \mathcal{E}[t_v] \quad \text{if } \mathcal{E} \neq [] \quad (2)
\end{array}
$$

FIGURE 5.11: Syntax and operational semantics (data fragment).

$$
\begin{array}{l}
\mathbb{D} := \{\texttt{Nat}\}, \mathbb{C} := \{(\texttt{Nat}, \texttt{z}), (\texttt{Nat}, \texttt{s})\}, \\
\mathbb{F} := \{\texttt{add}, \texttt{suc}\}, \\
\Sigma^{\mathbb{C}} := \{(\texttt{Nat}, \texttt{z}) \mapsto (), (\texttt{Nat}, \texttt{s}) \mapsto (\texttt{Nat})\}, \\
\Sigma^{\mathbb{F}} := \{(\texttt{Nat}, \texttt{add}) \mapsto ((\texttt{Nat}), \texttt{Nat}), (\texttt{Nat}, \texttt{suc}) \mapsto ((), \texttt{Nat})\}, \\
\mathcal{F} := \{ \\
\quad (\texttt{Nat}, \texttt{add}) \mapsto \{\texttt{z} \mapsto \mathit{return}(y_1), \texttt{s} \mapsto \texttt{suc}(\texttt{add}(x_1, y_1))\}, \\
\quad (\texttt{Nat}, \texttt{suc}) \mapsto \{\texttt{z} \mapsto \mathit{return}(\texttt{s}(\texttt{z})), \texttt{s} \mapsto \mathit{return}(\texttt{s}(\texttt{s}(x_1)))\}\}.
\end{array}
$$

FIGURE 5.12: Data fragment example: natural numbers with addition.

$$
\begin{array}{ll}
 & \texttt{add}(\texttt{s}(\texttt{s}(\texttt{z})), \texttt{s}(\texttt{z})) \\
\rhd_{\mathrm{v},(1)} & \texttt{suc}(\texttt{add}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))) \\
\rhd_{\mathrm{v},(1)} & \texttt{suc}(\texttt{suc}(\texttt{add}(\texttt{z}, \texttt{s}(\texttt{z})))) \\
\rhd_{\mathrm{v},(1)} & \texttt{suc}(\texttt{suc}(\mathit{return}(\texttt{s}(\texttt{z})))) \\
\rhd_{\mathrm{v},(2)} & \texttt{suc}(\texttt{suc}(\texttt{s}(\texttt{z}))) \\
\rhd_{\mathrm{v},(1)} & \texttt{suc}(\mathit{return}(\texttt{s}(\texttt{s}(\texttt{z})))) \\
\rhd_{\mathrm{v},(2)} & \texttt{suc}(\texttt{s}(\texttt{s}(\texttt{z}))) \\
\rhd_{\mathrm{v},(1)} & \mathit{return}(\texttt{s}(\texttt{s}(\texttt{s}(\texttt{z}))))
\end{array}
$$

FIGURE 5.13: Addition reduction example.

$$\frac{C = (T, ...) \qquad \Sigma^{\mathbb{C}}(C) = \overline{T'} \qquad \overline{_ \vdash t_v : T'}}{_ \vdash C(\overline{t_v}) : T}\ \textsc{T-Constr}$$

$$\frac{\Sigma^{\mathbb{C}}(C) = \overline{T}}{f, C \vdash x_i : T_i}\ \textsc{T-Var-x} \qquad \frac{\Sigma^{\mathbb{F}}(f) = (\overline{T}, ...)}{f, C \vdash y_i : T_i}\ \textsc{T-Var-y}$$

$$\frac{f \in \mathbb{F} \qquad \Sigma^{\mathbb{F}}(f) = (\overline{T'}, T) \qquad \overline{_ \vdash t : T'}}{_ \vdash f(\bar{t}) : T}\ \textsc{T-Fun}$$

$$\frac{_ \vdash t_v : T}{_ \vdash return(t_v) : T}\ \textsc{T-Ret}$$

$$\frac{\begin{array}{c}\Sigma^{\mathbb{F}}(f) = (..., T'),\ f = (T, ...) \\ \forall C \in \mathrm{dom}(\mathcal{F}(f)).\ f, C \vdash \mathcal{F}(f)(C) : T' \\ \mathrm{dom}(\mathcal{F}(f)) = \{C \in \mathbb{C} \mid C = (T, ...)\}\end{array}}{\vdash f \text{ OK}}\ \textsc{Wf-Fun}$$

FIGURE 5.14: Type system (data fragment).

Value terms (trivial):
$A_{C(\cdot,...,\cdot)}(\overrightarrow{t_v}) = C(\overrightarrow{t_v}) \qquad A_{x_i}() = x_i \qquad A_{y_i}() = y_i$

Computation terms:

$$\begin{array}{lcl}
A'_{f@(\cdot,\cdot)}(t_v^1, t_v^2) & = & t_v^1 \gg f(t_v^2, \bar{k}) \\
A'_{f@(\cdot,\cdot)}(t_c^1, t_v^2) & = & t_c^1 \gg \mathtt{sh}(f(t_v^2, \bar{k})) \\
A'_{f@(\cdot,\cdot)}(t_v^1, t_c^2) & = & t_c^2 \gg \mathtt{sh}(\bar{\mu}\{n \Rightarrow t_v^1 \gg f(n, \bar{k})\}) \\
A'_{f@(\cdot,\cdot)}(t_c^1, t_c^2) & = & t_c^1 \gg \mathtt{sh}(\bar{\mu}\{n_1 \Rightarrow t_c^2 \gg \mathtt{sh}(\bar{\mu}\{n_2 \Rightarrow n_1 \gg f(n_2, \bar{k})\})\})
\end{array}$$

(etc. for other arities)

$$A'_{return(\cdot)}(t_v) \quad = \quad t_v \gg \bar{k}$$

$$A_{\mathfrak{C}}(\overrightarrow{t}) \quad = \quad \mu\{\mathtt{sh}(\bar{k}) \Rightarrow A'_{\mathfrak{C}}(\overrightarrow{t})\}$$

for constructs $\mathfrak{C}$ forming computation terms

FIGURE 5.15: Syntactic abstractions (data fragment).

The data fragment has a simple type system, shown in Fig. 5.14, that checks whether function calls and variable uses comply with the signatures. The judgment $f, C \vdash t : T$ says that term t has type $T \in \mathbb{D}$ when in the body of function f for constructor case C. We use an underscore to indicate that the respective definition site is not relevant and is merely passed to the premises. For $f \in \mathbb{F}, \vdash f$ OK says that f is wellformed, i.e. its cases' bodies typecheck under the appropriate definition site contexts and these cases are exactly the ones for all the constructors of the correct type (the function's input type).

The syntactic abstractions for the data fragment, shown in Fig. 5.15, are straightforwardly extended from those for the simple CBV language from the previous section. Constructor signatures are translated by just taking them over syntactically identical. Thus the entire data type declarations are also taken over syntactically identical. Computation terms t_c surface-typed T are $\mathcal{PF}$-typed $\mathtt{Shift}\langle T\rangle$, and value terms t_v surface-typed T are $\mathcal{PF}$-typed T. Embedding of function definitions is also straightforwardly extended to an arbitrary number of arguments and arbitrary argument types.

$\mathbb{C}$: *codata type names*
$\mathbb{D} \subseteq \mathbb{C} \times \mathbb{D}'$ (*destructor names, where* $\mathbb{D}'$ *is some set of names*)
$\mathbb{F} \subseteq \mathbb{C} \times \mathbb{F}'$ (*function names, where* $\mathbb{F}'$ *is some set of names*)

$$
\begin{array}{lcl}
d & \in & \mathbb{D} \\
f & \in & \mathbb{F} \\
\\
t & ::= & t_v \mid t_c \\
t_v & ::= & f(\overline{t_v}) \mid x_i \mid y_i \qquad (i \in \mathbb{N}) \\
t_c & ::= & d(t, \overline{t}) \mid return(t_v)
\end{array}
$$

Global signatures and declarations:
$\Sigma^{\mathbb{D}} : \mathbb{D} \to \mathbb{C}^* \times \mathbb{C}$ *(destructor signatures)*
$\Sigma^{\mathbb{F}} : \mathbb{F} \to \mathbb{C}^*$ *(function signatures)*
$\mathcal{F} : \mathbb{F} \to (\mathbb{D} \twoheadrightarrow t_c)$ *(function declarations)*

Operational semantics:

$$
\begin{array}{lcl}
\mathcal{E} & ::= & d(\overline{v}, \mathcal{E}, \overline{t}) \mid [] \\
v & ::= & f(\overline{v}) \\
\\
d(f(\overline{v}), \overline{v'}) & \blacktriangleright_{\text{v}} & \mathcal{F}(f)(d)[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{v'}] \\
\mathcal{E}[t] & \triangleright_{\text{v}} & \mathcal{E}[t'] \quad \text{if } t \blacktriangleright_{\text{v}} t' \quad (1) \\
\mathcal{E}[return(t_v)] & \triangleright_{\text{v}} & \mathcal{E}[t_v] \quad \text{if } \mathcal{E} \neq [] \quad (2)
\end{array}
$$

FIGURE 5.16: Syntax and operational semantics (codata fragment).

5.2.2 Codata fragment and transposition

The dual (in the sense of Rendel, Trieflinger, and Ostermann (2015)) of the data fragment, the codata fragment, is presented in Fig. 5.16. Rendel, Trieflinger, and Ostermann (2015) deliberately designed the data and codata fragments such that their respective syntaxes are isomorphic, i.e. each construct has a structurally identical counterpart (and the presentation of the fragments in this section preserves this). What distinguishes the fragments is that one has data types and the other has codata types and the terms are accordingly organized into functions in different ways. This was already considered in-depth in Chapter 2, along with the matrix view of programs that this gives rise to and which is also the view chosen for $\mathcal{PF}$. Here we just quickly give a formal definition of when a data and a codata program are related by transposition, fitting with the formal language specifications from this section. A program $\mathcal{P} \in \textsc{Data}$ or $\mathcal{P} \in \textsc{Codata}$, i.e. from either the data fragment or codata fragment, is a triple $(\Sigma^{\mathbb{C}}/\Sigma^{\mathbb{D}}, \Sigma^{\mathbb{F}}, \mathcal{F})$ (with either constructor signatures $\Sigma^{\mathbb{C}}$ or destructor signatures $\Sigma^{\mathbb{D}}$).

Definition 5.2. *Two programs* $\mathcal{P}_d = (\Sigma^{\mathbb{C}}, \Sigma^{\mathbb{F}}_d, \mathcal{F}^d) \in \textsc{Data}$ *and* $\mathcal{P}_c = (\Sigma^{\mathbb{D}}, \Sigma^{\mathbb{F}}_c, \mathcal{F}^c) \in \textsc{Codata}$ *are* related by CoDATA-transposition*, in symbols* $\mathcal{P}_d \leftrightarrow_{\textsc{Codata}} \mathcal{P}_c$*, iff:* $\Sigma^{\mathbb{C}} = \Sigma^{\mathbb{F}}_c$, $\Sigma^{\mathbb{D}} = \Sigma^{\mathbb{F}}_d$*, and for all* $C \in \Sigma^{\mathbb{C}}$ *and* $d \in \Sigma^{\mathbb{D}}$ *it is* $\mathcal{F}^d(d)(C) = \mathcal{F}^c(C)(d)$*.*

The syntactic abstractions for our embedding of the codata fragment into $\mathcal{PF}$ are shown in Fig. 5.17 (top).[11] For any structurally identical pair of constructs $\mathfrak{C}_1$, $\mathfrak{C}_2$ from the data and codata fragment, respectively, the syntactic abstraction for $\mathfrak{C}_1$ is *almost* identical to that for $\mathfrak{C}_2$. The difference is caused by the fact that there is no local consumer abstraction for negative types, hence in order to keep the $\tilde{\mu}$-abstractions after transposition we introduce a generic positive wrapper type $\texttt{WrapP}\langle A\rangle\{\texttt{pos}(A)\}$. The

[11] Add to these the translation of destructor signatures; they are translated just like consumer function signatures in the first-order CBV language and the data fragment: $d(T, \overrightarrow{T}) : T'$ becomes $d(T, \overrightarrow{T}, \overline{T'})$ (the output type T' becoming a consumer type).

Value terms (trivial):
$A_{f(\cdot,\ldots,\cdot)}(\overrightarrow{t_v}) = f(\overrightarrow{t_v}) \qquad A_{x_i}() = x_i \qquad A_{y_i}() = y_i$

Computation terms:

$$\begin{array}{lcl}
A'_{d(\cdot,\cdot)}(t_v^1, t_v^2) & = & t_v^1 \gg d(t_v^2, \texttt{wrap}(\bar{k})) \\
A'_{d(\cdot,\cdot)}(t_c^1, t_v^2) & = & t_c^1 \gg \texttt{sh}(\texttt{posk}(d(t_v^2, \texttt{wrap}(\bar{k})))) \\
A'_{d(\cdot,\cdot)}(t_v^1, t_c^2) & = & t_c^2 \gg \texttt{sh}(\bar{\mu}\{\texttt{pos}(a) \Rightarrow t_v^1 \gg d(a, \texttt{wrap}(\bar{k}))\}) \\
A'_{d(\cdot,\cdot)}(t_c^1, t_c^2) & = & t_c^1 \gg \texttt{sh}(\bar{\mu}\{\texttt{pos}(a_1) \Rightarrow t_c^2 \gg \texttt{sh}(\bar{\mu}\{\texttt{pos}(a_2) \Rightarrow a_1 \gg d(a_2, \texttt{wrap}(\bar{k}))\})\})
\end{array}$$

(etc. for other arities)

$$A'_{return(\cdot)}(t_v) \quad = \quad t_v \gg \texttt{wrap}(\bar{k})$$

$$A_{\mathfrak{C}}(\overrightarrow{t}) \quad = \quad \mu\{\texttt{sh}(\bar{k}) \Rightarrow A'_{\mathfrak{C}}(\overrightarrow{t})\}$$

for constructs $\mathfrak{C}$ forming computation terms

Value terms (trivial):
$A_{C(\cdot,\ldots,\cdot)}(\overrightarrow{t_v}) = f(\overrightarrow{t_v}) \qquad A_{x_i}() = x_i \qquad A_{y_i}() = y_i$

Computation terms:

$$\begin{array}{lcl}
A'_{f(\cdot,\cdot)}(t_v^1, t_v^2) & = & t_v^1 \gg f(t_v^2, \texttt{wrap}(\bar{k})) \\
A'_{f(\cdot,\cdot)}(t_c^1, t_v^2) & = & t_c^1 \gg \texttt{sh}(\texttt{posk}(f(t_v^2, \texttt{wrap}(\bar{k})))) \\
A'_{f(\cdot,\cdot)}(t_v^1, t_c^2) & = & t_c^2 \gg \texttt{sh}(\bar{\mu}\{\texttt{pos}(a) \Rightarrow t_v^1 \gg f(a, \texttt{wrap}(\bar{k}))\}) \\
A'_{f(\cdot,\cdot)}(t_c^1, t_c^2) & = & t_c^1 \gg \texttt{sh}(\bar{\mu}\{\texttt{pos}(a_1) \Rightarrow t_c^2 \gg \texttt{sh}(\bar{\mu}\{\texttt{pos}(a_2) \Rightarrow a_1 \gg f(a_2, \texttt{wrap}(\bar{k}))\})\})
\end{array}$$

(etc. for other arities)

$$A'_{return(\cdot)}(t_v) \quad = \quad t_v \gg \texttt{wrap}(\bar{k})$$

$$A_{\mathfrak{C}}(\overrightarrow{t}) \quad = \quad \mu\{\texttt{sh}(\bar{k}) \Rightarrow A'_{\mathfrak{C}}(\overrightarrow{t})\}$$

for constructs $\mathfrak{C}$ forming computation terms

FIGURE 5.17: Syntactic abstractions for the codata fragment (top), and modified abstractions for the data fragment isomorphic to these (bottom).

```
function+ add(y1) := {
  z      ⇒ y1
  s(x1)  ⇒
    suc(add(x1,y1))
}

function+ suc := {
  z      ⇒ s(z)
  s(x1)  ⇒ s(s(x1))
}
-----------------------------
function− z := {
  add(y1)  ⇒ y1
  suc      ⇒ s(z)
}

function− s(x1) := {
  add(y1)  ⇒
    suc(add(x1,y1))
  suc      ⇒ s(s(x1))
}
```

```
function+ add(y1,k̄) := {
  z      ⇒ y1 ≫ k̄
  s(x1)  ⇒
    μ{sh(k̄) ⇒ x1 ≫ add(y1,wrap(k̄))} ≫ sh(posk(suc(k̄)))
}

function+ suc(k̄) := {
  z      ⇒ s(z) ≫ k̄
  s(x1)  ⇒ s(s(x1)) ≫ k̄
}
-----------------------------
function− z := {
  add(y1,k̄)  ⇒ y1 ≫ k̄
  suc(k̄)     ⇒ s(z) ≫ k̄
}

function− s(x1) := {
  add(y1,k̄)  ⇒
    μ{sh(k̄) ⇒ x1 ≫ add(y1,wrap(k̄))} ≫ sh(posk(suc(k̄)))
  suc(k̄)     ⇒ s(s(x1)) ≫ k̄
}
```

FIGURE 5.18: Addition function example from Fig. 5.2 and its transposition (left), and the respective embeddings (right). The embeddings are themselves related by transposition.

shift type is modified to take such a wrapped type: $\texttt{Shift}\langle A\rangle\{\texttt{sh}\langle A\rangle(\texttt{WrapP}\langle A\rangle)\}$. We use generic auxiliary functions posk and wrap to convert between consumers of A and consumers of $\texttt{WrapP}\langle A\rangle$. The key idea is now that we can modify the data fragment embedding in the same way such that it is actually isomorphic, in the sense of the syntactic abstractions for corresponding constructs being identical. We simply make use of the wrapping and unwrapping with WrapP in the relevant places, as shown at the bottom of Fig. 5.17. This is possible since in $\mathcal{PF}$ type parameters are polarity-polymorphic, so we can use WrapP for both negative and positive types. Changing the data fragment embedding leads to reduction being interspersed by "unnecessary" reduction steps that take care of wrapping and unwrapping, but that does not significantly affect the close correspondence between surface reduction and embedded $\mathcal{PF}$ reduction.

Since the embeddings are isomorphic, transposition of the surface programs is compatible with that for $\mathcal{PF}$ via the embedding as far as the terms are concerned. As one may have expected, the crucial difference lies in the translation of function definitions: The continuation parameter added in the translation goes to consumer function signatures for the data fragment, and to destructor signatures for the codata fragment. But in either translation, the relevant variable $\bar{k}$ is available in each constructor/destructor case, either as input of the consumer function or through the destructor pattern match, thus translation of the bodies of these cases (which will be re-arranged when transposing), which are constituents of the function definitions, is not affected either.

Fig. 5.18 shows an example data fragment program and the result of transposing it to a codata fragment program (on the left) and their respective embeddings (on the right). In the example, each embedding is likewise the result of transposing the respective other one, and this compatibility of the program transposition also holds

in the general case. For the following we lift the embedding $\mathcal{M}$ to programs and refer to two linearized $\mathcal{PF}$ programs as being related by transposition, written with infix $\leftrightarrow_{\mathcal{PF}}$, when they are linearizations of the same matrix program, i.e. *up to the linearization choice they are identical*. This transposition is of course the same kind of reorganization as for the data and codata fragments, and the following theorem directly relates the two transpositions.

Theorem 5.3 (Transposition compatibility). *For all programs $\mathcal{P}_d \in$ DATA, $\mathcal{P}_c \in$ CODATA: If $\mathcal{P}_d \leftrightarrow_{\text{CODATA}} \mathcal{P}_c$, then $\mathcal{M}(\mathcal{P}_d) \leftrightarrow_{\mathcal{PF}} \mathcal{M}(\mathcal{P}_c)$.*

This is a direct consequence of how we translate function definitions, namely in a structure-preserving way, and that the syntactic abstraction for some data fragment construct is always identical to that of its structurally identical codata fragment counterpart.

With this theorem we can see that we have *decomposed* surface transposition into the macro embedding and $\mathcal{PF}$ transposition. This has the benefit that we are not required anymore to take surface transposition itself into consideration. For instance, as explained in Section 2.4.1, proofs involving the surface transposition were often technically tedious due to the structural differences between data and codata types, despite being quite obvious when intuitively thinking about them. On the other hand, transposition of $\mathcal{PF}$ programs is really just that, matrix transposition with positive entities becoming negative entities or vice versa but not needing any further plumbing to deal with structural differences. These are simply gone: positive and negative data cannot be distinguished structurally.

Most importantly, with our decomposition with the structurally identical term embeddings as well as the structurally positive and negative data, we have clearly carved out the only structural difference between the data and the codata fragments, namely the organization into function definitions.

5.2.3 Codata's greater degree of macro freedom

It is quite apparent that the syntactic abstractions for the codata fragment just presented are rather verbose. If we do not care about transposition compatibility, we can give a simpler embedding, shown in Fig. 5.19. Specifically, we do not actually need `Shift` anymore, since a negative producer of *any* type is an abstraction which brings the consumer into scope. This also means that our preprocessing step in which we classified terms as computations terms and value terms actually becomes unnecessary. We can thus simplify not only the embedding but also the language fragment specification, merging computation and value terms such that, in particular, producer function calls may now take computation terms as arguments. We indicate that the simpler embedding also applies to this simpler form of the codata fragment specification, which does not distinguish computation and value terms, by writing the arguments of syntactic abstractions with a generic t (instead of t_c and t_v where necessary). Fig. 5.20 shows how the addition example, written according to the simpler codata fragment specification, is embedded using the simpler embedding. Especially, `s` can directly take as an argument the call to `add`, thus the `suc` destructor is now actually not needed anymore.

Now, in a certain sense this technical complication of having two codata fragment embeddings, one compatible with transposition but not as simple as conceivable, and vice versa for the other, might be regarded as the consequence of a conceptual difference between the data and the codata fragments that presents itself

$$A_{f(\cdot,\ldots,\cdot)}(\overrightarrow{t}) = f(\overrightarrow{t}) \qquad A_{x_i}() = x_i \qquad A_{y_i}() = y_i$$

$$A_{d(\cdot,\cdot,\ldots,\cdot)}(t', \overrightarrow{t}) = \mu\{\bar{k} \Rightarrow t' \gg d(\overrightarrow{t}, \bar{k})\}$$

FIGURE 5.19: Simpler syntactic abstractions for the codata fragment.

$\textbf{function}_\ \mathtt{z}\ := \{$ $\quad \mathtt{add}(y_1) \quad \Rightarrow y_1$ $\quad \mathtt{suc} \quad \Rightarrow \mathtt{s}(\mathtt{z})$ $\}$	$\textbf{function}_\ \mathtt{z}\ := \{$ $\quad \mathtt{add}(y_1,\bar{k}) \quad \Rightarrow y_1 \gg \bar{k}$ $\quad \mathtt{suc}(\bar{k}) \quad \Rightarrow \mathtt{s}(\mathtt{z}) \gg \bar{k}$ $\}$
$\textbf{function}_\ \mathtt{s}(x_1)\ := \{$ $\quad \mathtt{add}(y_1) \quad \Rightarrow$ $\quad\quad \mathtt{s}(\mathtt{add}(x_1,y_1))$ $\quad \mathtt{suc} \quad \Rightarrow \mathtt{s}(\mathtt{s}(x_1))$ $\}$	$\textbf{function}_\ \mathtt{s}(x_1)\ := \{$ $\quad \mathtt{add}(y_1,\bar{k}) \quad \Rightarrow$ $\quad\quad \mathtt{s}(\mu\{\bar{k} \Rightarrow x_1 \gg \mathtt{add}(y_1,\bar{k})\})$ $\quad \mathtt{suc}(\bar{k}) \quad \Rightarrow \mathtt{s}(\mathtt{s}(x_1)) \gg \bar{k}$ $\}$

FIGURE 5.20: Codata addition example from Fig. 5.18 (left; simplified version of the codata fragment specification), and its simpler embedding (right).

when viewing these through the lens of $\mathcal{PF}$. Specifically, the data fragment inherently restricts the reasonable macros that one may specify for it, and more so than the codata fragment. This subsection makes formally precise the property that a surface language (fragment) requires the distinction between computation and value terms at the $\mathcal{PF}$ type level in order for *any* macro embedding to be reasonably compatible with surface operational semantics and typing; it then shows that the data fragment exhibits this property while the codata fragment does not. The subsection that follows adds transformations between the simple and the verbose form of the codata abstractions to our toolbox for the practical programmer; this involves creating/removing destructors that correspond to constructors lifted to functions, like suc in the running example (addition).

For the purpose of the following definitions, we consider a *surface language* to have some term syntax, a deterministic reduction relation, which we denote $\triangleright_s$, and distinguished value terms t_v and values v, as in the languages considered previously in this chapter. As before, *computation terms* are the terms that the reduction relation $\triangleright_s$ is defined for.

Definition 5.3 (Value (semantic) compatibility)**.** *A macro embedding $\mathcal{M}$ for some surface language is* value compatible *iff:*

- *there is some function $\mathcal{F}$ such that: for all computation terms t and values v in the surface language with $t \triangleright_s^* v$, it is $\mathcal{F}(\mathcal{M}(t)) \triangleright^* \textbf{Result}(\mathcal{M}(v))$, and*
- *$\mathcal{M}(t_v^1) \neq \mathcal{M}(t_v^2)$ for any two value terms $t_v^1 \neq t_v^2$.*

The data fragment embedding shown above is value compatible, with

$$\mathcal{F}(x) := x \gg \mathtt{sh}(\bar{\mu}\{v \Rightarrow \textbf{Result}(\mathcal{M}(v))\})\},$$

which is a valid definition since $\mathcal{M}$ translates *all* computation terms to producers of the negative type `Shift`. (The use of the **Result** daimon is of course entirely inessential for the definition above and purely for illustrative purposes.)

Definition 5.4 (Value type contiguity)**.** *A macro embedding $\mathcal{M}$ for some surface language is* value type contiguous *iff*

- *for any type T of the surface language, the signatures declared for $\mathcal{M}(T)$ are structurally the same as those for T,*
- *for any value v and type T the judgment $\vdash v : T$ implies $\vdash \mathcal{M}(v) : \mathcal{M}(T)$, and*
- *there is a type with a hole T_c s.t. for any computation term t_c and type T the judgment $\vdash t_c : T$ implies $\vdash \mathcal{M}(t_c) : T_c[\mathcal{M}(T)]$.*

Definition 5.5 (Value type precision)**.** *A value type contiguous macro embedding $\mathcal{M}$ for some surface language has the property of* value type precision *iff $T_c \neq []$, where T_c is the holed type given by the definition of contiguity.*

The data fragment embedding we saw above exhibits value type precision, since the constructor signatures and values are taken over unchanged, and computation terms are typed $\texttt{Shift}\langle T\rangle$, thus $T_c = \texttt{Shift}\langle[]\rangle$. In fact, this property is necessarily exhibited by all data fragment embeddings we are interested in:

Definition 5.6 (Enforcing macro property)**.** *A surface language* enforces *a property iff it holds for all possible value compatible and value type contiguous macro embeddings into $\mathcal{PF}$.*

Theorem 5.4. *The data fragment enforces value type precision.*

Proof. Let $\mathcal{M}$ be any value compatible and value type contiguous macro translation for the data fragment. Assume that $\mathcal{M}$ does not exhibit value type precision; we show that this leads to a contradiction. The argument boils down to the fact that computation terms cannot be macro translated to values in $\mathcal{PF}$ without breaking (any reasonable) correspondence between surface and $\mathcal{PF}$ semantics through the macro translation (as abstractly expressed by value compatibility).

Consider the surface term $t := \texttt{add}(\texttt{z},\texttt{z})$, which types $\vdash t : \texttt{Nat}$. Since $\mathcal{M}$ does not exhibit value type precision, this means that the type T' of $\mathcal{M}(t)$ is equal to $\mathcal{M}(\texttt{Nat})$ (i.e. $T_c = []$), and thus due to value type contiguity T' is structurally identical to `Nat`, so $\mathcal{M}(t)$ is a value of type `Nat` in $\mathcal{PF}$. Because $\mathcal{M}$ is a macro translation, we know that $\mathcal{M}(t) = A(\mathcal{M}(\texttt{z}), \mathcal{M}(\texttt{z}))$ for some syntactic abstraction A. Since $\mathcal{M}$ is value compatible, and $t \rhd_s^* \texttt{z}$, we know that $\mathcal{F}(\mathcal{M}(t)) \rhd^* \textbf{Result}(\mathcal{M}(\texttt{z}))$. We also know that the statements in this paragraph hold analogously for all other surface computation terms in the place of t, e.g., $t' := \texttt{add}(\texttt{z},\texttt{s}(\texttt{z}))$; in particular, the previous sentence uses the *same* $\mathcal{F}$.

In summary, we have:

$$\mathcal{F}(A(\mathcal{M}(\texttt{z}),\mathcal{M}(\texttt{z}))) \rhd^* \textbf{Result}(\mathcal{M}(\texttt{z})),$$

and

$$\mathcal{F}(A(\mathcal{M}(\texttt{z})),\mathcal{M}(\texttt{s}(\texttt{z}))) \rhd^* \textbf{Result}(\mathcal{M}(\texttt{s}(\texttt{z}))),$$

and A applied to any two arguments is a value of type `Nat` in $\mathcal{PF}$. Because of the latter property, $A(\alpha_1,\alpha_2) = \texttt{s}^i(\texttt{z})$, $i \geq 0$ (1), $A(\alpha_1,\alpha_2) = \texttt{s}^i(\alpha_1)$, $i \geq 0$ (2), and $A(\alpha_1,\alpha_2) = \texttt{s}^i(\alpha_2)$, $i \geq 0$ (3), are the only possible cases for the form of A.

In cases (1) and (2), since (at least) the second argument is ignored, the two inputs to $\mathcal{F}$ in the reduction sequences above are identical, i.e. there is some v such that

$\mathcal{F}(v) \triangleright^* \mathbf{Result}(\mathcal{M}(\mathtt{z}))$ and $\mathcal{F}(v) \triangleright^* \mathbf{Result}(\mathcal{M}(\mathtt{s}(\mathtt{z})))$. Because reduction in $\mathcal{PF}$ is deterministic, this can only be the case if $\mathcal{M}(\mathtt{z}) = \mathcal{M}(\mathtt{s}(\mathtt{z}))$, but this is ruled out by value compatibility.

For case (3), simply flip the arguments to `add` in t' to arrive at the same contradiction.

We thus have a contradiction to our precondition that $\mathcal{M}$ is compatible. Therefore our assumption that $\mathcal{M}$ does not exhibit value type precision must have been wrong, and so $\mathcal{M}$ does exhibit value type precision. We let $\mathcal{M}$ be entirely generic, thus all macro translations for the data fragment exhibit value type precision, which by definition means that the data fragment enforces value type precision. □

The practical implication of this is that we cannot give a reasonable, i.e., value type contiguous and compatible, macro embedding for the data fragment in which computation and value types are not distinguished at the type level in $\mathcal{PF}$. In this regard, the codata fragment differs from the data fragment.

Theorem 5.5. *The codata fragment does not enforce value type precision.*

Proof. The simple version of the embedding for the codata fragment does not exhibit value type precision, since all computation terms typed T are typed $\mathcal{M}(T)$ after the translation.[12] □

In summary, $\mathcal{PF}$ has helped characterize a conceptual difference between the data and codata fragments that was not directly apparent beforehand. The data fragment enforces value type precision, while the codata fragment does not. This greater degree of freedom offers an initial explanation for why it might be inevitably necessary to consider two macro embeddings for the codata fragment and convert between them. However, this is still only an early result (in an exploratory theory development) and a possible influence on language design may need to be explored more.

5.2.4 Practical surface transposition

We saw that the greater freedom of the codata fragment when it comes to reasonable macro embeddings into $\mathcal{PF}$ arguably makes it necessary to convert between the result of the simpler embedding and that of the embedding amenable to surface transposition, that however forces upon us many additional destructors, in the worst case one per producer function. We view these transformations as pieces of the overall transformation process that has the data fragment, in the presentation with constructors directly applied to computation terms, on one end and the simple form of the codata fragment on the other end. We now briefly describe each bidirectional step; the running example used before (addition) illustrates each step of the process, as shown in Fig. 5.21.

Lift constructors As noted at the beginning of the previous section, first create a surface-level consumer function for each constructor that simply applies the constructor to the arguments of the function. (It suffices to do this for all constructors that are called with computation terms.) Then replace each constructor call on (at

[12]It can be readily checked that the simple codata fragment embedding is value compatible and value type contiguous (as is the verbose embedding); for instance, a destructor signature $d(T, \overrightarrow{T}) : T'$ becomes the structurally isomorphic $d(T, \overrightarrow{T}, \overline{T'})$ (merely turning the surface output type into a $\mathcal{PF}$ continuation type, but not touching the structural three-part form).

$\textbf{function}_{+}\ \texttt{add}(y_1)\ := \{\ \texttt{z}\ \Rightarrow\ y_1 \mid \texttt{s}(x_1)\ \Rightarrow\ \texttt{s}(\texttt{add}(x_1, y_1)\ \}$

Step 1: Lift constructors

$\textbf{function}_{+}\ \texttt{add}(y_1)\ := \{\ \texttt{z}\ \Rightarrow\ y_1 \mid \texttt{s}(x_1)\ \Rightarrow\ \texttt{suc}(\texttt{add}(x_1, y_1)\ \}$

$\textbf{function}_{+}\ \texttt{suc}\ := \{\ n\ \Rightarrow\ \texttt{s}(n)\ \}$

Step 2: Embed data fragment

$\textbf{function}_{+}\ \texttt{add}(y_1, \bar{k})\ :=$

$\{\texttt{z} \Rightarrow y_1 \gg \bar{k} \mid \texttt{s}(x_1) \Rightarrow \mu\{\texttt{sh}(\bar{k}) \Rightarrow x_1 \gg \texttt{add}(y_1, \texttt{wrap}(\bar{k}))\} \gg \texttt{sh}(\texttt{posk}(\texttt{suc}(\bar{k})))\ \}$

$\textbf{function}_{+}\ \texttt{suc}(\bar{k})\ := \{\ n\ \Rightarrow\ \texttt{s}(n) \gg \bar{k}\ \}$

Step 3: Transpose

$\textbf{function}_{-}\ \texttt{z}\ := \{\ \texttt{add}(y_1, \bar{k}) \Rightarrow y_1 \gg \bar{k} \mid \texttt{suc}(\bar{k}) \Rightarrow \texttt{s}(\texttt{z}) \gg \bar{k}\ \}$

$\textbf{function}_{-}\ \texttt{s}(x_1)\ :=$

$\{\ \texttt{add}(y_1, \bar{k}) \Rightarrow \mu\{\texttt{sh}(\bar{k}) \Rightarrow x_1 \gg \texttt{add}(y_1, \texttt{wrap}(\bar{k}))\} \gg \texttt{sh}(\texttt{posk}(\texttt{suc}(\bar{k})))$

$\mid \texttt{suc}(\bar{k}) \Rightarrow \texttt{s}(\texttt{s}(x_1)) \gg \bar{k}\ \}$

Step 4: Eliminate destructors

$\textbf{function}_{-}\ \texttt{z}\ := \{\ \texttt{add}(y_1, \bar{k}) \Rightarrow y_1 \gg \bar{k}\ \}$

$\textbf{function}_{-}\ \texttt{s}(x_1)\ := \{\ \texttt{add}(y_1, \bar{k}) \Rightarrow \texttt{s}(\mu\{\bar{k} \Rightarrow x_1 \gg \texttt{add}(y_1, \bar{k})\})\ \}$

Step 5: Un-embed codata fragment

$\textbf{function}_{-}\ \texttt{z}\ := \{\ \texttt{add}(y_1) \Rightarrow y_1\ \}$

$\textbf{function}_{-}\ \texttt{s}(x_1)\ := \{\ \texttt{add}(y_1) \Rightarrow \texttt{s}(x_1.\texttt{add}(y_1))\ \}$

FIGURE 5.21: From the data to the codata fragment by example.

least one) computations term(s) by calls to the respective function. By annotating the functions with the name of their corresponding constructor, this step can be easily undone; also, annotate the relevant constructors themselves with the information that they underwent this process.

Embed data fragment Macro-embed the constructs of the data fragment by the modified syntactic abstractions of Fig. 5.17 (bottom). Annotate each resulting μ-abstraction with the syntactic abstraction it was created from, and the process is readily invertible.

Transpose The central but at the same time easiest step: Transpose the underlying matrix.

Eliminate destructors All the destructors which resulted from consumer functions that were created in the first step are annotated with the corresponding constructor name. Replace each μ-abstraction *not* annotated to correspond to such a destructor call with the result of the simpler syntactic abstraction from Fig. 5.19:

$$A_{d(\cdot,\cdot,\ldots,\cdot)}(t', \overrightarrow{t}) = \mu\{\bar{k} \Rightarrow t' \gg d(\overrightarrow{t}, \bar{k})\}$$

Replace the μ-abstractions indeed corresponding to the to-be-eliminated destructors using the syntactic abstraction $A_{f(\cdot,\ldots,\cdot)}(\overrightarrow{t}) = f(\overrightarrow{t})$, then remove all these destructors and their cases in producer function definitions. Replacing the results of syntactic abstractions works likewise for the other direction. The functions corresponding to the relevant constructors are still annotated as such, so we know which ones to consider.

Un-embed codata fragment (simple embedding) Use the syntactic abstractions from Fig. 5.17 (top). Like the embedding for the data fragment, this step is easily made bidirectional by annotating the resulting μ-abstractions (and function calls just stay function calls anyway).

5.2.5 GA(Co)DT example: interpreter transposition

To conclude this section, we take a look at a somewhat more difficult example involving parametric polymorphism. It is the meta-circular interpreter with a user-defined generic function type, which, when transposed, "invents" closures for us. We gave a direct $\mathcal{PF}$ implementation in Section 4.4.1 of the previous chapter (in Fig. 4.18). Now we will see how we can write the interpreters in the GA(Co)DT matrix language discussed at the end of the first chapter, and embed them into $\mathcal{PF}$ and use transposition in $\mathcal{PF}$ to go from one to the other. We will do this in the steps described in the previous subsection. Type parameters simply carry over from the surface to $\mathcal{PF}$ but do not otherwise affect the embeddings.

We start with the meta-circular interpreter shown in Fig. 5.22, a GACoDT program which is one of the two linearized forms of a matrix program in the matrix language; the other linearized form, a GADT program, will be the end result of our steps. For the first step, lifting of constructors and introduction of additional destructors, we do not need to do anything. The only constructors applied are `cons` and `in` and the parameters of the constructor applications are value terms (variables and a producer function applied to variables, respectively), and the same goes for

```
codata Fun⟨∗,∗⟩ { Fun⟨A,B⟩.apply⟨A,B⟩(A) : B }
data Val { in(Fun⟨Val,Val⟩) }
data ValList { nil | cons(Val,ValList) }
data Exp { var(Nat) | app(Exp,Exp) | abs(Exp) }
data Nat { zero | succ(Nat) }

function− closure(e,l) := {
    apply(v)                 ⇒ eval(e,cons(v,l))
}

function+ apply_aux(v) := { in(f) ⇒ f.apply(v) }
function+ eval(l) := {
    var(n)                   ⇒ lookup(n,l)
    app(e1,e2)               ⇒ apply_aux(eval(e1,l),eval(e2,l))
    abs(e)                   ⇒ in(closure(e,l))
}
```

FIGURE 5.22: Meta-circular interpreter with user-defined generic function type as one of the linearized forms of a GA(Co)DT matrix program.

```
data− Fun⟨∗,∗⟩ { apply⟨A,B⟩(A,B̄) : Fun⟨A,B⟩ }
data+ Val { in(Fun⟨Val,Val⟩) }
data+ ValList { nil | cons(Val,ValList) }
data+ Exp { var(Nat) | app(Exp,Exp) | abs(Exp) }
data+ Nat { zero | succ(Nat) }

function− closure(e,l) : Fun⟨Val,Val⟩ := {
    apply(v,k̄)           ⇒ e ≫ eval(cons(v,l),k̄)
}

function+ apply_aux(v,k̄) := { in(f) ⇒ f ≫ apply(v,k̄) }
function+ eval(l,k̄) := {
    var(n)               ⇒ n ≫ lookup(l,k̄)
    app(e1,e2)           ⇒
      μ{sh(k̄) ⇒ e1 ≫ sh(posk(eval(l,wrap(k̄))))} ≫
       sh(μ̄{pos(v1) ⇒ μ{sh(k̄) ⇒ e2 ≫ sh(posk(eval(l,wrap(k̄))))} ≫
        sh(μ̄{pos(v2) ⇒ v1 ≫ apply_aux(v2,k̄)})})
    abs(e)               ⇒ in(closure(e,l)) ≫ k̄
}
```

FIGURE 5.23: Result of embedding the program in Fig. 5.22 into $\mathcal{PF}$.

```
data₊ Fun⟨*,*⟩ { closure(e,l) : Fun⟨Val,Val⟩ }
data₊ Val { in(Fun⟨Val,Val⟩) }
... /* other data types unchanged */

function₊ apply⟨A,B⟩(v,k̄) : Fun⟨A,B⟩ := {
    closure(e,l)              ⇒ e ≫ eval(cons(v,l),k̄)
}

function₊ apply_aux(v,k̄)  := {... /* unchanged */ }
function₊ eval(l,k̄)  := {... /* unchanged */ }
```

FIGURE 5.24: Result of transposing the program in Fig. 5.23 w.r.t. Fun.

```
data Fun⟨*,*⟩ { closure(e,l) : Fun⟨Val,Val⟩ }
data Val { in(Fun⟨Val,Val⟩) }
... /* other data types same as in original program */

function₊ apply⟨A,B⟩(v) : Fun⟨A,B⟩ := {
    closure(e,l)          ⇒ eval(e,cons(v,l))
}

function₊ apply_aux(v)  := { ... /* same as in original program */ }
function₊ eval(l,k̄)  := { ... /* same as in original program */ }
```

FIGURE 5.25: Un-embedding the program in Fig. 5.24.

the only producer function applied, `closure`. The second step, embedding into $\mathcal{PF}$ using the more verbose macros amenable to transposition, results in the program shown in Fig. 5.23.

Transposing this program with respect to type `Fun` results in the program shown in Fig. 5.24. Un-embedding this program results in the GADT program shown in Fig. 5.25. As intended, this is the same as one would have obtained by surface-transposing the initial GACoDT program with respect to `Fun`.

There are a few things to remark about this process and possible further processing of the result. First, for practical (e.g. performance) reasons, it might be desirable to carry out the embedding in a *lazy* fashion. That is, a term is only really replaced by the result of macro-embedding it if some step that works on the $\mathcal{PF}$ level actually needs to inspect it somehow. In particular, in the transposition step, if the term is not involved in the underlying matrix, i.e. by being in a relevant function body or containing a relevant local abstraction that is to be transposed, the embedding can be delayed. Later, the un-embed step is then simplified by just keeping all not-yet-embedded terms as they are.

Second, as in Section 4.4.1, there is only one constructor for `Fun`, and this constructor specifies the type parameters to be `Val`, and `Val`) is only there to tie a recursive knot for `Fun`. Thus, *safe monomorphization* and then *recursive type elimination*, as described in Section 4.4.1, can be applied to the transposition result to simplify it (doing so in $\mathcal{PF}$ is directly reflected in the types of the surface language since the

type parameter mechanism is the same for both). Again, this allows for the removal of auxiliary wrapper functions, in this case `apply_aux`, which may be facilitated by a linguistic construct that makes it explicit that some type (like `Val`) has exactly the recursive knot-tying purpose (this linguistic enhancement applies in the same way to $\mathcal{PF}$ and the surface language).

5.3 Control Effects

We have seen how to recover declarative surface constructs like function application in $\mathcal{PF}$ at "shifted type", expanding upon ideas of Zeilberger (2008b). In the result of embedding such constructs, continuations were of course always used linearly. In this section we turn to constructs manipulating control flow, with probably the most famous (or infamous) one being call/cc. Zeilberger (2008b) gives a syntactic abstraction for its restricted variant let/cc[13] that we can easily translate to $\mathcal{PF}$:

$$A_{(\text{let/cc } k.\cdot)}(t[\bar{k}]) = \mu\{\mathtt{sh}(\bar{k}) \Rightarrow t[\bar{k}] \gg \mathtt{sh}(\bar{k})\}$$

The idea is that the "return" continuation bound to k is now available for use earlier in the control flow; using it somewhere within t "jumps" out of the computation and passes control flow to the context surrounding the let/cc. We now consider enhancing first-order and higher-order languages with let/cc and full call/cc based on this syntactic abstraction. Regarding *delimited* continuation, $\mathcal{PF}$ is arguably not as well suited to emulate those, at least in its current form; Chapter 6 contains some preliminary thoughts on how to recover delimited continuations in an extension of $\mathcal{PF}$ or a system related to $\mathcal{PF}$ instantiated from a common meta system.

5.3.1 First-order CBV language enhanced with let/cc

To present a simple cohesive example, we take the simple first-order CBV language presented at the beginning of this chapter and enhance this language with let/cc, embedded using the syntactic abstraction just given. Formally, we enhance the language by adding the following reduction rule:

$$\mathcal{E}[(\text{let/cc } k.t[k])] \;\rhd_{\text{v}}\; \mathcal{E}[t[\text{cont}(\mathcal{E})]]$$

For this, we introduce a new class of variables, *continuation variables* k, bound by let/cc (there may be nested let/cc constructs, in which case we distinguish the different k by a subscript). Continuation variables are one of two kinds of *continuation terms* t_k, a new syntactic category we introduce:

$$t_k \;::=\; k \mid \text{cont}(\mathcal{E})$$

The other kind of continuation term $\text{cont}(\mathcal{E})$ lifts evaluation contexts to terms; as shown above, reducing a let/cc results in the relevant k being substituted with the evaluation context surrounding the let/cc. We add the let/cc construct itself, and a new construct that allows to call continuation terms, to our computation terms, plus we enhance evaluation contexts with a variant for these calls:

$$t_c \;::=\; ... \mid \text{call}(t_k, t) \mid (\text{let/cc } k.t_c)$$

$$\mathcal{E} \;::=\; ... \mid \text{call}(\text{cont}(\mathcal{E}), \mathcal{E})$$

Note that, for our simple language, we restrict the "first-class-ness" of continuations such that they may only be referred to within these call constructs; they cannot be passed around freely. We have the expected reduction rule for such calls that throws

[13] Zeilberger (2008b) calls the construct "callcc", but the actual syntactic abstraction he gives is for what is known as "let/cc" in, e.g., Scheme.

Continuation terms:
$A_{\mathsf{cont}(\mathcal{E})}() = \mathcal{T}(\mathcal{E}) \quad A_k() = \bar{k}$

Computation terms enhancement:
$A_{\mathsf{call}(\cdot,\cdot)}(t_k, t_c) = \mu\{_ \Rightarrow t_c \gg \mathtt{sh}(t_k)\}$
$A_{\mathsf{call}(\cdot,\cdot)}(t_k, t_v) = \mu\{_ \Rightarrow t_v \gg t_k\}$

FIGURE 5.26: Syntactic abstractions for continuation constructs.

`try_inv_sq`	$\mapsto$ (let/cc $k.\mathtt{sq}@(\mathtt{if_zero}@(x, \mathsf{call}(k, y), \mathtt{div}@(100, x))))$
`if_zero`	$\mapsto (y_1, y_2)$ (call-by-name)

FIGURE 5.27: Let/cc example in the enhanced first-order CBV language.

away the surrounding evaluation context in favor of that contained in the call:

$$\mathcal{E}'[\mathsf{call}(\mathsf{cont}(\mathcal{E}), v)] \ \rhd_{\mathrm{v}} \ \mathcal{E}[\mathit{return}(v)]$$

The syntactic abstractions for continuation terms and for call are shown in Fig. 5.26, where we use the context translation $\mathcal{T}$ from Fig. 5.5 for the cont construct. While embedded value terms have type `Nat` and embedded computation terms have type `Shift`$\langle$`Nat`$\rangle$ as before, continuation terms have type $\overline{\mathtt{Nat}}$. Finally, we additionally allow functions with two non-matched parameters, accessed by variables y_1, y_2, and straightforwardly extend the relevant macro embeddings.

In this enhanced language we can give our first let/cc example, shown in Fig. 5.27 (we omit the definitions of the squaring function `sq` and the division (followed by rounding) function `div`). The function `try_inv_sq` is given two inputs x and y and tries to compute $\lfloor 100/x \rfloor^2$, producing the default value y if division would fail because x is zero.[14] If division would fail, the result is *immediately* y, i.e. without squaring it. In other words, we "escape" the computation, discarding the evaluation context $\mathtt{sq}([])$, and jump back to before we started (which is exactly the mechanism that let/cc provides in Scheme).

In Fig. 5.28, showing reduction steps on the surface and in $\mathcal{PF}$ side-by-side, we see that our macro embedding is compatible with $\mathcal{PF}$ reduction for our example. As explained in Section 5.1.2, there is a one-to-one correspondence between machine steps of configurations of an evaluation machine for the surface language and reduction steps in $\mathcal{PF}$. Machine configurations, which consist of a context and a term, are translated by translating the context, obtaining a continuation k where the input variable corresponds to the hole, and the term, then stripping off the `Shift` abstraction from it and substituting k for the variable freed by this. When considering $\mathcal{E}_0[\mathtt{sq}([])][\mathsf{call}(\mathcal{E}_0, \mathcal{E}_2)]$, taking this particular presentation of the term as indicative of where the evaluation focus lies, we would translate the configuration $(\mathcal{E}_0[\mathtt{sq}([])], \mathsf{call}(\mathcal{E}_0, \mathcal{E}_2))$. In particular, since $\mathsf{call}(\mathcal{E}_0, 5)$ is translated to a μ-abstraction which ignores its input argument, i.e. its form is $\mu\{_ \Rightarrow C\}$ for some command C, the context is just ignored here, i.e. the result of translating the configuration is

[14] Checking for zero and branching accordingly requires (in this example) that only the relevant branch is evaluated; this can be implemented by the call-by-name function `if_zero` shown in Fig. 5.27.

$$\begin{array}{ll|ll}
 & \mathcal{E}_0[\mathtt{try_inv_sq}(0,5)] & & 0 \gg \mathtt{try_inv_sq}(5,k_0) \\
\triangleright_v & \mathcal{E}_0[\mathtt{sq}(\underbrace{\mathtt{if_zero}(0,\mathrm{call}(\mathcal{E}_0,5),\mathtt{div}(100,0))}_{t'})] & \triangleright & \mathtt{sq}(t')[k_0] \gg \mathtt{sh}(k_0) \\
 & & \triangleright^2 & t'[k_0] \gg \mathtt{sh}(\mathtt{sq}(k_0)) \\
\triangleright_v & \mathcal{E}_0[\mathtt{sq}(\mathrm{call}(\mathcal{E}_0,5))] & \triangleright^2 & 5 \gg k_0 \\
\triangleright_v & \mathcal{E}_0[\mathit{return}(5)] & \triangleright & \mathbf{Result}(5) \quad \text{(e.g.)}
\end{array}$$

FIGURE 5.28: Example reduction and embedded reduction in $\mathcal{PF}$ for the example in Fig. 5.27 (e.g. $\mathcal{E}_0 := []$, $k_0 := \tilde{\mu}\{n \Rightarrow \mathbf{Result}(n)\}$).

just C, with no need to substitute in a continuation. This reflects that calling a continuation passed by let/cc discards the current evaluation context. There is also a general correspondence between surface and $\mathcal{PF}$ reduction mediated by the embedding which can be demonstrated by extending the results from the first section of this chapter; details can be found in Appendix C.

5.3.2 A higher-order language with let/cc

We now extend the language with (call-by-value) first-class functions (and with lists of numbers, to facilitate a slightly more interesting example). We add functional (λ-) abstractions and applications (written as a pair of function and argument term with a space ␣in between) to computation terms:

$$t_c \ ::= \ \ldots \mid \lambda n.t_c \mid t_c \sqcup t_c$$

For simplicity we only consider first-class functions from Nat to Nat. We add a new kind of variable n to the value terms which is used for the formal parameter of functional abstractions; in nested λ-abstractions a subscript i, i.e., in n_i, is used to distinguish binding sites. The types of non-matched parameters of first-order functions may now be either Nat, lists of Nat or the function type; the result of a first-order function may only be a number or a list of numbers.[15] Note that in this simplified setting, first-order function parameter variables y_i and lambda abstractions are the only possible terms of function type. Finally, we have the usual contraction rule for a meeting of lambda abstraction and application, as well as the extension of evaluation contexts with contexts for application terms and that of values with lambda abstractions.

$$\begin{array}{l}
((\lambda n.t) \sqcup v) \ \blacktriangleright_v \ t[n \mapsto v] \\
\mathcal{E} \ ::= \ \ldots \mid (\mathcal{E} \sqcup t) \mid (v \sqcup \mathcal{E}) \\
v \ ::= \ \ldots \mid (\lambda n.t)
\end{array}$$

Note that lambda abstractions are not allowed as the result of a first-order function, due to the restriction of results of such functions to numbers.

Since first-class functions are just a special kind of codata, the embedding follows that of the codata fragment, with the function type being encoded as a negative data type with one destructor $\mathtt{apply}(\mathtt{Nat},\overline{\mathtt{Nat}})$. The constructs added to the computation terms are thus embedded as follows:

$$\begin{array}{lcl}
A_{(\lambda n.\cdot)}(t) & = & \mu\{\mathtt{apply}(n,\bar{k}) \Rightarrow t \gg \mathtt{sh}(\bar{k})\} \\
A_{(\cdot \sqcup \cdot)}(t_1,t_2) & = & \mu\{\mathtt{sh}(\bar{k}) \Rightarrow t_2 \gg \mathtt{sh}(\tilde{\mu}\{a \Rightarrow t_1 \gg \mathtt{apply}(a,\bar{k})\})\}
\end{array}$$

[15] The latter are specified with the usual data type with a nil constructor and a cons constructor (written infix ::), and abbreviated with the usual notation, e.g. [4,5,2].

$$\begin{array}{ll}
\texttt{map} & \mapsto (\mathit{return}(\texttt{nil}), (y_1 \mathbin{\llcorner\!\lrcorner} x_1)\texttt{::}(\texttt{map@}(x_2, y_1))) \\
\texttt{ign_fst} & \mapsto (\mathit{return}(y)) \\
\texttt{if_zero} & \mapsto (y_1, y_2) \quad \text{(call-by-name)} \\
\\
\texttt{search} & \mapsto (\text{let/cc}\ k.\ \texttt{ign_fst@}(\texttt{map@}(x, \lambda n.\texttt{if_zero@}(n, \text{call}(k, n), n)), y))
\end{array}$$

FIGURE 5.29: Let/cc example with CBV first-class functions.

As an example[16], consider the first-order function `search`, shown in Fig. 5.29, which makes use of a first-class function (and of the higher-order function `map`, defined as usually). The `search` function is given a list and checks whether some element is zero and if yes, returns the first such element.[17] In summary, to do so, it emulates a loop using `map` and a return statement, as known from C-like languages (Java, C, C++), using let/cc. More precisely, it uses let/cc as its topmost syntactic node and, within the body of the let/cc, maps a first-class function over the list which checks if the given element is zero or not. When the element is zero, the continuation captured by the let/cc is called with the element to return that element as the result of `search`. Otherwise, the first-class function simply returns the element unchanged. If for no element the captured continuation is called (since no element is zero), a default value passed as an additional argument to `search` is returned; this is implemented with a call to an auxiliary function `ign_fst` which ignores one of its arguments (in this case, the result of `map`) and returns the other (in this case, the default value).

Fig. 5.30 shows the reduction steps of a term using `search`, side by side with the reduction steps in $\mathcal{PF}$ of the respective macro translations of the terms in the reduction sequence. A general correspondence between reduction in the higher-order let/cc language and reduction of the embedded terms in $\mathcal{PF}$ can be demonstrated by straightforwardly extending the results from the above sections to take first-class functions into account; first-class function calls can here be treated similarly to first-order function calls.

5.3.3 Passing first-class functions to call/cc

The language extension with first-class functions above showed that the embedding into $\mathcal{PF}$ can be straightforwardly extended to combine let/cc and first-class functions, allowing for examples known from, e.g., Scheme, like the `search` function shown above which uses `map` to emulate a loop and let/cc to emulate a "return" statement as known from C-like languages. However, languages like Scheme even allow to pass a first-class function as a parameter to call/cc (or rather, a first-class function *must* be passed, since in Scheme call/cc is not a binder, just an operator that takes such an argument). This section modifies the language and the embedding to

[16] Adapted from the Community Scheme Wiki: `http://community.schemewiki.org/?call-with-current-continuation` (accessed Oct. 26, 2020).

[17] Since the result is always zero if a zero is found, "the *first* such element" is not actually a relevant property in this simplified example, but it is easy to imagine a more general search function with a boolean predicate (e.g., the first odd element of $[2, 5, 3, 6, 7]$ is 5).

$$
\begin{array}{ll|ll}
 & \mathcal{E}_0[\texttt{search}(\underbrace{[4,0,2]}_{l},7)] & & l \gg \texttt{search}(7,k_0) \\
\triangleright_v & \mathcal{E}_0[\texttt{ign_fst}(\underbrace{\texttt{map}(l,f)}_{t'},7)] & \triangleright & \texttt{ign_fst}(t',7)[k_0] \gg \texttt{sh}(k_0) \\
 & (f := \lambda n.\texttt{if_zero}(n,\mathrm{call}(\mathcal{E}_0,n),n)) & \triangleright^2 & t'[k_0] \gg \texttt{sh}(\texttt{ign_fst}(7,k_0)) \\
\triangleright_v & \mathcal{E}_0[\texttt{ign_fst}((f\ 4)\texttt{::}(\texttt{map}([0,2],f)),7)] & \triangleright^2 & f[k_0] \gg \texttt{apply}(4,\bar{\mu}\{a \Rightarrow ...\}) \\
\triangleright_v & \mathcal{E}_0[\texttt{ign_fst}(4\texttt{::}(\texttt{map}([0,2],f)),7)] & \triangleright^3 & 4 \gg \bar{\mu}\{a \Rightarrow \texttt{map}(...)[k_0] \gg ...\} \\
\triangleright_v & \mathcal{E}_0[\texttt{ign_fst}(4\texttt{::}((f\ 0)\texttt{::}(\texttt{map}([2],f))),7)] & \triangleright^3 & f[k_0] \gg \texttt{apply}(0,...) \\
\triangleright_v & \mathcal{E}_0[\texttt{ign_fst}(4\texttt{::}((\mathrm{call}(\mathcal{E}_0,0))\texttt{::}(\texttt{map}([2],f))),7)] & \triangleright^3 & 0 \gg k_0 \\
\triangleright_v & \mathcal{E}_0[\mathit{return}(0)] & \triangleright & \mathbf{Result}(0) \quad \text{(e.g.)}
\end{array}
$$

FIGURE 5.30: Example reduction and embedded reduction in $\mathcal{PF}$ for the example in Fig. 5.29 (e.g. $\mathcal{E}_0 := [], k_0 := \bar{\mu}\{n \Rightarrow \mathbf{Result}(n)\}$).

reflect this, replacing let/cc with call/cc (the former can of course be defined using the latter), and gives a slightly more involved example reminiscent of coroutines[18].

The surface reduction rule for call/cc looks like this, where f is some lambda abstraction:

$$\mathcal{E}[(\text{call/cc}\ f)] \ \triangleright_v\ \mathcal{E}[(f \llcorner \text{cont}(\mathcal{E}))]$$

That is to say, the first-class function f is invoked with a continuation which embeds the current context. Thus the only first-class functions allowed to be passed as an argument to call/cc are those which take a continuation term as their input; we have a new syntactic form for such function abstractions (which have continuation variables k as their formal parameter), as well as applications (function to continuation term), extending computation terms (call/cc itself is also added, replacing let/cc):

$$t_c \ ::= \ ... \mid (\text{call/cc}\ t) \mid (\lambda k.t_c) \mid (t_c \llcorner t_k)$$

The syntactic abstraction for the function abstraction and application for functions taking continuations are like those seen in the previous section, just with the input being of continuation type (like $\overline{\texttt{Nat}}$) and hence not needing to be evaluated.

$$
\begin{array}{lcl}
A_{(\lambda k.\cdot)}(t) & = & \mu\{\texttt{apply}(\bar{k},\bar{k}_1) \Rightarrow t \gg \texttt{sh}(\bar{k}_1)\} \\
A_{(\cdot \llcorner \cdot)}(t_1,t_2) & = & \mu\{\texttt{sh}(\bar{k}) \Rightarrow t_1 \gg \texttt{apply}(t_2,\bar{k})\}
\end{array}
$$

First-order functions are now also allowed to return continuation terms, not just value terms. Therefore, a computation term can reduce to a continuation term. In particular, call/cc may be invoked with some computation term that reduces to a continuation term, thus we also need a reduction rule for call/cc called with a continuation term that embeds a context.

$$\mathcal{E}[(\text{call/cc}\ \text{cont}(\mathcal{E}_0))] \ \triangleright_v\ \mathcal{E}[\text{call}(\text{cont}(\mathcal{E}_0),\text{cont}(\mathcal{E}))]$$

After such a reduction step, the next step will always be to throw away the outer evaluation context $\mathcal{E}$ as specified by the reduction rule for continuation calls. However, after this call, $\mathcal{E}$ is now still available as a continuation, substituted into the body of cont$(\mathcal{E}_0)$, so it is possible to return to it in the remaining computation.

In summary, call/cc can take either a value term f (a lambda abstraction), a computation term (still to be reduced), or a continuation term (embedding a context) as

[18] Adapted from the Community Scheme Wiki: http://community.schemewiki.org/?call-with-current-continuation (accessed Oct. 26, 2020).

its argument. Accordingly, there are three syntactic abstractions:

$$
\begin{array}{lcl}
A_{(\text{call/cc}\ \cdot)}(f) & = & \mu\{\mathtt{sh}(\bar{k}) \Rightarrow f \gg \mathtt{apply}(\bar{k},\bar{k})\} \\
A_{(\text{call/cc}\ \cdot)}(t_k) & = & \mu\{\mathtt{sh}(\bar{k}) \Rightarrow \bar{k} \gg t_k\} \\
A_{(\text{call/cc}\ \cdot)}(t_c : \mathtt{Fun}) & = & \mu\{\mathtt{sh}(\bar{k}) \Rightarrow t_c \gg \mathtt{sh}(\tilde{\mu}\{f \Rightarrow f \gg \mathtt{apply}(\bar{k},\bar{k})\})\} \\
A_{(\text{call/cc}\ \cdot)}(t_c : \overline{T}) & = & \mu\{\mathtt{sh}(\bar{k}) \Rightarrow t_c \gg \mathtt{sh}(\tilde{\mu}\{\bar{k}_0 \Rightarrow \bar{k} \gg \bar{k}_0\})\}
\end{array}
$$

In the syntactic abstraction for passing a lambda abstraction to call/cc, the outer continuation is passed to the `apply` destructor both as the function's argument and as its result continuation. For passing a continuation term the embedding directly wraps a command that triggers the continuation (which simply stays as it is under the embedding) with the outer continuation as its input; i.e., the continuation t_k is a continuation that takes a continuation as its input, which logically corresponds to double negation. Passing a computation term is embedded similarly, just with evaluation of that term prefixed; there are two embeddings, applied depending on whether the evaluation of the computation term will result in a lambda value (i.e. the embedded term is of `Fun` type) or a continuation. In the example that follows below, there are functions which may either return a continuation or a function value, for which we use a simple sum type with constructors `fun` and `cnt`; we straightforwardly extend the reduction rules and macros for call/cc to allow discriminating between the two. We also assume that there is a wrapper for continuation calls and lambda applications which matches on these constructors and picks the correct construct; we omit this wrapper for ease of reading when presenting the example.

In the example in Fig. 5.31, we see a continuation that takes a continuation as its input, as mentioned above, in action. In order to understand this example in detail, let us first go over the basic setup. The idea is that we model two routines which pass control back and forth between them, a.k.a. coroutines. More precisely, we have the functions `loop_1` and `loop_2` defined as infinite loops which over and over call `stuff_1` and `stuff_2`, respectively, which in turn carry out some computation that we are not further interested in as far as this example is concerned. After carrying out `stuff_1`, `loop_1` does *something* to pass control to the other routine (which we will setup to be `loop_2`), then recurses. Linking the two routines initially is achieved by calling the `cmp_1` function, which expects as its argument a first-class function that embeds the other routine `cmp_2`.[19] Before calling `loop_1` to get the computation started, `cmp_1` stores that first-class function on the heap. (We omit any mention of heap addresses since we only need a single storage cell anyway.) For writing to and reading from the heap, the functions `set` and `get` are used, which are implemented "natively", i.e. using the $\mathcal{PF}$ daimon for mutable state discussed in Section 4.2.3. Since all functions need to return some value, we let `set` return the unit value `()` from the unit data type. We also assume that `stuff_1` and `stuff_2` return `()`, and use a simple function `seq`, that ignores its first input and returns its second input, to model sequential computation (we use the infix abbreviation ; for `seq`; see Fig. 5.31).

Let us now look at the *something* necessary for passing control between the routines mentioned above, i.e., this part:

$$\mathtt{set}@(\text{call/cc}\ \mathtt{get}@())$$

What this does first is retrieve what is currently on the store, which is either the initial first-class function (for the respective other routine), or, later, a continuation that allows to jump back to where one previously jumped out of a routine. That

[19] One can also switch the roles of `cmp_1` and `cmp_2`; this does not make a difference for this example except for determining which routine goes first.

`seq` $\mapsto$ $(return(y))$
(`seq`@(t_1, t_2) is abbreviated with $t_1; t_2$. `seq` is associative. Leading `()`; omitted.)
`set` "native" (daimonic) using **SetBox**, see text
`get` "native" (daimonic) using **OpenBox**, see text

`loop_1` $\mapsto$ (`stuff_1`@(); `set`@(call/cc `get`@()); `loop_1`@())
`loop_2` $\mapsto$ (`stuff_2`@(); `set`@(call/cc `get`@()); `loop_2`@())

`cmp_1` $\mapsto$ (`set`@(k); `loop_1`@())
`cmp_2` $\mapsto$ (`set`@(k); `loop_2`@())

FIGURE 5.31: Call/cc coroutines example.

(except for the first time) continuation, say k, is then called with the current continuation, say, k', which begins by calling `set` upon its input. Thus, when k' is eventually called within the computation specified by the body of k, it returns to the routine we originally came from and then immediately sets the storage cell to the remaining computation of the other routine; this is because on the other routine the same process plays out, calling k' with the current continuation.

As a concrete example, consider the reduction sequence on the left-hand side of Fig. 5.32. The term (call/cc `get`()) is in the focus of the reduction for the first time just after it says "(`stuff_1` happens)", and hence the actual computation of the first routine has been run, for the first time. The `get`() fetches `cmp_2`, encapsulated into a first-class function k_2, from the store; k_2 is then called with the current continuation (embedding context) $\mathcal{E}_1$ and immediately sets the storage cell to that continuation, to allow us to go back to the first routine, then goes on with `loop_2`(). This means that first "`stuff_2` happens", and we then again encounter (call/cc `get`()). Since `get`() now retrieves the continuation stored previously, this time a continuation is called with the current continuation, namely $\mathcal{E}_1$ with $\mathcal{E}_2$ (highlighted with a gray shading); $\mathcal{E}_1$ brings us to the first routine, where the first thing that happens is to set the storage cell to $\mathcal{E}_2$, to facilitate the return to the second routine. This return then happens later on in the same way, by calling that continuation contained in the storage cell with the current continuation, and so on, always moving back and forth between the routines and in effect alternately running `stuff_1`() and `stuff_2`().

On the right-hand side of Fig. 5.32, we again see (just as in the reduction sequence figures above) the corresponding reduction steps in $\mathcal{PF}$ after macro embedding. In particular, the command corresponding to the term $\mathcal{E}_2[\text{call}(\mathcal{E}_1, \mathcal{E}_2)]$ (highlighted in gray in Fig. 5.32) is translated from a machine configuration such that the context $\mathcal{E}_2$ is discarded, in the same way as explained at the end of Section 5.3.1; as for let/cc in Section 5.3.1, this reflects that calling a continuation passed by call/cc discards the current evaluation context. As before, the general close correspondence between surface and embedded steps can be shown by straightforwardly extending the results from above to take the new syntactic abstractions of this section into consideration.

	$\mathcal{E}_0[\texttt{cmp_1}(k_2)]$		$k_2 \gg \texttt{cmp_1}(k_0)$
	$(k_2 := \lambda k.\texttt{cmp_2}(k))$		
$\triangleright_v$	$\mathcal{E}_0[\texttt{set}(k_2);\texttt{loop_1}()]$	$\triangleright$	$\texttt{set}(k_2) \gg \texttt{sh}(\bar{\mu}\{a \Rightarrow \texttt{loop_1}() \gg \texttt{sh}(\bar{\mu}\{b \Rightarrow a \gg \texttt{seq}(b, k_0)\})\})$
$\triangleright_v$	$\mathcal{E}_0[\texttt{loop_1}()]^{k_2}$	$\triangleright^2$	$(\texttt{loop_1}() \gg \texttt{sh}(\bar{\mu}\{b \Rightarrow () \gg \texttt{seq}(b, k_0)\}))^{k_2}$
$\triangleright_v$	$\mathcal{E}_0[\texttt{stuff_1}();\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_1}()]^{k_2}$	$\triangleright$	$(\texttt{stuff_1}() \gg \texttt{sh}(\bar{\mu}\{a \Rightarrow \texttt{seq}(...) \gg ...\}))^{k_2}$
	(stuff_1 happens)		
$\triangleright_v^*$	$\mathcal{E}_0[\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_1}()]^{k_2}$	$\triangleright^*$	$(\texttt{set}(\text{call/cc}\,\texttt{get}()) \gg \bar{\mu}\{b \Rightarrow \texttt{loop_1}() \gg ...\})^{k_2}$
$\triangleright_v$	$\mathcal{E}_1[((\texttt{get}())\ \mathcal{E}_1)]^{k_2}$	$\triangleright^2$	$(\texttt{get}() \gg \texttt{sh}(\bar{\mu}\{\texttt{fun}(f) \Rightarrow f \gg \texttt{apply}(\mathcal{E}_1, \mathcal{E}_1) \mid ...\}))^{k_2}$
$\triangleright_v$	$\mathcal{E}_1[(k_2\ \mathcal{E}_1)]^{k_2}$	$\triangleright^2$	$(k_2 \gg \texttt{apply}(\mathcal{E}_1, \mathcal{E}_1))^{k_2}$
	$(\mathcal{E}_1 := \mathcal{E}_0[\texttt{set}([]);\texttt{loop_1}()])$		
$\triangleright_v$	$\mathcal{E}_1[\texttt{set}(\mathcal{E}_1);\texttt{loop_2}()]^{k_2}$	$\triangleright^2$	$(\mathcal{E}_1 \gg \texttt{set}(\bar{\mu}\{a \Rightarrow \texttt{loop_2}() \gg \texttt{sh}(\mathcal{E}_1)\}))^{k_2}$
$\triangleright_v$	$\mathcal{E}_1[\texttt{loop_2}()]^{\mathcal{E}_1}$	$\triangleright^2$	$(\texttt{loop_2}() \gg \texttt{sh}(\mathcal{E}_1))^{\mathcal{E}_1}$
$\triangleright_v$	$\mathcal{E}_1[\texttt{stuff_2}();\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_2}()]^{\mathcal{E}_1}$	$\triangleright$	$(\texttt{stuff_2}() \gg \texttt{sh}(\bar{\mu}\{a \Rightarrow \texttt{seq}(...) \gg ...\}))^{\mathcal{E}_1}$
	(stuff_2 happens)		
$\triangleright_v^*$	$\mathcal{E}_1[\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_2}()]^{\mathcal{E}_1}$	$\triangleright^*$	$(\texttt{set}(\text{call/cc}\,\texttt{get}()) \gg \bar{\mu}\{b \Rightarrow \texttt{loop_2}() \gg ...\})^{\mathcal{E}_1}$
$\triangleright_v$	$\mathcal{E}_2[\text{call}(\texttt{get}(), \mathcal{E}_2)]^{\mathcal{E}_1}$	$\triangleright^2$	$(\texttt{get}() \gg \texttt{sh}(\bar{\mu}\{... \mid \texttt{cnt}(\bar{k}) \Rightarrow \mathcal{E}_2 \gg \bar{k}\}))^{\mathcal{E}_1}$
$\triangleright_v$	$\mathcal{E}_2[\text{call}(\mathcal{E}_1, \mathcal{E}_2)]^{\mathcal{E}_1}$	$\triangleright^2$	$(\mathcal{E}_2 \gg \mathcal{E}_1)^{\mathcal{E}_1}$
	$(\mathcal{E}_2 := \mathcal{E}_1[\texttt{set}([]);\texttt{loop_2}()])$		
$\triangleright_v$	$\mathcal{E}_0[\texttt{set}(\mathcal{E}_2);\texttt{loop_1}()]^{\mathcal{E}_1}$	$\triangleright^2$	$(\mathcal{E}_2 \gg \texttt{set}(\bar{\mu}\{a \Rightarrow \texttt{loop_1}() \gg \texttt{sh}(\mathcal{E}_0)\}))^{\mathcal{E}_1}$
$\triangleright_v$	$\mathcal{E}_0[\texttt{loop_1}()]^{\mathcal{E}_2}$	$\triangleright^2$	$(\texttt{loop_1}() \gg \texttt{sh}(\mathcal{E}_0))^{\mathcal{E}_2}$
$\triangleright_v$	$\mathcal{E}_0[\texttt{stuff_1}();\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_1}()]^{\mathcal{E}_2}$	$\triangleright^2$	$(\texttt{stuff_1}() \gg \texttt{sh}(\bar{\mu}\{a \Rightarrow \texttt{seq}(...) \gg ...\}))^{\mathcal{E}_2}$
	(stuff_1 happens)		
$\triangleright_v^*$	$\mathcal{E}_0[\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_1}()]^{\mathcal{E}_2}$	$\triangleright^2$	$(\texttt{set}(\text{call/cc}\,\texttt{get}()) \gg \bar{\mu}\{b \Rightarrow \texttt{loop_1}() \gg ...\})^{\mathcal{E}_2}$
$\triangleright_v^*$	$\mathcal{E}_1[\texttt{stuff_2}();\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_2}()]^{\mathcal{E}_1}$	$\triangleright^*$	... *(see above)*
$\triangleright_v^*$	$\mathcal{E}_0[\texttt{stuff_1}();\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_1}()]^{\mathcal{E}_2}$	$\triangleright^*$	...
$\triangleright_v^*$	$\mathcal{E}_0[\texttt{set}(\text{call/cc}\,\texttt{get}());\texttt{loop_1}()]^{\mathcal{E}_2}$	$\triangleright^*$	...
etc.			

FIGURE 5.32: Example reduction and embedded reduction in $\mathcal{PF}$ for the coroutines example in Fig. 5.31 (e.g. $\mathcal{E}_0 := []$, $k_0 := \bar{\mu}\{() \Rightarrow \textbf{Result}(())\}$).

5.4 Case Study: Transposing a Java Subset

This section walks through an example given by Lämmel and Rypacek (2008), who, in a formally precise categorical (i.e. category theory) setting, show how algebraic and coalgebraic approaches to programming are semantically equivalent. They term this result the *Expression Lemma* due to the relation to the extensibility duality with its canonical expression language example. They represent the two approaches (algebraic and coalgebraic) with an idiomatic Java program using mutable state with two classes for two kinds of expressions in a simple expression language and a Haskell program with two functions operating on such expressions, respectively; the two Languages Java and Haskell, together with a certain style of programming, can intuitively be understood to correspond to categorical semantics, but the mapping from these concrete surface languages to category theory is not made formally precise by Lämmel and Rypacek (2008). This section utilizes what the previous sections of this chapter established to close this gap within the formal framework $\mathcal{PF}$ and demonstrates the compatibility between this development and that of Lämmel and Rypacek.

In the *algebraic* approach to programming, a program can be thought of as consisting of recursive data types and *folds*, also called *catamorphisms* over these. Folds are well-known from functional programming practice, e.g. for the special case of lists, where the recursive result for the rest of the list is combined with its head as specified by some function. Generally, folding a function f over a term of some recursive data type means to apply the fold to the subterms of that same type, obtaining the recursive results, then apply f to these results and possibly other terms (like the head of the list). Categorically, a recursive data type D is conceived as being formed from a *functor* F that maps types to types (actually, categories to categories, but we work only with functors from one category to itself), where the input replaces the recursive occurrences of D in the signatures of the type declaration, such that one can think of D as being the result of forming the least fixed point μF of F. In categorical semantics, the *objects* of a category are types and functions between those are its *morphisms* (also called *arrows*), and the fixed point of F is described by means of a morphism $F\,\mu F \longrightarrow \mu F$ which "collapses" one layer of recursive structure. This kind of morphism is called an *initial F-algebra*; in general a morphism $FX \longrightarrow X$ is called an F-algebra, and one can think of this as a function as would be passed to a fold, with X being the recursive result type.

Coalgebraic semantics are dual to algebraic semantics, in the categorical sense of inverting the arrows. In this context we consider a different kind of functor B, also called an *interface functor*, which concerns the form of the output of computation rather than that of the input like F does. In practice, types specified by outputs are for instance found in Object-Oriented Programming, where an interface (to be implemented by a class) consists of method signatures. An interface functor B can be understood as taking a type as its input and replacing each *corecursive* occurrence of the type specified by the interface with this input; the type specified by the interface one can think of as being the result of forming the greatest fixed point νB of B. Methods can be implemented, in particular, by calling the method on fields of a class which has the same interface type, i.e. recursive subobjects. To let a class implement an interface, one has to implement all the methods specified by it. The essence of this can be described by an *unfold*, also called *anamorphism*, which produces an *unfolded* value of νB from some initial value of some to-be-unfolded type and a function f which describes how to produce the different parts of the output (one per method)

from a value of this type, by applying f on the initial value and in the result corecursively unfolding[20] the relevant (as identified by B) subterms which are of the to-be-unfolded type. Categorically, f is a morphism $X \longrightarrow BX$ called a *B-coalgebra* (dually to algebras), where X is this type to-be-unfolded (the corecursive type).

For example, consider streams where one may access the head which is an integer or the tail which itself is a stream, and a class with two (constructor) parameters, also for streams. The class could be implemented, for instance, by returning as the head of the stream the sum of the two heads of the streams passed as parameters, and for the tail invoking the constructor of the class on the tails (effectively zipping the streams with addition). A coalgebra for the stream interface functor B takes some value of some type X and produces a pair of an integer (the head of the stream) and a value of type X. When we specialize X to $(\nu B)^2$, pick the appropriate (i.e., semantically fitting with the class) coalgebra, the unfold construction for $(\nu B)^2$ and this coalgebra gives us a morphism from $(\nu B)^2$ to νB that corresponds to the class. This is because unfolding a $(\nu B)^2$ using this coalgebra means to first apply the coalgebra to the input, which gives us a $B\,(\nu B)^2 = \texttt{Int} \times (\nu B)^2$ (adding the heads), and then corecursively unfold the $(\nu B)^2$ in the second component (zipping the tails).

The Expression Lemma forms a semantic connection between the algebraic and the coalgebraic approach, in the following way: The algebras and coalgebras that semantically make up the programs are themselves combined from so-called distributive laws (a special form of natural transformation going between functors). An algebraic program gives rise to a single distributive law obtained by combining these distributive laws in a certain way, and what was described (in Lämmel and Rypacek's work) informally as a coalgebraic program that is (by "folklore" (Lämmel and Rypacek, 2008)) expected to have the same semantics, gives rise to the same distributive law. The Expression Lemma says that any distributive law, including the distributive law "extract[ed]" (Lämmel and Rypacek, 2008) from both programs, can be used for either a coalgebraic construction or an algebraic construction such that both result in the same morphism from μF to νB.

Our walkthrough begins by embedding into $\mathcal{PF}$ a subset of Java (building on the codata fragment macros from Section 5.2.2) that is sufficient for the kind of example Lämmel and Rypacek consider. We consider the difference between this Java subset and the codata fragment and then use program transposition to arrive at the data fragment and, eventually, Haskell.

We then specify a rather straightforward mapping from the data and codata fragment to categorical semantics. For some program, these also give rise to a distributive law that captures the overall semantics of the program, along the same lines as informally described for Java and Haskell programs. Transposition between the data and codata fragment[21] reflects the development surrounding the Expression Lemma in the sense that *the distributive laws (which may be plugged into the Expression Lemma) extracted from two programs (one data and one codata) are the same if and only if the programs are related by transposition.*

The last part of our walk through the example of Lämmel and Rypacek (2008) using the $\mathcal{PF}$ lens considers the difference between the data fragment and (a substantive subset of) Haskell (as for the codata fragment and Java). Concretely, the formal correspondence between the concrete surface languages (Java and Haskell in this case) and the categorical (co)algebraic semantics, not considered by Lämmel and

[20] Note that we are specifying how a value of νB, a greatest fixed point, is to be *destructed*, i.e. what we are describing here is an infinite process to be partially triggered *on demand*.

[21] This surface transposition is compatible with $\mathcal{PF}$ transposition as shown in Section 5.2.2 and can thus be carried out directly in $\mathcal{PF}$.

Rypacek (2008), is established by considering the changes made to the macro embeddings of the (co)data fragment (which in turn is mapped to categorical semantics) in order to account for evaluation strategies (following Section 5.1) or extra-logical aspects like mutable state.

The section finishes by considering $\mathcal{PF}$ transposition for programs with a different kind of surface feature not considered in the example, non-linear control flow constructs. We will see how, when adding exception handling as a feature of the surface language using an embedding (following the ones shown in Section 5.3), transposition in $\mathcal{PF}$ preserves this feature. Concretely, transposing a Java program containing exception handling almost directly results in a program that fits an embedding of some ML dialect with a similar feature.

5.4.1 Embedding a Java subset with mutability

We consider a subset of Java with interfaces, classes, mutable fields and local variables, but without inheritance or Generics (though a simple variant of the latter would be straightforward to add in the parametric polymorphism variant of $\mathcal{PF}$). Fig. 5.33 gives the macro embedding of that language into $\mathcal{PF}$; we assume primitive types like **int** to be available along with their value literals. We also assume that any primitive operator, like $+$, has an $\mathcal{PF}$ function that it translates to. This translation follows the same approach as that for method calls, i.e. the continuation k is passed as the last argument of the function corresponding to the operator, e.g. $+(4, 2, k)$.

The Java example of Lämmel and Rypacek (2008) is shown in Fig. 5.34. It is a Java implementation of a simple expression language with nodes for number literals (class `Num`) and addition nodes (class `Add`), which both implement the interface `Expr` which specifies the methods available to be applied to expressions: evaluation to a number (method `eval`) and replacing each number literal v contained in the tree by $v \mathrm{mod} v_0$, with some number v_0 passed to the relevant method `modn` as an additional argument. The following illustrates key aspects of the embedding into $\mathcal{PF}$ by means of that example.

Embedding the `eval` method implementation from the `Num` class results in this clause of a local producer which corresponds to the object created using **new**:

$$\texttt{eval}(\bar{k}) \Rightarrow v \gg \bar{k}$$

Likewise, embedding the `modn` method implementation results in its other clause:

$$\texttt{modn}(v_0, \bar{k}) \Rightarrow \mu\{\bar{k} \Rightarrow \mathbf{OpenBox}(v_0, \bar{k})\} \gg \tilde{\mu}\{a_0 \Rightarrow \mu\{\bar{k} \Rightarrow \mathbf{OpenBox}(v, \bar{k})\} \gg \tilde{\mu}\{a \Rightarrow \mathbf{Mod}(a, a_0, \tilde{\mu}\{b \Rightarrow \mathbf{SetBox}(v, b, \mathbf{void} \gg \bar{k})\})\}\}$$

The local producer that these two clauses comprise becomes, after transposition, a constructor annotated as local. However, it would have been just fine to create a top-level producer function for this object since there is exactly one such "object producer" for each class constructor anyway. The other `Expr` producer is for objects of the `Add` class and also becomes a constructor under transposition. Thus, somewhat unsurprisingly since our embedding is built around the codata embedding, the core of the transposition result is this data type:

$$\mathbf{data}\ \texttt{Expr}\ \{\ \texttt{num}(\mathbf{Ref}\langle\mathbf{int}\rangle) \mid \texttt{add}(\mathbf{Ref}\langle\texttt{Expr}\rangle, \mathbf{Ref}\langle\texttt{Expr}\rangle)\ \}$$

A bit more interesting is how the calls to these constructors now appear in the surrounding producer function responsible for object initialization (which, being of type `WrapP`, is not itself affected by transposition). Within the bodies of these producer functions the innermost commands ("following" commands responsible for

Interfaces:

$$\phi(\textbf{interface}\ T\ \{\ \overrightarrow{T_i\ m_i(\ \overrightarrow{T_i^j}^j\ )}^i\ \}) = \textbf{codata}\ T\ \{\ \overrightarrow{m_i(\ \overrightarrow{T_i^j}^j, \overline{T_i})}^i\ \}$$

Classes:

$$\phi(\textbf{class}\ c\ \textbf{impl}.\ T\ \{$$
$$\overrightarrow{T_k\ y_k}^k; c(\overrightarrow{T_k\ y_k}^k)\{\overrightarrow{\textbf{this}.y_k = y_k}^k\};\ \overrightarrow{T_i\ m_i(\ \overrightarrow{x_i^j}^j\ )\{b_i\}}^i\ \}) =$$
$$\textbf{function}_{_}\ c(\overrightarrow{y_k : T_k}^k) : T\ \{\bar{k} \Rightarrow \phi(\text{<init>}(\overrightarrow{T_k\ y_k = y_k}^k;\ \overrightarrow{T_i\ m_i(\ \overrightarrow{x_i^j}^j\ )\{b_i\}}^i\ ))\}$$

Object initialization:

$$\phi(\text{<init>}(s;i)) = \phi(s)[\phi(\text{<init>}(i))]$$
$$\phi(\text{<init>}(\ \overrightarrow{T_i\ m_i(\ \overrightarrow{x_i^j}^j\ )\{b_i\}}^i\ )) = \texttt{pos}(\mu\{\ \overrightarrow{m_i(\ \overrightarrow{x_i^j}^j, \bar{k}_i) \Rightarrow \phi'(b_i)}^i\ \}) \gg \bar{k}$$

Bodies:

$$\phi(\textbf{return}\ t) = \mu\{\bar{k} \Rightarrow \phi(t) \gg \bar{k}\}$$
$$\phi(s;b) = \mu\{\bar{k} \Rightarrow \phi(s)[\phi'(b)]\}$$

Statements:

$$\phi(T\ x = t) = \textbf{NewBox}(t, \bar{\mu}\{x \Rightarrow []\})$$
$$\phi(x = t) = \phi(t) \gg \texttt{posk}(\bar{\mu}\{a \Rightarrow \textbf{SetBox}(x, a, [])\})$$
$$\phi(t.m(\overrightarrow{t_i}^i)) = \phi_{\text{eval}}(\{x \mapsto \phi(t)\} \cup \overrightarrow{x_i \mapsto \phi(t_i)}^i, x \gg \texttt{posk}(m(\overrightarrow{x_i}^i, \bar{\mu}\{x \Rightarrow []\})))$$

Terms:

$$\phi(x) = \mu\{\bar{k} \Rightarrow \textbf{OpenBox}(x, \bar{k})\}$$
$$\phi(t.m(\overrightarrow{t_i}^i)) = \mu\{\bar{k} \Rightarrow \phi_{\text{eval}}(\{x \mapsto \phi(t)\} \cup \overrightarrow{x_i \mapsto \phi(t_i)}^i, x \gg \texttt{posk}(m(\overrightarrow{x_i}^i, \bar{k})))\}$$
$$\phi(\textbf{new}\ c(\overrightarrow{t_i}^i)) = \mu\{\bar{k} \Rightarrow \phi_{\text{eval}}(\overrightarrow{x_i \mapsto \phi(t_i)}^i, c(\overrightarrow{x_i}^i) \gg \bar{k})\}$$

Auxiliary: Chained evaluation:

$$\phi_{\text{eval}}(\emptyset, C) = C$$
$$\phi_{\text{eval}}(\{x \mapsto t\} \cup \sigma, C) = t \gg \texttt{posk}(\bar{\mu}\{x \Rightarrow \phi_{\text{eval}}(\sigma, C)\})$$

FIGURE 5.33: Translating a Java subset into $\mathcal{PF}$.

```
interface Expr { int eval(); void modn(); }

class Num implements Expr {
    int v;
    Num(int v) { this.v = v; }
    int eval() { return v; }
    void modn(int v0) { v = v % v0; }
}

class Add implements Expr {
    Expr l; Expr r;
    Add(Expr l, Expr r) { this.l = l; this.r = r; }
    int eval() { return l.eval() + r.eval(); }
    void modn(int v0) { l.modn(v0); r.modn(v0); }
}
```

FIGURE 5.34: Java example from Lämmel and Rypacek (2008).

field initialization and the statements specified in the class constructor) are now simply $\mathtt{pos}(\mathtt{num}(v)) \gg \bar{k}$ and $\mathtt{pos}(\mathtt{add}(l,r)) \gg \bar{k}$, respectively. The variables v, l, and r are bound via daimonic commands, corresponding to mutable fields.

We next consider a more restricted subset of Java and its relation to codata, before comparing the transposition result above to transposition results of that restricted subset and the codata fragment, and to Haskell.

5.4.2 Functional update restriction and codata

Lämmel and Rypacek (2008) picked this example such that it falls into a specific class of Java programs, namely ones where the use of mutable state is inessential and can easily be avoided, merely losing some superficial convenience (as well as, arguably, using a less idiomatic style of "object-oriented programming" than one may be used to). They informally describe how to get from a Java program falling in this class to categorical semantics, then do the same for a Haskell program intended to fulfill the same task, and finally demonstrate that and how precisely the semantics are equivalent, proving their Expression Lemma.

We will now consider a mutable-state-less subset of the surface language (Java subset) described by the macros above that can serve as the target of an intermediate step. A program in the full Java subset that makes use of mutable state can be transformed to an equivalent program in this more restricted subset that, instead of mutating fields, returns a new object with the updated field. Using *functional updates* in OO, instead of mutation, is close to the formal categorical semantics used by Lämmel and Rypacek (2008), precisely, *coalgebraic* semantics, and was also used to illustrate such semantics practically (Jacobs, 1996). The restricted Java subset we consider can be further modified to arrive at the codata fragment, which arguably really capture the essence of programming coalgebraically. This modification is fairly minor, demonstrating that "functional update Java" is really just superficially different from the codata fragment. The following details the functional update restriction,

```
interface Expr { int eval(); Expr modn(); }

class Num implements Expr {
    int v;
    Num(int v) { this.v = v; }
    int eval() { return v; }
    Expr modn(int v0) { return new Num(v % v0); }
}

class Add implements Expr {
    Expr l; Expr r;
    Add(Expr l, Expr r) { this.l = l; this.r = r; }
    int eval() { return l.eval() + r.eval(); }
    Expr modn(int v0) { return new Add(l.modn(v0), r.modn(v0)); }
}
```

FIGURE 5.35: Java example from Lämmel and Rypacek (2008), translated to functional update style (differences shaded in gray).

together with a sketch of how to transform programs, and the delta to apply to it to arrive at the codata fragment.

The functional update restriction for the macros above could actually be specified in two sentences directly relating to daimonic commands for mutable state:

1. The daimonic command **SetBox** is disallowed.
2. The daimonic command **NewBox** is only allowed in object initialization.

However, this would allow method bodies with useless method calls like

```
e.eval(); return e;
```

The only body that we really need for functional update is that for **return**. By disallowing other forms of bodies, we automatically exclude all uses of **SetBox**, and of **NewBox** outside of object initialization, so this condition is actually sufficient:

Only bodies of the form **return** t are allowed.

Translating the example from Fig. 5.34 to functional update style such that it complies with this restriction gives the program shown in Fig. 5.35. Where previously some method had return type **void**, it now has the type of the interface it was specified in, in our example: `Expr`. Accordingly, a method implementation that modified a field, setting it to some value v, now returns a new object with that field initialized with v, as in the method implementation for `modn` in class `Num`. Thus, a call to a method that previously had return type **void** now returns a new object. In the example, method `modn` in class `Add` calls `modn` on the fields of type `Expr` (a recursive structure); in functional update style, these calls return objects of type `Expr` which are then passed as arguments to return a new object instead of recursively modifying the current object.

The modifications that turn this restricted Java subset into the codata fragment can be described as consisting of two steps. The first eliminates the last remaining

Interfaces:

$\phi(\textbf{interface}\ T\ \{\ \overrightarrow{T_i\ m_i(\ \overrightarrow{T_i^j}^j\ )}^i\ \})$ $= \textbf{codata}\ T\ \{\ \overrightarrow{m_i(\ \overrightarrow{T_i^j}^j, \overline{T_i})}^i\ \}$

Classes:

$\phi(\textbf{class}\ c\ \textbf{impl}.\ T\ \{$

$\overrightarrow{T_k\ y_k}^k; c(\ \overrightarrow{T_k\ y_k}^k)\{\ \overrightarrow{\textbf{this}.y_k = y_k}^k\}; \overrightarrow{T_i\ m_i(\ \overrightarrow{x_i^j}^j\ )\{b_i\}}^i\ \}) =$

$\textbf{function}_{_}\ c(\ \overrightarrow{y_k : T_k}^k) : T\ \{\ \overrightarrow{m_i(\ \overrightarrow{x_i^j}^j, \bar{k}_i) \Rightarrow \phi'(b_i)}^i\ \}$

Bodies (only return, macro same as in Fig. 5.33):

$\phi(\textbf{return}\ t)$ $= \mu\{\bar{k} \Rightarrow \phi(t) \gg \bar{k}\}$

Terms:

$\phi(x)$ $= x$

other syntactic abstractions and macros same as in Fig. 5.33

No object initialization or statements

FIGURE 5.36: Translating a functional update Java subset into $\mathcal{PF}$. (Syntactic abstractions are a subset of those in Fig. 5.33.)

traces of mutability. Due to the restrictions imposed on the macros, storage cells could not be modified anymore anyways, thus it is actually possible to reuse the same surface syntactic abstractions, merely changing the macros specified for these. The second step does change the syntactic abstractions, but in a superficial way.

Let us consider the new macros of the first step, shown in Fig. 5.36. Since we restricted bodies to only return clauses, the only place where (using the previous macros) statements for variable assignment were still possible was in object initialization. But we do not use mutable state, and hence fields, anymore, so we can just directly translate a class to a producer function, simply ignoring the assignments specified in the constructor and making assignment statements unnecessary. A class is now really just a different notation for a producer function. In line with this removal of field storage, variables can now be taken over unchanged, eliminating **OpenBox**; the other macros for terms are orthogonal to issues of mutable state and stay the same.

Now it is rather straightforward to see that these remaining syntactic abstractions are merely a notational variation of those of the codata fragment. This is obviously case for the **interface** abstraction. As mentioned, one can ignore the assignments in constructors, hence classes are also just a variation of producer function syntax; that is, if we remove the now superfluous keyword **return**. The syntax of terms is also fully identical once we likewise remove the keyword **new**.

5.4.3 Program transposition and the Expression Lemma

After the preparatory steps above we have arrived at a program in the codata fragment, shown in Fig. 5.37 (top), which can be transposed to obtain a data fragment

$$
\begin{aligned}
&\mathbb{C} := \{\texttt{Expr}\}, \mathbb{D} := \{(\texttt{Expr}, \texttt{add}), (\texttt{Expr}, \texttt{modn})\},\\
&\mathbb{F} := \{(\texttt{Expr}, \texttt{Num}), (\texttt{Expr}, \texttt{Add})\},\\
&\Sigma^{\mathbb{D}} := \{(\texttt{Expr}, \texttt{eval}) \mapsto ((), \texttt{Int}), (\texttt{Expr}, \texttt{modn}) \mapsto ((\texttt{Int}), \texttt{Expr})\},\\
&\Sigma^{\mathbb{F}} := \{(\texttt{Expr}, \texttt{Num}) \mapsto (\texttt{Int}), (\texttt{Expr}, \texttt{Add}) \mapsto (\texttt{Expr}, \texttt{Expr})\},\\
&\mathcal{F} := \{\\
&\quad (\texttt{Expr}, \texttt{Num}) \mapsto \{\texttt{eval} \mapsto \mathit{return}(x), \texttt{modn} \mapsto \texttt{Num}(x \,\%\, y)\},\\
&\quad (\texttt{Expr}, \texttt{Add}) \mapsto \{\texttt{eval} \mapsto \texttt{eval}(x_1) + \texttt{eval}(x_2), \texttt{modn} \mapsto \texttt{Add}(\texttt{modn}(x_1, y), \texttt{modn}(x_2, y))\}\}.
\end{aligned}
$$

$$
\begin{aligned}
&\mathbb{D} := \{\texttt{Expr}\}, \mathbb{C} := \{(\texttt{Expr}, \texttt{Num}), (\texttt{Expr}, \texttt{Add})\},\\
&\mathbb{F} := \{(\texttt{Expr}, \texttt{eval}), (\texttt{Expr}, \texttt{modn})\},\\
&\Sigma^{\mathbb{C}} := \{(\texttt{Expr}, \texttt{Num}) \mapsto (\texttt{Int}), (\texttt{Expr}, \texttt{Add}) \mapsto (\texttt{Expr}, \texttt{Expr})\},\\
&\Sigma^{\mathbb{F}} := \{(\texttt{Expr}, \texttt{eval}) \mapsto ((), \texttt{Int}), (\texttt{Expr}, \texttt{modn}) \mapsto ((\texttt{Int}), \texttt{Expr})\},\\
&\mathcal{F} := \{\\
&\quad (\texttt{Expr}, \texttt{eval}) \mapsto \{\texttt{Num} \mapsto \mathit{return}(x), \texttt{Add} \mapsto \texttt{eval}(x_1) + \texttt{eval}(x_2)\},\\
&\quad (\texttt{Expr}, \texttt{modn}) \mapsto \{\texttt{Num} \mapsto \texttt{Num}(x \,\%\, y), \texttt{Add} \mapsto \texttt{Add}(\texttt{modn}(x_1, y), \texttt{modn}(x_2, y))\}\}.
\end{aligned}
$$

FIGURE 5.37: Running example in the codata fragment (top) and data fragment (bottom).

Map data consumers to morphisms:

$$[\![f \mapsto \overrightarrow{c \mapsto t}]\!] \quad := \quad (\![\![f \mapsto \overrightarrow{c \mapsto t}]\!]_{\text{alg}}\!)_{[\![T]\!]}, \text{ where } f = (T, ...)$$

Map data consumers to algebras:

$$[\![-]\!]_{\text{alg}} \;:\; (\mathbb{F} \to (\mathbb{C} \nrightarrow t)) \to \{h \;:\; F(B\,\mu F) \longrightarrow (B\,\mu F) \mid F, B \;:\; \mathbf{C} \to \mathbf{C}\}$$

$$[\![f \mapsto \overrightarrow{c \mapsto t}]\!]_{\text{alg}} \quad := \quad \nabla\, \overrightarrow{[\![(f, c_i, t_i)]\!]}^{i}$$

if $\Sigma^{\mathbb{F}}(f) = ((), T_{\text{pr}})$ and $\exists T$ s.t. for all c_i:

$$c_i = (T, ...) \wedge (\Sigma^{\mathbb{C}}(c_i) = T^n \vee \exists \overrightarrow{T_{\text{pr}}}.\Sigma^{\mathbb{C}}(c_i) = \overrightarrow{T_{\text{pr}}})$$

$$[\![f \mapsto \overrightarrow{c \mapsto t}]\!]_{\text{alg}} \quad := \quad \nabla\, \overrightarrow{((\mathsf{in}_{[\![T]\!]} \circ \iota_i) \circ -) \circ ([\![(f, c_i, t_i)]\!])_{\mu[\![T]\!]}}^{i}$$

if $\Sigma^{\mathbb{F}}(f) = (\overrightarrow{T_{\text{pr}}}, T)$ and for all c_i:

$$c_i = (T, ...) \wedge (\Sigma^{\mathbb{C}}(c_i) = T^n \vee \exists \overrightarrow{T'_{\text{pr}}}.\Sigma^{\mathbb{C}}(c_i) = \overrightarrow{T'_{\text{pr}}})$$

Map data consumer equations to distributive laws:

$$[\![-]\!] \;:\; \mathbb{F} \times \mathbb{C} \times t \to \{\eta \;:\; FB \longrightarrow BF \mid F, B \;:\; \mathbf{C} \to \mathbf{C}\}$$

$$[\![(f, c, t)]\!] \quad := \quad \mathrm{FV}(t) \mapsto t \;:\; (\Pi\overrightarrow{[\![T_{\text{pr}}]\!]})[\![T_{\text{pr}}]\!] \longrightarrow [\![T_{\text{pr}}]\!](\Pi\overrightarrow{[\![T_{\text{pr}}]\!]}),$$

if $\Sigma^{\mathbb{C}}(c) = \overrightarrow{T_{\text{pr}}}$, $\Sigma^{\mathbb{F}}(f) = ((), T_{\text{pr}})$

$$[\![(f, c, t)]\!] \quad := \quad \overrightarrow{x_i}^{i \le n} \mapsto t\,\overrightarrow{[f(x_i) \mapsto x_i]}^{i \le n} \;:\; \mathsf{Id}^n[\![T_{\text{pr}}]\!] \longrightarrow [\![T_{\text{pr}}]\!]\mathsf{Id}^n,$$

if $c = (T, ...)$, $\Sigma^{\mathbb{C}}(c) = T^n$, $\Sigma^{\mathbb{F}}(f) = ((), T_{\text{pr}})$

$$[\![(f, c, c(\overrightarrow{t}))]\!] \quad := \quad \mathrm{FV}_x(\overrightarrow{t}) \mapsto \mathrm{FV}_y(\overrightarrow{t}) \mapsto \overrightarrow{t} \;:\; (\Pi\overrightarrow{[\![T'_{\text{pr}}]\!]})\mathsf{Id}^{\Pi\overrightarrow{[\![T_{\text{pr}}]\!]}} \longrightarrow \mathsf{Id}^{\Pi\overrightarrow{[\![T_{\text{pr}}]\!]}}(\Pi\overrightarrow{[\![T'_{\text{pr}}]\!]}),$$

if $c = (T, ...)$, $\Sigma^{\mathbb{C}}(c) = \overrightarrow{T'_{\text{pr}}}$, $\Sigma^{\mathbb{F}}(f) = (\overrightarrow{T_{\text{pr}}}, T)$

$$[\![(f, c, c(\overrightarrow{f(...)}^{i}))]\!] \quad := \quad \triangle^{(n)} \;:\; \mathsf{Id}^n\mathsf{Id}^{\Pi\overrightarrow{[\![T_{\text{pr}}]\!]}} \longrightarrow \mathsf{Id}^{\Pi\overrightarrow{[\![T_{\text{pr}}]\!]}}\mathsf{Id}^n,$$

if $c = (T, ...)$, $\Sigma^{\mathbb{C}}(c) = T^n$, $\Sigma^{\mathbb{F}}(f) = (\overrightarrow{T_{\text{pr}}}, T)$

FIGURE 5.38: Categorical semantics for the data fragment.

Map codata producers to morphisms:
$$[\![f \mapsto \overrightarrow{d \mapsto t}]\!] \quad := \quad [\!(\, [\![f \mapsto \overrightarrow{d \mapsto t}]\!]_{\text{coa}} \,)\!]_{[\![T]\!]}, \text{ where } f = (T, \ldots)$$

Map codata producers to coalgebras:
$$[\![-]\!]_{\text{coa}} : (\mathbb{F} \to (\mathbb{D} \twoheadrightarrow t)) \to \{h : (F\,\nu B) \longrightarrow B(F\,\nu B) \mid F, B : \mathbf{C} \to \mathbf{C}\}$$

$$[\![f \mapsto \overrightarrow{d \mapsto t}]\!]_{\text{coa}} \quad := \quad \Delta\, \overrightarrow{[\![(f, d_i, t_i)]\!]}^{i}$$
if $\Sigma^{\mathbb{F}}(f) = \overrightarrow{T_{\text{pr}}}$ and $\exists T$ s.t. for all d_i:
$$d_i = (T, \ldots) \wedge ((\exists \overrightarrow{T_{\text{pr}}}. \Sigma^{\mathbb{D}}(d_i) = (\overrightarrow{T_{\text{pr}}}, T)) \vee \Sigma^{\mathbb{D}}(d_i) = ((), T_{\text{pr}}))$$
$$[\![f \mapsto \overrightarrow{d \mapsto t}]\!]_{\text{coa}} \quad := \quad \Delta\, \overrightarrow{([\![(f, d_i, t_i)]\!])_{\nu[\![T]\!]} \circ \mathsf{Id}^n(\pi_i \circ \mathsf{out}_{[\![T]\!]})}^{i}$$
if $\Sigma^{\mathbb{F}}(f) = T^n$ and for all d_i:
$$d_i = (T, \ldots) \wedge ((\exists \overrightarrow{T_{\text{pr}}}. \Sigma^{\mathbb{D}}(d_i) = (\overrightarrow{T_{\text{pr}}}, T)) \vee \Sigma^{\mathbb{D}}(d_i) = ((), T_{\text{pr}}))$$

Map codata producer equations to distributive laws:
$$[\![-]\!] : \mathbb{F} \times \mathbb{D} \times t \to \{\eta : FB \longrightarrow BF \mid F, B : \mathbf{C} \to \mathbf{C}\}$$

$[\![(f, d, t)]\!] \quad := \quad [\![(d, f, t)]\!]$ in the data fragment semantics, with the codata destructor (producer function) signatures used as data consumer function (constructor) signatures

FIGURE 5.39: Categorical semantics for the codata fragment.

program, also shown in Fig. 5.37 (bottom). In the development of Lämmel and Rypacek (2008), the counterpart to the Java program is a Haskell program; the next subsection discusses differences between Haskell and the data fragment. This subsection defines a mapping from the data and codata fragment to categorical semantics, making the informal connection between (co)algebraic programs and category theory of Lämmel and Rypacek formally precise. Using this semantics mapping, program transposition is compatible with the way an algebraic and a coalgebraic program are semantically related by the Expression Lemma.

Specifically, for any two programs related by transposition, a so-called *distributive law* can be “extract[ed]” (Lämmel and Rypacek, 2008) (which turns out to be the same for both programs); as we will see, this distributive law can be obtained by combining individual distributive laws induced by the *equations* that comprise a (co)data program. Any distributive law (as stated by the Expression Lemma), including this combined distributive law extracted from the program, can be either employed in an algebraic construction or in a coalgebraic construction, in both cases obtaining the same morphism from the object that represents the least fixed point μF of the relevant data type functor (also called a *sum-of-products functor*) F to the object that represents the greatest fixed point νB of the relevant codata type functor (also called an *interface functor*) B.

An exposition to category theory and categorical semantics is beyond the scope of this work (the use of category theory in functional programming theory is well-established, see for instance Meijer, Fokkinga, and Paterson (1991)). Here we focus on the essential ideas required for a high-level understanding of the semantic mapping and the Expression Lemma.

The basic idea is that a (statically typed) programming language's types are mapped to the *objects* and its functions are mapped to the *morphisms* (or *arrows*) of some category $\mathbf{C}$. The concrete choice of $\mathbf{C}$ is not important beyond that it needs to admit all the constructions we need, like products, sums, and exponentials; the category of sets **SET** (with functions between sets as morphisms) works.

Let us begin by considering the semantics of the static typing information of a (co)data program: its (co)data types and signatures. Under a structural reading, a

recursive data type can be seen as the least fixed point of some "function" where the inputs replace the recursive occurrences in the constructor signatures. The same goes for codata types and greatest fixed points. In categorical terms, the "functions" are *functors* mapping a category to some other category (which means mapping objects *and morphisms*), more specifically *endofunctors* from **C** to itself, and the "least fixed points" and "greatest fixed points" are indirectly defined via the algebra and coalgebra construction; more on this follows below. For some functor F for a data type, the type obtained by forming its least fixed point, representing the data type itself, is denoted by μF, and likewise for codata types with functor B and greatest fixed point νF. For some (co)data type T, we denote the functor associated to it according to its declaration by $[\![T]\!]$.

As an example, the functor $[\![\texttt{Nat}]\!]$ associated with the data type `Nat` is $1 + \mathsf{Id}$. As with all functors associated to data types, this functor, which is hence also called a *sum-of-products functor*, is a sum over all functors associated with the individual constructor signatures, which in turn are products (1 is the empty product) formed from the result of mapping the types in the signature in the following way: If the type is a recursive occurrence of the currently considered type, map it to the identity functor Id; otherwise map it to the constant functor for the type.[22] (In particular, one might obtain a product of several Id, say n of them, which we abbreviate Id^n.) An example for an *interface functor* is $[\![\texttt{Int}]\!] \times \mathsf{Id}$, associated with codata type `Stream` with destructors $\texttt{head}() : \texttt{Int}$ and $\texttt{tail}() : \texttt{Stream}$ (where we treat `Int` as a primitive type T_{pr} for which we assume a given mapping $[\![-]\!]$ to a constant functor). Generally, we obtain an interface functor by forming the product of the functors associated with each destructor signature. As we saw, for destructors without arguments we just have identity functors and constant functors. If, as in the `modn` destructor in the case study example, we do have arguments, we form an *exponential* with the functor associated to the arguments appearing in the exponent; for modn, this exponential functor is $\mathsf{Id}^{[\![\texttt{Int}]\!]}$ since modn takes an `Int` as an argument and returns an `Expr`, which is the corecursively specified codata type.

Finally, let us consider the actual dynamic content of the program, its consumer or producer definitions. Mapping data consumers and codata producers to morphisms in **C** is defined in Fig. 5.38 and Fig. 5.39, respectively. The idea is that a consumer function is an algebraic *fold* and a producer function is a coalgebraic *unfold*. Conceptually, we replace occurrences of recursive consumer function calls[23] by holes and in this spirit map a function declaration ϕ to an *algebra* $[\![\phi]\!]_{\mathrm{alg}}$. The algebra can then be turned into the morphism $(\![[\![\phi]\!]_{\mathrm{alg}}]\!)_{[\![T]\!]}$ that semantically represents the function, by the fold construction[24] $(\!|-|\!)_{[\![T]\!]}$ for the relevant data type T. An algebra is a morphism $FX \longrightarrow X$, where F is a functor and X is an object of **C**; in the mapping, F is the functor associated with the relevant data type T. For example, replacing recursive calls in the `eval` function with holes, we obtain

$$\texttt{eval} \mapsto \{\texttt{Num} \mapsto x, \texttt{Add} \mapsto [\,] + [\,]\},$$

which is why `eval` is mapped to an algebra $(\texttt{Int} + \mathsf{Id}^2)\texttt{Int} \longrightarrow \texttt{Int}$. Operationally, the behavior of the consumer function can be described in terms of this "holed" definition as first computing the recursive results and then plugging them into the

[22] We do not consider mutually recursive types.

[23] We only consider simple function definitions which only refer to themselves (especially excluding mutual recursion).

[24] This can easily be made precise by a categorical diagram (see e.g. Lämmel and Rypacek (2008)), by means of which one can also define μF (and similarly for unfolds $[\!(-)\!]_B$ and νB), but for the purpose of this work the usual understanding of a fold (unfold), a.k.a. catamorphism (anamorphism), from functional programming suffices.

relevant term, e.g. computing the results of `eval` for its two subterms and plugging them into $[] + []$ in order to add them together. Where the recursive results are located in the structure is specified by the functor for the data type, e.g. two occurrences in the second injection for `Expr`. A similar approach maps producer functions to morphisms via coalgebras.

Mapping a function to a (co)algebra (to be used for a fold/unfold) is the central part of our development and decisive for the Expression Lemma and its compatibility with program transposition. It is perhaps best understood when conceiving of a program as a set of *equations*, which specify the result of a meeting of a consumer (a consumer function or destructor call) and a producer (a constructor or producer function call). Combining the subset of these equations for a specific function one obtains the specification for that function, but considering the entire set of equations, when two program are related by transposition, they both give rise to the same set. This is because transposition merely rearranges the same cases contained in the function definitions into a different collection of function definitions. The mapping from consumer functions to algebras, as well as from producer functions to coalgebras, follows this idea, using some operators established in category theory to combine the result of mapping equations.

Such an equation is mapped (cf. the bottom of Fig. 5.38) to a *distributive law*, which is a special kind of *natural transformation*, a structure-preserving mapping between functors, of the form $FB \longrightarrow BF$. The functor F describes the form of the input, pertaining to the relevant constructor or producer function signature; for constructors, the various such functors combine to form the functor associated with the relevant data type, as explained above. The functor B describes the form of the output, pertaining to the relevant consumer function or destructor signature; for destructors, the various such functors combine to form the functor associated with the relevant codata type. For example, the first equation of the definition of the mapping, yielding a distributive law from $(\Pi\overrightarrow{[\![T_{\mathrm{pr}}]\!]})[\![T_{\mathrm{pr}}]\!]$ $(= (\Pi\overrightarrow{[\![T_{\mathrm{pr}}]\!]}))$ to $[\![T_{\mathrm{pr}}]\!](\Pi\overrightarrow{[\![T_{\mathrm{pr}}]\!]})$ $(= [\![T_{\mathrm{pr}}]\!])$, is for equations like that for `eval` and `Num`, where the constructor signature's types as well as the function return type are all primitive (and hence F and B are both constant functors), with no recursive occurrences.

The other three equations of the definition are for other kinds of equations; in order, examples for these are, respectively, `eval` and `Add`, `modn` and `Num`, and `modn` and `Add`.[25] In line with how algebras conceptually take recursive results as inputs, for the mapping to distributive laws recursive function calls to f inside the right-hand sides of the equations for f are likewise considered as inputs to the components of the distributive law. Only the last of the four kinds of equations is mapped to a natural transformation between two non-constant functors; it is hence the only one where the component signatures vary, being parametric in what is conceptually the recursive result type. There, the operator $\triangle^{(n)}$ is used to combine a tuple of identical exponentials with Id in the base to an exponential with a tuple of Id in the base. For the concrete equation for `modn` and `Add`, in colloquial (Haskell-like) terms we can describe this as turning a pair of parametric functions of type `Int` $\to a$ to a single parametric function `Int` $\to (a, a)$, which corresponds to the right-hand side of the equation that specifies to apply $f =$ `modn` on both subterms of `Add` (cf. the component function `addModn` of Lämmel and Rypacek (2008)).

As mentioned, multiple distributive laws are combined to a single algebra or coalgebra. The result is a morphism from, in the case of two distributive laws (and

[25] The definition does not cover all possible cases of equations in catamorphic/anamorphic programs; only the ones appearing in the running example are given (the remaining cases are similar).

this readily generalizes to arbitrary numbers), $(F_1 + F_2)(B\,\mu(F_1 + F_2))$ to $(B\,\mu(F_1 + F_2))$ (algebra), or from $(F\,\nu(B_1 + B_2))$ to $(B_1 + B_2)(F\,\nu(B_1 + B_2))$ (coalgebra). That is, in order to combine the distributive laws, for one of the two kinds of functors the relevant functor (F or B) has to be the same for both distributive laws. For example, the algebra for `modn` is obtained by combining the distributive law for `modn` and `Num` with that for `modn` and `Add`, where the functor pertaining to the output $B = \mathsf{Id}^{\mathtt{Int}}$ is the same in both cases, and the functors pertaining to the input are $F_1 = \mathtt{Int}$ and $F_2 = \mathsf{Id}^2$, respectively. For coalgebras the situation is reversed, e.g. combining the distributive law for `eval` and `Num` with that for `modn` and `Num` to a coalgebra is possible since $B = \mathtt{Int}$ in both cases.

A few words about the technical details of how the distributive laws are combined are in order, since here, in general, one needs to deal with the recursive structure. In the case where the sources FB and targets BF of the distributive laws are all constant functors, e.g. as for the algebra for `eval` ($F_1 = \mathtt{Int}$, $F_2 = \mathsf{Id}^2$, $B = \mathtt{Int}$), it suffices to form their cotuple[26] (for algebras), which is denoted by $\triangledown$, or their tuple (for coalgebras), denoted by $\triangle$. In the case with non-constant functors, it is additionally necessary to "collapse" one recursive layer using $\mathsf{in}_F : F\,\mu F \longrightarrow \mu F$ (for algebras) or to "roll out" one such layer using $\mathsf{out}_B : \nu B \longrightarrow B\,\nu B$ (for coalgebras).[27] For example, to obtain the algebra $(F\,(\mu F)^{\mathtt{Int}} \longrightarrow (\mu F)^{\mathtt{Int}}$, with $F = \mathtt{Int} + \mathtt{Id}^2)$ for `modn` the distributive laws for the two relevant equations for `Num` and `Add` ($\mathtt{Int} \longrightarrow \mathtt{Int}^{\mathtt{Int}}$ and $\mathsf{Id}^2\mathsf{Id}^{\mathtt{Int}} \longrightarrow (\mathsf{Id}^2)^{\mathtt{Int}}$) are combined as follows: Precompose, for each distributive law λ_i, its component at μF with $((\mathsf{in}_F \circ \iota_i) \circ -)$, then cotuple the results. For the first of the two distributive laws we consider, the precomposition means that the morphism that comprise the target exponential $\mathtt{Int}^{\mathtt{Int}}$ are precomposed with $\mathsf{in}_F \circ \iota_1$, such that the target exponential becomes $(\mu F)^{\mathtt{Int}}$ (accessing the component is omitted in the relevant definition in Fig. 5.38 since the target of the distributive law is a constant functor). For the second distributive law, the target exponential also becomes $(\mu F)^{\mathtt{Int}}$, since the component at μF has the target exponential $((\mu F)^2)^{\mathtt{Int}}$ where the base is injected into $F\,\mu F$ with ι_2. After cotupling, the resulting morphism has source $\mathtt{Int} + ((\mu F)^{\mathtt{Int}})^2$ (which is the same as $F\,(\mu F)^{\mathtt{Int}}$) and (as we just saw) target $(\mu F)^{\mathtt{Int}}$, as desired for an algebra where the result depends upon an `Int`. Obtaining coalgebras from distributive laws works in a dual fashion, postcomposing distributive law components with out_B and the relevant projection (to be precise, postcompose with mapping these over tuples with Id^n) and then tupling the results.

The Expression Lemma states that to any distributive law a certain algebraic construction and a certain coalgebraic construction can be applied which both result in the same morphism from μF to νB. The intention of Lämmel and Rypacek (2008) in proving this result is to demonstrate the semantic equivalence of algebraic and coalgebraic programs by "extract[ing]" (Lämmel and Rypacek, 2008) a distributive law λ, combined from individual distributive laws using the $\otimes$ and $\oplus$ operators, by informally describing how these can be understood as "component functions" which can be combined to form classes or Haskell functions. We have seen how categorical semantics for the data and codata fragment can be given in a fully formal way, and we conclude this subsection by showing how this notion of extracting a distributive law is compatible with program transposition, in the sense that one obtains the *same* distributive law for either of the two programs related by transposition.

[26] I.e. $(f \triangledown g) \circ \iota_1 = f$, $(f \triangledown g) \circ \iota_2 = g$.

[27] Categorically, the objects μF and νB are actually *defined* in terms of in_F and out_B respectively, namely by the condition that (co)algebras uniquely factor through those (in the (co)algebra category, that is; see e.g. Lämmel and Rypacek (2008) for details).

```
data Expr = Num Int | Add Expr Expr

eval :: Expr → Int
eval (Num i) = i
eval (Add l r) = eval l + eval r

modn :: Expr → Int → Expr
modn (Num i) v = Num (i ‘mod‘ v)
modn (Add l r) v = Add (modn l v) (modn r v)
```

FIGURE 5.40: Haskell program corresponding to the Java program in Fig. 5.34 (Lämmel and Rypacek, 2008).

We define a consumer or producer function definition to *semantically induce a family (a set of sets) of distributive laws*

$$\{\{\lambda_{i,j} \mid j \in \{1,...,m_i\}\} \mid i \in \{1,...,n\}\}$$

iff its semantics $[\![-]\!]$ are composed from these $\lambda_{i,j}$ as defined in Fig. 5.38 and Fig. 5.39. Let $\mathcal{F}_{\text{data}}$ be a data program and $\mathcal{F}_{\text{codata}}$ be a codata program.[28]

Theorem 5.6 (Transposition — Distributive laws)**.** *When $\mathcal{F}_{data}$ and $\mathcal{F}_{codata}$ are related by* CODATA *transposition, i.e. $\mathcal{F}_{data} \leftrightarrow_{\text{CODATA}} \mathcal{F}_{codata}$ (see Definition 5.2), then*

$$\bigcup_{i\in\{1,...,n\}} \Lambda_i = \bigcup_{i\in\{1,...,n'\}} \Lambda'_i,$$

where $\Lambda_1, ..., \Lambda_n$ ($\Lambda'_1, ..., \Lambda'_{n'}$) are the sets of distributive laws semantically induced by $\mathcal{F}_{data}$ ($\mathcal{F}_{codata}$).

Proof. By the definition of the data and codata fragment semantics (cf. Fig. 5.38 and Fig. 5.39), the distributive laws semantically induced by the programs are semantically mapped from the equations which comprise the program. Since transposition only rearranges those equations, the distributive laws, like the equations, are the same for both programs. □

This means we can combine either family of distributive laws, with $\otimes$ and $\oplus$, into a single distributive law, which in both cases is the same (due to the distributivity of $\otimes$ and $\oplus$).

5.4.4 Haskell vs. data fragment in $\mathcal{PF}$

The original mutable state transposition result can perhaps be likened to some hypothetical mutable state extension of some functional language like Haskell. For the example program of Lämmel and Rypacek (2008) we saw above, the use of mutable state would be so restricted that in could be eliminated in a similar way as described for the Java subset. In fact, to present their algebraic dual to the coalgebraic program, Lämmel and Rypacek (2008) directly use Haskell; this dual program is shown

[28] Here we implicitly assume the presence of the relevant signatures and thus simply equate a program with its function declarations $\mathcal{F}$.

Value terms (trivial):
$A_{(c\ \cdot\ ...\ \cdot)}(\overrightarrow{t_v}) = c(\overrightarrow{t_v}) \qquad A_{x_i}() = x_i \qquad A_{y_i}() = y_i$

Computation terms:

$$
\begin{array}{lcl}
A'_{(f\ \cdot\ \cdot)}(t^1_v, t^2_v) & = & t^1_v \gg f(t^2_v, \bar{k}) \\
A'_{(f\ \cdot\ \cdot)}(t^1_c, t^2_v) & = & t^1_c \gg \mathtt{sh}(f(t^2_v, \bar{k})) \\
A'_{(f\ \cdot\ \cdot)}(t^1_v, t^2_c) & = & t^2_c \gg \mathtt{sh}(\bar{\mu}\{n \Rightarrow t^1_v \gg f(n, \bar{k})\}) \\
A'_{(f\ \cdot\ \cdot)}(t^1_c, t^2_c) & = & t^1_c \gg \mathtt{sh}(\bar{\mu}\{n_1 \Rightarrow t^2_c \gg \mathtt{sh}(\bar{\mu}\{n_2 \Rightarrow n_1 \gg f(n_2, \bar{k})\})\})
\end{array}
$$

(etc. for other arities)

$$A'_{return(\cdot)}(t_v) \quad = \quad t_v \gg \bar{k}$$

$$A_{\mathfrak{C}}(\overrightarrow{t}) \quad = \quad \mu\{\mathtt{sh}(\bar{k}) \Rightarrow A'_{\mathfrak{C}}(\overrightarrow{t})\}$$

for constructs $\mathfrak{C}$ forming computation terms

Function definitions:

$$\mathcal{M}(f :: T \to (T_1 \to ...(T_{n-1} \to T_n))\ \backslash\mathtt{n}\ \overrightarrow{f\ (c_i\ \overrightarrow{x_i})\ \overrightarrow{y_j}^{1\leq j\leq n} = t_i}^{i}) =$$
$$\mathbf{function}_+\ f(\overrightarrow{y_j : T_j}^{1\leq j\leq n}, \bar{k}) := \{\overrightarrow{c_i(\overrightarrow{x_i}) \Rightarrow \mathcal{M}'(t_i)}^{i}\}$$

(for value terms use $\mathcal{M}'(return(t_v)) = t_v \gg \bar{k}$ instead)

Data type definitions:

$$\mathcal{M}(\mathbf{data}\ D = \overrightarrow{c_i\ \overrightarrow{T_i}}^{i}) = \mathbf{data}_+\{\overrightarrow{c_i(\overrightarrow{T_i})}^{i}\}$$

FIGURE 5.41: Macro embedding for a Haskell-lookalike language.

Haskell	CU	$\mathcal{PF}$
$E : \tau$	$E \overset{\text{exp}}{:} N$	$E \overset{\text{prd}}{:} N$
(τ_1, τ_2) $()$	$N \& M$ $\uparrow 1$	$\mathbf{data}_-\ \mathtt{Pair}\langle A, B\rangle\{\mathtt{fst}(\overline{A}) \mid \mathtt{snd}(\overline{B})\}$ $\mathtt{Shift}\langle\mathtt{Unit}\rangle\ (\mathbf{data}_-\ \mathtt{Shift}\langle A\rangle\{\mathtt{sh}(\overline{A})\})$
$\mathtt{Either}\ \tau_1\ \tau_2$ $\mathtt{Void}$	$\uparrow(\downarrow N \oplus \downarrow M)$ $\perp$	$\mathtt{Shift}\langle\mathtt{Sum}\langle N, M\rangle\rangle$ $(\mathbf{data}_+\ \mathtt{Sum}\langle A, B\rangle\{\mathtt{inl}(A) \mid \mathtt{inr}(B)\})$ $\mathbf{data}_-\ \mathtt{Void}\{\mathtt{absurd}()\}$
$\tau \to \mathtt{Void}$	$\overset{n}{\neg} N$	$\mathbf{data}_-\ \mathtt{Neg}\langle A\rangle\{\mathtt{neg}(A)\}$
$\tau_1 \to \tau_2$	$\overset{n}{\neg} N \parr M$	$\mathbf{data}_-\ \mathtt{Fun}\langle A, B\rangle\{\mathtt{apply}(A, \overline{B})\}$

FIGURE 5.42: "Polarity Pocket Dictionary" of Zeilberger (2008b) (added third column: $\mathcal{PF}$ types).

in Fig. 5.40. But there are some differences between Haskell and the data fragment, just as the codata fragment would be less similar to the functional update Java subset when extending this subset with (non-mutable-state) features like inheritance. However, while inheritance is not among the features that $\mathcal{PF}$ attempts to capture, the difference between Haskell and the data fragment can be analyzed within the framework $\mathcal{PF}$.

As a first step, we can macro-embed a small Haskell-lookalike that is a mere notational veriation of the data fragment, as shown in Fig. 5.41, which is sufficient for the example in Fig. 5.40. The embedding acts as if Haskell's function type did not exist and thus as if $T_1 \to ...(T_{n-1} \to T_n)$ was merely its notation for a multiple argument first-order function signature. First-order functions that discriminate on the first argument as in the data fragment are sufficient for the kind of programming with algebraic data types that is dual to the restricted form of object-oriented programming which gives rise to a coalgebraic semantics view as considered by Lämmel and Rypacek (2008).

The following sketches how to get closer to actual Haskell, inspired by the "Polarity Pocket Dictionary" of Zeilberger (2008b), the Haskell part of which is repeated in Fig. 5.42 (first and second column).[29] In this little table Zeilberger shows how he considers Haskell types to correspond to **CU** types; the added third column shows the corresponding $\mathcal{PF}$ types.

In $\mathcal{PF}$, the function type is just a special kind of codata with a single destructor `apply`, as shown in the last row of Fig. 5.42 (in **CU**, combining consumer argument types of a destructor corresponds to their negative sum $⅋$, and negative producers need to be embedded into consumers using $\overset{n}{\neg}$). So in order to bring the embedding shown above closer to Haskell, we just have to add this little piece of the codata fragment[30] and properly deal with nested `apply` patterns. The resulting language can be seen as a restriction of the data and codata language of Binder et al. (2019), discussed in Section 2.4.1.

There we also saw how local matches enable nested patterns, and in this way the nested constructor patterns of Haskell can also be supported, utilizing the pragmatic extension of $\mathcal{PF}$ with local annotations discussed in Section 4.3.1. Regarding the as-patterns of Haskell ($x@(C...)$), as already mentioned in Section 4.3.2, these may be realized by using the other pragmatic extension discussed in that section.

A key feature of Haskell that we did not incorporate yet is *lazy evaluation*. Here we build on the embedding of call-by-name of Section 5.1.3 that utilizes the `Shift` type to delay computation, or the slightly more elaborate call-by-need embedding (Section 5.1.4) that builds on the call-by-name embedding and additionally makes use of daimonic commands. More specifically, we can give versions of the data fragment embedding which use these evaluation strategies instead of call-by-value, in the way the call-by-value data fragment macros generalize those of Section 5.1.1.[31] However, these would not quite give us the behavior of Haskell yet, since the flavor of CBN we considered in Section 5.1.3 always fully evaluated the pattern matched upon argument. In Haskell, the evaluation only proceeds as far as necessary for the discrimination specified by the pattern, e.g., for a shallow pattern for a sum type, it ends after having determined which of the injections was applied.

In **CU**, the type that captures this behavior is a slightly more complex construction involving shifts, seen in the fourth row from the top of Fig. 5.42: $\uparrow(\downarrow N \oplus \downarrow M)$. Here N and M are negative types, which means that expression of these types still contain (delayed) computation. Since they are negative, they can not directly be combined to a positive sum $N \oplus M$; rather, they first must be embedded into positive types using the shift $\downarrow$ from the negative to the positive side. The shift $\uparrow$ in the other direction then embeds the result back into a negative type. Intuitively, the connection to the evaluation behavior described above can probably be best understood when reading the type starting from the outside. The outermost shift tells us that we have an expression with computation that will result in a value of the sum type that the shift is applied to. But even when this computation has been carried out, there is still computation contained in the result, since it is either of type $\downarrow N$ or $\downarrow M$; it is

[29]Note that Zeilberger (2008b) warns: "These translations are provided only for intuition, and do not capture all the properties of the source languages (e.g., the purity of Haskell). Establishing a formal correspondence is far beyond the scope of this paper." The present work likewise does not attempt to establish a full formal correspondence to Haskell; nevertheless, the author hopes that fleshing out Zeilberger's ideas some more will help leverage them in the future and maybe achieve a more complete correspondence.

[30]The function codata type can then itself undergo transposition again (the original defunctionalization).

[31]To simplify things, here we work in the setting without function types; adding these and changing to lazy behavior are two rather orthogonal developments anyway.

Bodies:

$$\phi(\textbf{return } t) = \mu\{\bar{k} \Rightarrow \phi(t) \gg \bar{k}\}$$

$$\phi(\textbf{try } \{b_1\} \textbf{ catch}(\texttt{Exception } x) \ \{b_2[x]\}) = \mu\{\bar{k} \Rightarrow \mu\{\bar{k}_{\text{exc}} \Rightarrow \phi(b_1) \gg \bar{k}\} \gg \texttt{sh}(\bar{\mu}\{x \Rightarrow \phi(b_2)[x] \gg \bar{k}\})\}$$

$$\phi(\textbf{throw } t) = \mu\{_ \Rightarrow \phi(t) \gg \bar{k}_{\text{exc}}\}$$

enhance method signatures and calls with an additional continuation argument for exceptions where needed (see text)

FIGURE 5.43: Extending the Java subset embedding with exception handling.

only a value through the delaying of an expression. This corresponds to the shallow Haskell match on the sum type which does not trigger the evaluation of the injected terms.

As an example of how to implement a shallow Haskell match in $\mathcal{PF}$ following the connection to polarity just laid out, let us consider the `eval` function from the running example. As with the embedding in Fig. 5.41, this becomes a positive consumer function, but its input type is translated differently, resulting in a type `LazyExpr` following that in the fourth row of Fig. 5.42, so that we have

$$\textbf{function}_{+} \ \texttt{eval}(\bar{k} : \overline{\texttt{Int}}) : \texttt{LazyExpr} := \ldots$$

with

$$\textbf{data}_{+} \ \texttt{LazyExpr} \ \{\texttt{Num}(\texttt{Shift}\langle\texttt{Int}\rangle) \mid \texttt{Add}(\texttt{Pair}\langle\texttt{LazyExpr}, \texttt{LazyExpr}\rangle)\}.$$

The type `LazyExpr` is isomorphic to the type $\texttt{Sum}\langle N, M\rangle$ in Fig. 5.42,[32] with N instantiated to $\texttt{Shift}\langle\texttt{Int}\rangle$ and M instantiated to the negative pair type (second row of Fig. 5.42) where both type parameters are `LazyExpr`. A producer of a negative pair can be either invoked with destructor `fst` or destructor `snd`, respectively triggering evaluation of either the first or the second component and passing the result to the continuation passed as the destructor argument. The bodies of `eval` translate to

$$\texttt{Num}(n) \Rightarrow n \gg \texttt{sh}(\bar{k})$$

and

$$\texttt{Add}(p) \Rightarrow p \gg \texttt{fst}(\bar{\mu}\{e_1 \Rightarrow p \gg \texttt{snd}(\bar{\mu}\{e_2 \Rightarrow e_1 \gg \texttt{eval}(\bar{\mu}\{n_1 \Rightarrow e_2 \gg \texttt{eval}(\bar{\mu}\{n_2 \Rightarrow +(n_1, n_2, \bar{k})\})\}) \})\}).$$

In both cases the evaluation of the constructor arguments still has to be triggered in the body (so one could also ignore any of the arguments in a different implementation and they would not get evaluated at all), as desired.

5.4.5 Embedding exception handling

Let us close this section by considering a feature that can be encoded in $\mathcal{PF}$ that is present, looking at the languages that Lämmel and Rypacek (2008) use for illustration, in real Java but not in standard Haskell: exception handling. We also take a look at how exception handling constructs, as encoded along the lines presented in Section 5.3.1, are preserved when a program containing them is transposed.

[32] Note that the negative-to-positive shifts $\downarrow$ do not explicitly appear in the $\mathcal{PF}$ type, since type parameters in $\mathcal{PF}$ are polarity-agnostic.

Fig. 5.43 shows the extension of our embedding with exception handling. The only syntactic category for which a new syntactic abstraction is added are method bodies. The **try-catch** construct itself consists of two bodies b_1 and b_2; b_1 is either a **return** clause or a **throw** clause and b_2 catches the exception, bound to variable x, that is possibly thrown by b_1, either directly with **throw** or during the evaluation of the term given in the **return** clause. Of course, if b_1 does not throw an exception, the result of running the **try-catch** construct is what b_1 returns. If b_1 does throw an exception, the responsibility of producing the result falls to b_2 (which has the exception bound to x in its environment). Macro-wise, **try-catch** is implemented by nested μ-abstractions which bring two continuations into the context for b_1. With continuation $\bar{k}_{\text{exc}}$, one can pass control to the **catch** clause, while continuation $\bar{k}$ is simply the outer continuation that the entire **try-catch** returns its result to. Similarly to the macro for calling a continuation term from Section 5.3.1, a **throw** ignores its outer continuation and passes its result directly to $\bar{k}_{\text{exc}}$. For simplicity, we assume that $\bar{k}_{\text{exc}}$ is a fresh variable not used by the programmer and different from any other variable appearing in macros. Every method signature of a method which is called in an environment in which $\bar{k}_{\text{exc}}$ is available is now translated such that it does not only have the continuation argument for its result, but also a continuation argument for the "exceptional" result (which in patterns will also always be called $\bar{k}_{\text{exc}}$), inserted just before the result continuation argument.[33] In the translation of a call of such a method, $\bar{k}_{\text{exc}}$ is passed as that new argument if it is available in the environment (otherwise, a dummy continuation like $\tilde{\mu}\{_ \Rightarrow \textbf{Error}\}$ is passed instead). In our very restricted variant of exception handling using only this globally recognizable identifier $\bar{k}_{\text{exc}}$ and no exception subtypes specified for **throw** clause argument variables, a thrown exception is always caught by the next surrounding **try-catch**.

As an example, consider adding a new class `Err` as an implementation of interface `Expr`, as shown in Fig. 5.44 (modifying the example of Fig. 5.35). Both its methods directly throw exceptions, and since in the method bodies of `Add` the methods are called on the recursive subtrees, the bodies of `Add` are now changed, wrapping the calls in a **try-catch** (the **catch** clause simply returns some dummy value). Let us now look at how the result of embedding this into $\mathcal{PF}$, using the macros shown above, looks like; we focus on the body of `Add#eval` (the **try-catch**). This becomes:

$$\begin{aligned}&\mu\{\bar{k} \Rightarrow\\&\quad \mu\{\bar{k}_{\text{exc}} \Rightarrow \phi(\textbf{return}\ l.\texttt{eval}() + r.\texttt{eval}()) \gg \bar{k}\}\\&\quad \gg\\&\quad \texttt{sh}(\tilde{\mu}\{_ \Rightarrow \phi(\textbf{return}\ -1) \gg \bar{k}\})\},\end{aligned}$$

where $\phi(\textbf{return}\ l.\texttt{eval}() + r.\texttt{eval}())$ is:

$$l \gg \texttt{eval}(\bar{k}_{\text{exc}}, \tilde{\mu}\{v_1 \Rightarrow r \gg \texttt{eval}(\bar{k}_{\text{exc}}, \tilde{\mu}\{v_2 \Rightarrow +(v_1, v_2, \bar{k})\})\}).$$

After transposition with respect to `Expr`, this μ-abstraction is now found in the consumer function corresponding to destructor `eval` (which encodes the method `eval`). The signature of this consumer function has two continuation arguments, like the destructor:

$$\texttt{eval}(\overline{\texttt{Exception}}, \overline{\textbf{int}})$$

We can now eliminate the differences between this resulting program and an equivalent data fragment program that captures its algebraic programming essence as

[33] The only reason for that order is that it makes examples more legible than adding it at the end, since this argument is always a variable in calls, as opposed to the result continuation which may be specified in greater detail.

```
interface Expr { int eval(); Expr modn(); }

class Num implements Expr {...}

class Add implements Expr {
    Expr l; Expr r;
    Add(Expr l, Expr r) { this.l = l; this.r = r; }
    int eval() {
        try {
            return l.eval() + r.eval();
        } catch(Exception e) {
            return -1;
        }
    }
    Expr modn(int v0) {
        try {
            return new Add(l.modn(v0), r.modn(v0));
        } catch(Exception e) {
            return new Add(new Num(v0), new Num(v0));
        }
    }
}

class Err implements Expr {
    int eval() { throw new Exception("implementation missing"); }
    Expr modn(int v0) { throw new Exception("implementation missing"); }
}
```

FIGURE 5.44: Java example with exception handling (modified version of Fig. 5.35).

$$\phi(\textbf{case callcc}\ (\textbf{fn}\ \texttt{exit} \Rightarrow \texttt{Right}\ t_1)\ \textbf{of}\ \texttt{Left}\ _ \Rightarrow t_2 \mid \texttt{Right}\ x \Rightarrow x) \ =$$
$$\mu\{\bar{k} \Rightarrow \mu\{\bar{k}_{\text{exc}} \Rightarrow \phi(t_1) \gg \bar{k}\} \gg \texttt{sh}(\bar{\mu}\{_ \Rightarrow \phi(t_2) \gg \bar{k}\})\}$$
$$\phi(\textbf{throw}\ \texttt{exit}\ (\texttt{Left}\ ())) \ = \ \mu\{_ \Rightarrow \texttt{tt}() \gg \bar{k}_{\text{exc}}\}$$

FIGURE 5.45: Exception handling macros for an ML-lookalike.

```
open SMLofNJ.Cont
datatype ('a, 'b) either = Left of 'a | Right of 'b
datatype expr = Num of int | Add of (expr * expr) | Err
(* expr * (unit,int) either cont -> int *)
fun eval (Add (l,r),_)
    = (case callcc (fn exit ⇒ Right (eval (l,exit) + eval (r,exit)))
          of Left _ ⇒ ~1
           | Right i ⇒ i)
  | eval (Err,exit)
    = throw exit (Left ())
  | eval (Num v,_) = v
...
```

FIGURE 5.46: ML program obtained via $\mathcal{PF}$-transposing the Java program in Fig. 5.44.

outlined above, as far as the other syntactic abstractions are concerned, while pretty much keeping the syntactic abstractions utilized for Java exception handling.

More precisely, we can for instance view the data fragment as a notational variation of a first-order restriction of ML and use the macros from Fig. 5.45 to add a restricted variant of ML error handling with call/cc (Danvy and Lawall, 1992) to this ML-lookalike language.[34] These syntactic abstractions are structurally the same as those for Java exception handling (Fig. 5.43) and so is their mapping to $\mathcal{PF}$ (except for the `Exception` x, which was not used in the example, not being accessible anymore, i.e. we replace `Exception` with `Unit`). Fig. 5.46 shows the result of $\mathcal{PF}$-transposing the Java program in Fig. 5.44, macro-expressed to appear like ML.[35]

Compared to real ML error handling, the syntax forces the programmer to specify how to respond (e.g. by returning -1) to the error case globally per call/cc. Specifically, this is realized by an artificially required case discrimination for an `either` type wrapping the call/cc, wrapping the regular result with `Right` and "sending" the exception as a `Left ()`); this reflects the **catch**-clause of Java. Observe also how "sending" the exception happens by using **throw** with the `exit` continuation, which is, as mentioned above, an argument of `eval`; importantly, and differing from Java, this `exit` argument also appears in the surface language.[36]

[34] We deliberately employ call/cc instead of the simpler exception handling mechanism that also exists in ML, in order to demonstrate further processing of the running example program (together with a corresponding adaptation of the macros) to more fully use the power of call/cc (see below).

[35] This program is valid SML/NJ (http://www.smlnj.org/); tested with version 110.98.1.

[36] In SML/NJ, there is a type constructor `cont` (in module `SMLofNJ.Cont`) which corresponds to the $\mathcal{PF}$ continuation type judgment.

```
...
(* expr * (unit,int) either cont -> int *)
fun eval (Add (l,r),_)
    = callcc (fn exit ⇒ eval (l,exit) + eval (r,exit))
  | eval (Err,exit)
    = throw exit ~1
  | eval (Num v,_) = v
...
```

FIGURE 5.47: ML program adapted from that of Fig. 5.46, using call/cc more liberally for more localized error responses.

As we did with Haskell in the previous subsection, restrictions like the artificially required emulation of **catch**-clauses can be tracked in a formally precise way within $\mathcal{PF}$. Besides allowing to localize control over responding to error cases, we can also lift the first-order restriction and adapt the exception handling macros accordingly to obtain a full-fledged call/cc to which one can actually pass a first-class function (what is called call/cc in Fig. 5.43 actually just has the power of a let/cc); the formal details of this development in $\mathcal{PF}$ were discussed in Section 5.3.3. Changing the macros to support full call/cc in this way, one can then for instance restructure the program in Fig. 5.46 to locally respond to error cases, e.g. in the `Err` case throw -1 to the `exit` continuation, as shown in Fig. 5.47.

Overall, transposing a program in the functional update restriction fragment of Java that we considered, enhanced with simple exception handling, gives us a program in a language with a syntax that reuses the syntactic constructs for exception handling. We can make some superficial changes to the notation to obtain (a fragment of) some functional language that supports some form of exception handling, like ML. In summary, macro embeddings together with transposition in $\mathcal{PF}$ give us a rather straightforward path from an object-oriented language like Java to a functional language like ML, when both support exception handling. In other words, when considering a certain feature that impacts control flow in a way not present in the data and codata languages, like exception handling, we see that $\mathcal{PF}$, together with appropriate macros for that feature, conservatively extends the transposition process in the sense that the constructs of the feature itself stay as they are.

For instance, this process may serve as the building block for tools that allow to automatically analyze what some object-oriented program using exception handling, say, in Java, could have as a semantic equivalent in, say, ML. The tool could then inform the programmer of the relationship between the exception handling features and how this manifests in the programs, and, extending what was mentioned in Section 2.4.1, allow them to convert between the programs at the press of a button.

Chapter 6

Outlook: Towards Systematic and Symmetric Design

In the previous chapter, we have seen how various surface languages can be embedded into the foundational system $\mathcal{PF}$ by sets of macros. In particular, as shown in the case study in Section 5.4, this enables to systematically compare different surface languages and semantically equivalent programs specified in different languages, especially when they are related by program transposition in the sense of the Expression Problem. It also allows to factor soundness proofs into (a) demonstrating correspondence between the surface language and the image of the macro embeddings within $\mathcal{PF}$, and (b) the soundness proof for $\mathcal{PF}$, which can be reused for each surface language. Part (b) was developed in chapter 4 closely following logical intuitions about proofs, refutations, contradictions, and their term assignments. All of this would not have been possible without the large stack of prior work on transferring ideas between programming languages and logic that began with Curry-Howard. The present work's approach has been to "zoom out", trying to find a wide view to utilize this prior knowledge to better deal with the complexity of the programming language design space, specifically, by more economically exploiting the extensibility duality and the De Morgan duality. The foundational system $\mathcal{PF}$ consolidates these dualities and allows to start leveraging this knowledge for a more systematic exploration of the design space.

Of course, this work is only a tiny piece on the road to a more systematic programming language design methodology that integrates the logical perspective via the "enhanced" Curry-Howard correspondence. In part, its intention is to provide motivation for taking a more holistic approach to programming language analysis and design, systematically integrating results from various related areas, potentially helped by a common "standard" formal framework. The latter also has the potential to allow better reasoning about the design and to arrive at reasonable design prescriptions, i.e. it could lead us to a type V theory in the terminology mentioned in the introduction. Now, besides what has been discussed for $\mathcal{PF}$, there are some other more or less directly related areas of research, in the broadest sense part of the Curry-Howard program, that seem promising enough to drive forward the design methodology to consider their integration into the "standard" framework. Thus, these final pages will be devoted to briefly summarizing these areas and speculating about their potential impact and connections to be unveiled, in the spirit of what this work has attempted for the extensibility and De Morgan dualities.

Dialogues, Sessions, and Communication As was briefly hinted at, one possibility of further development of $\mathcal{PF}$ is to extend it into a kind of session type (Honda,

1993; Honda, Vasconcelos, and Kubo, 1998) system with what could be called session data and codata types. The reason for considering this as a consequential next step is the conceptual connection between polarity and the dialogical (i.e. game semantics) view (due to Lorenzen (1967)) of proofs and refutations as games between two participants, the proponent and the opponent. This dialogical interpretation subsumes the usual interpretation of classical logic propositions as just truth values, i.e. zero game moves, and that of intuitionistic logic as proof terms, i.e. only one move (Blass, 1992). In a Curry-Howard-like way, the proponent and opponent and the formulas they prove/refute correspond to a client and a server and their interface specifications, i.e. session types (Caires and Pfenning, 2010). This is a setting in which it is natural to think of resources being exchanged and in particular their *amount* being tracked, which relates this to *linear types* (based on *linear logic*, due to Girard (1987)), which Wadler prophesied to have the potential to "change the world" (Wadler, 1990). In a nutshell, a linear type system is like a substructural logic, dropping the weakening and contraction rules from classical or intuitionistic logic, which is why types/propositions cease to be *ephemeral* and give rise to a resource interpretation. Similarly to how the dialogical interpretation is more general than traditional interpretations, the interpretation of proofs as programs can be generalized to *proofs as processes* (Abramsky, 1994), with propositions corresponding to session types (Caires and Pfenning, 2010; Wadler, 2012). Research on the precise correspondence between linear logic and process calculi (Abramsky, 1994; Bellin and Scott, 1994), and, eventually, session types (Caires and Pfenning, 2010), has in particular led to systems in which the type system guarantees deadlock freedom (Wadler, 2012).

An interesting theoretical aspect that makes clear the placement of $\mathcal{PF}$ "between" practical surface languages and session type systems is the correspondence between cut elimination and communication (between processes) (Caires and Pfenning, 2010; Wadler, 2012) that goes along with "proofs as processes, propositions as session types". Cut elimination normalizes evidence for a (contingent) contradiction, which is also the logical correspondence of reduction in $\mathcal{PF}$ (cf. the evidence shrinking argument used in the soundness proof of $\mathcal{PF}$ in Section 4.2.2). Reduction in intuitionistic natural deduction based surface languages also corresponds to normalization of evidence, namely of proofs. So communication takes the place of reduction in Wadler's "change[d] [...] world" (Wadler, 1990), with the reduction in $\mathcal{PF}$ taking sort of a middle ground by normalizing commands (corresponding to contradiction evidence) instead of natural deduction based terms, but not yet truly giving rise to a communication interpretation. The "interaction" between producers and consumers is somewhat reminiscent of a dialogue and thus communication (with polarity dictating the "flow" of the communication), but still removed from being able to model interaction of processes. Finding the proper way to generalize $\mathcal{PF}$ into a system that allows to emulate processes and sessions is also of practical interest, enabling to leverage the design framework and its duality economy for *concurrent programming*, which is prevalent in the modern world.[1]

[1] Wadler (2012) remarks that "today, mobile phones, server farms, and multicores make us all concurrent programmers" and adds that (as of the time of his writing) "[m]any process calculi have emerged [...] but none is as canonical as λ-calculus, and none has the distinction of arising from Curry-Howard." The author thinks that this quest for "canonical" systems is important (and has already been significantly advanced by Wadler and others), but that the bias towards the λ-calculus (cf. e.g. Section 3.1) is not necessarily helpful for it.

Controlled Symmetry Breaking Another avenue for future work, mentioned in Section 5.3, arises from the question of how to recover *delimited* continuations in $\mathcal{PF}$. Undelimited continuations are at the conceptual heart of this system, which provides a straightforward path to the embedding of languages with undelimited first-class continuations (see Section 5.3). However, delimited continuations are reified "slices" of the whole continuation frame, which means that conceptually they are like functions, with the delimiter placed where this function outputs its result; the type of this output is called the *answer type*. Now, in $\mathcal{PF}$, as in the calculus **CU** that it is based on, we can recover first-class functions by a negative (data) type. So one option could be to use functions emulated like that for delimited continuations, but then delimited and undelimited continuations would be realized in two quite different ways. In particular, this would not reflect the idea that a delimited continuation is "sliced" from an undelimited one. A possible alternative comes from a work of Zeilberger (2010) (in a **CU**-related setting) that relates delimited continuations with deliberately breaking the symmetry between the negative and the positive polarity in a controlled way: the answer type is necessarily positive.

This follows the intuition that the purpose of the computation that is a slice of an (undelimited) continuation is to "produce some piece of observable data" (Zeilberger, 2010); positive types specify this data canonically, other than negative types, which specify the canonical ways to destruct data. Zeilberger (2010) introduces what he calls "generalized polarity", for which he moves from the classical setting, starting from a system similar to **CU** (Zeilberger, 2008b) (discussed in Section 3.3 and forming the basis for $\mathcal{PF}$), back to an intuitionistic system. Remember that intuitionistic logic is characterized by a single consequence, corresponding to the single output each term produces in "normal" surface languages. Formally, Zeilberger generalizes the command typing atom # to an *ultimate consequence* P, i.e., the (positive) answer type (that types the result of the delimited continuation). The rule for the positive negation connective, which embeds positive refutations (which conceptually correspond to undelimited continuations) into proofs (values) is accordingly parameterized by P' (P is used for the input type) such that it types delimited continuations with answer type P'.[2]

$$\frac{\forall(\Delta \Rightarrow p \overset{\text{val}}{:} P) : \qquad \Gamma, \Delta \vdash \phi(p) \overset{\text{cmd}}{:} P'}{\Gamma \vdash (\lambda\phi) \overset{\text{cnt}}{:} P \triangleright P'}$$

The symbol $\triangleright$ is part of the judgment notation; read $(\lambda\phi) \overset{\text{cnt}}{:} P \triangleright P'$ as $\lambda\phi$ is a delimited continuation with input type P and answer type P'. Crucially, the generalization of # to a consequence P' also requires to change the rule for negative negation, which types expressions (negative continuation in the terminology we used for $\mathcal{PF}$), which can be used to model laziness. In a nutshell, a negative proof of such a negated negative proposition requires to lead any contingent negative refutation to a contradiction, as in **CU**, but this must now also be *polymorphic over consequences* (Zeilberger, 2010), i.e. the proof-by-contradiction argument must work for arbitrary answer types.

We will not discuss further details of the system of Zeilberger (2010), but note that his overall résumé is "that "intuitionistic polarity" arise[s] from the same principles

[2] The rule is taken from Fig. 3 of Zeilberger (2010), slightly adapted to fit the notation of Zeilberger (2008b).

as classical polarity, but in the more general framework of delimited continuation-passing, and that this setting enriches constructive logic with delimited control operators." Viewed the other way around, we can say that to obtain a Curry-Howard-like correspondence for delimited control, we can generalize the classical setting to an intuitionistic one in a principled way. Doing so gives up on the elegant symmetry of classical logic, but in a controlled way that carves out the essence of "delimiting". Speculating on further development of $\mathcal{PF}$, this could mean that it might be helpful to add a meta-layer to the system which allows to analyze certain generalizations like that of command typing. It is an open question of whether this also indicates that this meta-layer should even allow to derive $\mathcal{PF}$ and $\mathcal{PF}$-related, but constructive systems by instantiating it somehow, and, if yes, if and how to combine this approach with the macro embedding approach discussed in the present work. The more fundamental, dialogical approach to "constructive-ness" discussed above may be instrumental in better understanding this. As a final remark, the logical interpretation of answer types as being related to positive polarity also seems to beg the question of whether there is a somehow dual correspondence: Is there something like a "question type" related to negative polarity?[3] Adding a meta-layer to the framework might facilitate systematically discussing questions such as this one.

Extensibility Symmetry for the Lambda Cube As mentioned in Section 2.4.2, other quite obviously interesting extensions of $\mathcal{PF}$ could be generalizations of the system along the other dimensions of the lambda cube (Barendregt, 1991) besides the parametric polymorphism one (terms depending on types; already realized in Section 4.4), i.e. for types depending on terms and types depending on types. Such a generalization would have to be done in a way that upholds the extensibility symmetry (which can be made technically precise as program transposition being a single, polarity-polymorphic transformation). Just as this required GA(Co)DTs in the case of parametric polymorphism, for the other lambda cube dimensions it is likewise to be expected that finding these symmetric spots may require more general variants of type declaration mechanisms than standardly considered.

Logics for Modularity Remember that we started our analysis of the extensibility duality with a common goal of programmers and logicians: *reuse* of programs/proofs (cf. Section 2.1.4). With data and codata one can structure the program in two different ways (cf. also the notions of algebraic and coalgebraic programming discussed in Section 5.4.3), which also corresponds to two different reuse facilities. With both consumer and producer functions one can abstract over a certain pattern, and they both are the smallest reusable unit of the respective approach, and they provide a well-defined interface for instantiating the pattern. On another conceptual level, a (co)data type declaration is itself a kind of an "instantiable" unit of the program, in the sense that it prescribes the (interface) structure that the function units have to adhere to, and which can hence be relied on at their usage sites (like a consumer function being able to be applied to any constructor call of its data type). A general term for such a unit is *module*. The characteristics of a module system include that it allows to structure the program into smaller pieces to help control its complexity and that modules refer to each other by well-defined interfaces; usually some kind of *information hiding* is also involved, where modules have a *private* part and a *public* one which is often only their interface (but see below). Modularity is an important aspect of programming languages that this work, and hence the system $\mathcal{PF}$,

[3] This idea is due to Ingo Skupin (personal communication).

does not attempt to cover in its generality, beyond the specific instances of consumer and producer functions. However, due to the practical relevance and the relation to foundational concepts behind $\mathcal{PF}$ just sketched, let us take a (speculative) look at how research on modularity, particularly if it likewise attempts to port ideas from the realm of logic, could be integrated in the further development of a foundational system based on $\mathcal{PF}$.

But before we start, it is worth reconsidering the nature of transfer between the realms of logic and programming languages. Throughout this work, we have only considered the kind of transfer that is, in essence, the same as that of the original Curry-Howard correspondence. That is, we were always considering program terms, in the broadest sense, as corresponding to evidence (e.g. proofs, refutations, contradictions), and as a natural consequence along with this the correspondence between types, in the broadest sense, and propositions (or more generally logical formulae; possibly also distinguishing different judgments). But that is not the only conceivable way how results from logic can inform programming language research. For instance, we can model the *semantics* of a program in some form of logic, or more generally some formal system. We have seen an example of this when demonstrating the compatibility of program transposition and the Expression Lemma, which is a semantic categorical statement, i.e. the relevant formal system for modelling the semantics is *category theory*. When we want to examine whether a program fulfills its intended purpose, it can instead often be useful to work with (some kind of) *first-order predicate logic*, in which we can state some property to be verified, e.g. the commutativity of some function.

Research on such forms of *knowledge representation* informs programming language research when it comes to how to best design a language and its module system to facilitate *modular reasoning*; for instance, how to enable the programmer to easily verify the correctness of some module assuming that the modules is references are correct. Regarding transfer from the realm of logic, it has been noted by Ostermann et al. (2011) that while classical logic has informed traditional approaches to modularity, like functional abstraction and abstract data types, this may not be the best perspective for (somewhat) more recent developments like aspect-oriented programming (AOP). Specifically, information hiding is deliberately weakened in these approaches for practical reasons, for instance to facilitate the modularization of *cross-cutting concerns* like, to cite a classic AOP example, signaling that a display update has occurred (Kiczales and Mezini, 2005). Ostermann et al. (2011) point out that the underlying discrepancy is that programmers, for concrete practical benefit, do not reason classically (in the sense of classical logic); they forego classical rules of reasoning, like *monotonicity* of entailment, i.e. that a truth of some statement cannot be invalidated by adding further axioms (assuming they preserve consistency, of course). Rather, programmers often operate with the understanding that a certain property, considered in the context of some module, may be invalidated by a change within a different module, for instance for some cross-cutting concern, contrary to the principles of information hiding and classical reasoning. In summary, a certain awareness of implementation details of other modules is arguably not something that can always be avoided. To still be able to reasonably control the complexity of reasoning about properties of some system, upholding some degree of modular reasoning, Ostermann et al. (2011) propose to turn to non-classical logic, for instance non-monotonic logic, for guidance on the proper kind of modularity for programming.

The author thinks that a design framework for programming languages, like $\mathcal{PF}$, if its design prescriptions are to truly be of practical relevance (especially for large

code bases where modularization is essential), generally needs to integrate this insight that classical reasoning is often not sufficient. But there is also a more direct relation to $\mathcal{PF}$ that should be kept in mind when further developing it. As hinted at, this system already contains a rudimentary "module system", allowing to write new "modules" in the form of producer or consumer functions, together with (co)data type declarations. In Section 2.4.1, it was argued that languages with both data and codata that allow the programmer to choose how to decompose a part of a program into modules are an improvement over existing languages which force or strongly encourage a certain decomposition, like OO languages which inherently favor the producer decomposition (classes) or many functional languages which inherently favor the consumer function decomposition. The general situation in many existing languages is referred to by Tarr et al. (1999) as the *tyranny of the dominant decomposition*, regarding which Ostermann et al. remark that the "best" decomposition depends on the "points of view of the different stakeholders" (Ostermann et al., 2011). The choice of decomposition also entails an "information hiding policy" (Ostermann et al., 2011).[4] Languages with symmetric data and codata at least allow the choice of different decompositions per type (one can either use a data or a codata type), so one could say that different "stakeholders" can get their way with at least part of the program, following their desired information hiding policy. Data types make public the available producers and in a sense keep private the consumers, i.e., they do not guarantee which consumers exist and which do not, allowing to, e.g., independently add new ones, and vice versa for codata types.

However, even when one stakeholder prefers the data type decomposition and the other the codata type decomposition, symmetric data and codata in one language can have a concrete benefit. Under the assumption that at some points during the development process the whole program (or rather the part for the relevant data/codata type) is available, program (matrix) transposition allows to switch the decomposition, giving priority to one stakeholder's preference at certain times and to the other stakeholder's at other times. In the interpreter example of Section 4.3 and Section 4.4, one stakeholder could start writing, using codata, the interpreter in the meta-circular style, since they are not concerned with low-level implementation details; transposition gives these for free by "inventing" a closure data type and then a different stakeholder who is knowledgeable about such details can refine the program, working in this data type decomposition. This a rather small and theoretical example from the field of programming languages itself, but it is conceivable that comparable situations arise in different contexts and for larger code bases. Being able to easily implement program transposition depends crucially on the symmetry between data and codata, so with this argument for the relevance of program transposition we also have additional support for the importance of this symmetry, besides general design parsimony.

Closing Words This ends the speculative outlook on incorporating related research into the design framework. It is the impression of the author that a lot of research effort that in the broadest sense affects the design of programming languages has been and still is invested into many of these particular areas, advancing them individually. There is nothing wrong with that, but it is worth pointing out that striving

[4] In the words of Ostermann et al. (2011): "What one stakeholder would hide as an implementation detail behind an interface is of primary importance to another stakeholder, who would hence choose a different decomposition that exposes that information."

to keep within sight of related fields is likewise important. The author's central proposal is hence to systematically develop a framework that is able to support this endeavor, especially uncovering connections that have the potential to cut down the problem space. An important yardstick for this framework is that it should incorporate known symmetries and other "economic" knowledge; it should also stay connected to the problems programmers face in practice. Such a foundational system can never be quite finished, and will not even be able to cover all aspects of programming language design known at a given time. Nevertheless, the present work has attempted to motivate the utility of such a system. It only had the limited goal of developing a parsimonious unified framework for the extensibility and De Morgan dualities, but, considering this, it is able to recover a rather wide variety of linguistic features, demonstrating that from the proper perspective the design space does not seem quite so vast any more. Arguably, this more holistic perspective is already more prevalent in the study of logic and formal systems, so the transfer of ideas between programming languages and logic that began with Curry-Howard will continue to be a great asset. Making use of the wider Curry-Howard correspondence, while trying to take a more holistic approach to the analysis and design of programming languages can sustainably transform programming languages and with them the everyday work of software developers.

Appendix A

Lemmas for Soundness Proofs

A.1 Logical core

The following lemmas are for the logical core of $\mathcal{PF}$.

Lemma A.1 (Term substitution). *For all producers p, substitutions σ and argument contexts Ξ, if $\Xi \vdash_\Sigma p \overset{prd}{:} T$ and $\vdash_\Sigma \sigma : \Xi$, then $\vdash_\Sigma p\sigma \overset{prd}{:} T$. And analogously for consumers.*

Proof. By simultaneous structural induction over producers and consumers. We show just the case for a producer p, the consumer case is analogous. It is $p = \mathsf{X}\sigma'$ for some X and some substitution σ'. By inversion, we know $\Xi \vdash_\Sigma \sigma' : \Xi'$ for a certain Ξ' as given in the signature of X, and we need to show $\vdash_\Sigma \sigma'\sigma : \Xi'$ for that same Ξ'. Now σ' is a list of assignments from variables to terms (producers or consumers), say $\overrightarrow{x_i \mapsto t_i}^i$, and Ξ' is a list of types assigned to variables, say $\overrightarrow{x_i \overset{\mathcal{J}_i}{:} T_i}^i$. To show $\vdash_\Sigma \sigma'\sigma : \Xi'$ we need to show $\vdash_\Sigma t_i\sigma \overset{\mathcal{J}_i}{:} T_i$ for all the i. By inversion, we know $\Xi \vdash_\Sigma t_i \overset{\mathcal{J}_i}{:} T_i$, and thus we have the desired term typing by the induction hypothesis. □

Lemma A.2 (Command substitution). *For all commands C, substitutions σ and argument contexts Ξ, if $\Xi \vdash_\Sigma C \overset{cmd}{:} \#$ and $\vdash_\Sigma \sigma : \Xi$, then $\vdash_\Sigma C\sigma \overset{cmd}{:} \#$.*

Proof. We know $C = p \gg c$ for some producer p and consumer c, and by inversion $\Xi \vdash_\Sigma p \overset{\text{prd}}{:} T$ and $\Xi \vdash_\Sigma c \overset{\text{cns}}{:} T$, for some T. Thus, by the term substitution lemma, $\vdash_\Sigma p\sigma \overset{\text{prd}}{:} T$ and $\vdash_\Sigma c\sigma \overset{\text{cns}}{:} T$. Since $C\sigma = p\sigma \gg c\sigma$, we have the desired conclusion by T-CMD. □

A.2 Polymorphic logical core

The following lemmas are for the logical core of the polymorphic extension of $\mathcal{PF}$.

Lemma A.3 (Term substitution). *For all producers p, substitutions σ and argument contexts Ξ, if $\Xi \vdash_\Sigma p \overset{prd}{:} T$ and $\vdash_\Sigma \sigma : \Xi$, then $\vdash_\Sigma p\sigma \overset{prd}{:} T$. And analogously for consumers.*

Proof. By simultaneous structural induction over producers and consumers, virtually identically to the proof of Lemma A.1. We again only consider a producer p (consumer analogous). It is $p = \mathsf{X}\langle\overrightarrow{T'}\rangle\sigma'$ for some X and $\overrightarrow{T'}$ and some substitution σ'. By inversion, we know $\Xi \vdash_\Sigma \sigma' : \Xi'[\overrightarrow{A \mapsto T'}]$ for a certain Ξ' and $\overrightarrow{A}$ as given in the signature of X, and we need to show $\vdash_\Sigma \sigma'\sigma : \Xi'[\overrightarrow{A \mapsto T'}]$ for that same Ξ'

and $\overrightarrow{A}$. Now σ' is a list of assignments from variables to terms (producers or consumers), say $\overrightarrow{x_i \mapsto t_i}^i$, and $\Xi'' := \Xi'[\overrightarrow{A \mapsto T'}]$ is a list of types assigned to variables, say $\overrightarrow{x_i \overset{\mathcal{J}_i}{:} T_i}^i$. To show $\vdash_\Sigma \sigma'\sigma : \Xi''$ we need to show $\vdash_\Sigma t_i\sigma \overset{\mathcal{J}_i}{:} T_i$ for all the i. By inversion, we know $\Xi \vdash_\Sigma t_i \overset{\mathcal{J}_i}{:} T_i$, and thus we have the desired term typing by the induction hypothesis. □

Lemma A.4 (Command substitution)**.** *For all commands C, substitutions σ and argument contexts Ξ, if $\Xi \vdash_\Sigma C \overset{cmd}{:} \#$ and $\vdash_\Sigma \sigma : \Xi$, then $\vdash_\Sigma C\sigma \overset{cmd}{:} \#$.*

Proof. Identical to the proof of Lemma A.2, since type parameters do not appear in rule T-CMD. □

Lemma A.5 (Type instantiation invariance)**.** *For all maps τ from type variables to types, if $\Xi \vdash C \overset{cmd}{:} \#$, then also $\Xi\tau \vdash C\tau \overset{cmd}{:} \#$.*

Proof. We know $C = p \gg c$ for some producer p and consumer c, and by inversion $\Xi \vdash_\Sigma p \overset{\text{prd}}{:} T$ and $\Xi \vdash_\Sigma c \overset{\text{cns}}{:} T$, for some T. By simultaneous structural induction over producers and consumers we show that producers and consumers themselves are type instantiation invariant, i.e. that, for arbitrary T, $\Xi \vdash_\Sigma p \overset{\text{prd}}{:} T$ and $\Xi \vdash_\Sigma c \overset{\text{cns}}{:} T$ imply $\Xi\tau \vdash_\Sigma p\tau \overset{\text{prd}}{:} T\tau$ and $\Xi\tau \vdash_\Sigma c\tau \overset{\text{cns}}{:} T\tau$; from this we get the desired command typing for $C\tau = p\tau \gg c\tau$ by T-CMD. The inductive proof proceeds in the same manner: use inversion and apply the induction hypothesis to draw τ over the typings of the producers and consumers in the substitution that constitutes p or c. □

Appendix B

Proof Details for Reduction Correspondence

Proof for Lemma 5.1. It is

$$t_1 = f@(n,m), t_2 = \pi_i(\mathcal{D}(f))[y \mapsto m, [x_S \mapsto pred(n)]],$$

where the x_S substitution only happens if $i = 2$, i.e. n is a successor of some number and $pred(n)$ refers to that number. It follows that:

$$\begin{aligned} & \mathcal{M}'(t_1)[k_0] \\ = \ & n \gg f(m, k_0) \\ \triangleright \ & C_i[m \mapsto m, k \mapsto k_0, [n \mapsto pred(n)]] \\ = \ & \mathcal{M}'(t_2)[k_0] \end{aligned}$$

Here C_i is the command in the relevant case of the translated body of f (the zero case for $i = 1$ and the successor case for $i = 2$), so the last equation holds definitionally; cf. how we defined the translation of function bodies. □

Proof for Lemma 5.2. There are two possible rules with which $t_1 \triangleright_{\mathrm{v}} t_2$ could have been derived.

For the first, we know that $t_1 = \mathcal{E}'[t_1'], t_2 = \mathcal{E}'[t_2'], t_1' \blacktriangleright_{\mathrm{v}} t_2'$. In particular, this means that the possible machine configurations $c_1 \fallingdotseq t_1$ are of the form $c_1 = (\mathcal{E}, \mathcal{E}_0[t_1'])$, where $\mathcal{E}[\mathcal{E}_0] = \mathcal{E}'$. We continue by induction on the structure of $\mathcal{E}_0$.

- $\mathcal{E}_0 = []$. It follows that $c_1 = (\mathcal{E}, t_1') \rightsquigarrow_{(1)} (\mathcal{E}, t_2') \fallingdotseq t_2$.
- $\mathcal{E}_0 = \mathcal{E}_0'[\mathcal{E}_1]$, where $\mathcal{E}_0'$ has depth 1. It follows that

  $$c_1 = (\mathcal{E}, \mathcal{E}_0'[\mathcal{E}_1[t_1']]) \rightsquigarrow_{(3)} (\mathcal{E}[\mathcal{E}_0'], \mathcal{E}_1[t_1']) =: c_1' \fallingdotseq t_1.$$

  By the induction hypothesis we have the remaining configurations $c_2', ..., c_k' \fallingdotseq t_1$ and $c_2 \fallingdotseq t_2$ with $c_1' \rightsquigarrow c_2' \rightsquigarrow ... \rightsquigarrow c_k' \rightsquigarrow c_2$.

Finally, we have to consider the second evaluation step rule, for which we know that $t_1 = \mathcal{E}'[return(t_v)], t_2 = \mathcal{E}'[t_v]$, where $\mathcal{E}'$ has depth ≥ 1. For the machine configuration $c_1 \fallingdotseq t_1$ this means that its form is $c_1 = (\mathcal{E}, \mathcal{E}_0[return(t_v)])$ with $\mathcal{E}[\mathcal{E}_0] = \mathcal{E}'$. Similarly to before, we finish the proof by induction on $\mathcal{E}_0$.

- $\mathcal{E}_0 = []$. It follows that there is some $\mathcal{E}^1$ and some single frame $\mathcal{E}^2$ with $\mathcal{E}^1[\mathcal{E}^2] = \mathcal{E} = \mathcal{E}[[]] = \mathcal{E}'$, such that $c_1 = (\mathcal{E}', return(t_v)) \rightsquigarrow_{(2)} (\mathcal{E}^1, \mathcal{E}^2[t_v]) \fallingdotseq t_2$.
- $\mathcal{E}_0 = \mathcal{E}_0'[\mathcal{E}_1]$, where $\mathcal{E}_0'$ has depth 1. It follows that

  $$c_1 = (\mathcal{E}, \mathcal{E}_0'[\mathcal{E}_1[return(t_v)]]) \rightsquigarrow_{(3)} (\mathcal{E}[\mathcal{E}_0'], \mathcal{E}_1[return(t_v)]) =: c_1' \fallingdotseq t_1.$$

  By the induction hypothesis we have the remaining configurations $c_2', ..., c_k' \fallingdotseq t_1$ and $c_2 \fallingdotseq t_2$ with $c_1' \rightsquigarrow c_2' \rightsquigarrow ... \rightsquigarrow c_k' \rightsquigarrow c_2$. □

Proof for Theorem 5.1, rule (1). For rule (1), contraction in a context, the computational gloss for the desired $\mathcal{PF}$ reduction

$$\mathcal{T}(c_1) = \mathcal{M}'(t_1)[\mathcal{T}(\mathcal{E})] \triangleright \mathcal{T}(c_2) = \mathcal{M}'(t_2)[\mathcal{T}(\mathcal{E})],$$

with $t_1 \blacktriangleright_{\mathrm{v}} t_2$, is: "Running the computation t_1 while keeping track of some remaining computation leads to the computation t_2 with the same remaining computation to keep track of." Intuitively, we have to show that the translation of each possible redex is compatible with this gloss, i.e., that the remaining computation is carried along unchanged while contracting the redex. But this is easy since our configuration translation is exactly aligned with the universal-quantification-like abstraction over the remaining computation that underlies the notion of contraction correspondence employed in Lemma 5.1. That is, our remaining computation $\mathcal{T}(\mathcal{E})$ is instantiated for the opaque, inaccessible, universally quantified continuation and we can just directly use Lemma 5.1:

$$\mathcal{T}(c_1) = \mathcal{M}'(t_1)[\mathcal{T}(\mathcal{E})] \overset{\text{Lem. 5.1}}{\triangleright} \mathcal{M}'(t_2)[\mathcal{T}(\mathcal{E})] = \mathcal{T}(c_2) \qquad \square$$

Proof for Theorem 5.1, rule (2). For rule (2), recomposition with a context, the computational gloss for the desired $\mathcal{PF}$ reduction step from

$$\mathcal{T}(c_1) = \mathcal{M}'(\mathit{return}(v))[\mathcal{T}(\mathcal{E}_0[\mathcal{E}'])] = v \gg \mathcal{T}(\mathcal{E}_0[\mathcal{E}']$$

to

$$\mathcal{T}(c_2) = \mathcal{M}'(\mathcal{E}'[v])[\mathcal{T}(\mathcal{E}_0)],$$

where $\mathcal{E}'$ has depth 1, is: "Running a value embedded as a computation with some remaining computation, corresponding to a stack, kept track of leads to the continuation that corresponds to the innermost part of that stack applied to the value, while keeping track of the initial segment of the stack." We observe that this is a direct consequence of Lemma 5.4, which is a variation of the definitional equation for $\mathcal{T}$ for instantiation with values.

$$\mathcal{T}(c_1) = v \gg \mathcal{T}(\mathcal{E}_0[\mathcal{E}']) \triangleright \mathcal{T}'(\mathcal{E}_0[\mathcal{E}'])[v] \overset{\text{Lem. 5.4}}{=} \mathcal{M}'(\mathcal{E}'[v])[\mathcal{T}(\mathcal{E}_0)] = \mathcal{T}(c_2) \qquad \square$$

Proof for Theorem 5.1, rule (3). For rule (3), redex search, or accumulation of the context, the computational gloss for the desired $\mathcal{PF}$ reduction step

$$\mathcal{T}(c_1) = \mathcal{M}'(\mathcal{E}'[t_c])[\mathcal{T}(\mathcal{E})] \triangleright \mathcal{T}(c_2) = \mathcal{M}'(t_c)[\mathcal{T}(\mathcal{E}[\mathcal{E}'])],$$

where $\mathcal{E}'$ has depth 1, is: "A computation that is some other computation followed by some one-step continuation while keeping track of remaining computation leads to that other computation with the one-step continuation followed by the previous remaining computation kept track of." We use Lemma 5.3 (1), which relates the translation of a single stack frame (= one-step continuation) (like $\mathcal{E}'$) to the translation of the term obtained by appending the stack frame at the end (obtaining e.g. $\mathcal{E}'[t_c]$), to obtain:

$$\mathcal{T}(c_1) = \mathtt{sh}(\mathcal{M}^{\text{ctx}}(\mathcal{E}')[\mathcal{T}(\mathcal{E})]) \gg \mathcal{M}(t_c)$$

Now we know what $\mathcal{T}(c_1)$ reduces to and can conclude:

$$\mathcal{T}(c_1) \triangleright \mathcal{M}'(t_c)[\mathcal{M}^{\text{ctx}}(\mathcal{E}')[\mathcal{T}(\mathcal{E})]] = \mathcal{M}'(t_c)[\mathcal{T}(\mathcal{E}[\mathcal{E}'])] = \mathcal{T}(c_2) \qquad \square$$

Appendix C

Reduction Correspondence for let/cc

The step relation for the evaluation machine for the simple first-order CBV language, i.e.

(A.1)	$(\mathcal{E}, t)$	$\rightsquigarrow$	$(\mathcal{E}, t')$	if $t \blacktriangleright_v t'$
(A.2)	$(\mathcal{E}, \mathit{return}(v))$	$\rightsquigarrow$	$(\mathcal{E}_0, \mathcal{E}'[v])$	if $\mathcal{E} = \mathcal{E}_0[\mathcal{E}']$, $\mathcal{E}'$ has depth 1
(A.3)	$(\mathcal{E}, \mathcal{E}'[t_c])$	$\rightsquigarrow$	$(\mathcal{E}[\mathcal{E}'], t_c)$	if $\mathcal{E}'$ has depth 1,

is extended as follows for the let/cc and call constructs:

(B.1)	$(\mathcal{E}, (\text{let/cc}\ k.t[k]))$	$\rightsquigarrow$	$(\mathcal{E}, t[\text{cont}(\mathcal{E})])$
(B.2)	$(\mathcal{E}, \text{call}(\text{cont}(\mathcal{E}'), v))$	$\rightsquigarrow$	$(\mathcal{E}', \mathit{return}(v))$

Using the same definition of term associated to machine configuration as for the first-order CBV language (i.e. $(\mathcal{E}, t) \fallingdotseq \mathcal{E}[t]$), we show that Lemma 5.2 also holds for the first-order CBV language with let/cc.

Lemma C.1. *For any two terms t_1, t_2 with $t_1 \rhd_v t_2$ and any machine configuration $c_1 \fallingdotseq t_1$, there are configurations $c'_1, ..., c'_k \fallingdotseq t_1$ ($k \geq 0$) and $c_2 \fallingdotseq t_2$ such that: $c_1 \rightsquigarrow c'_1 \rightsquigarrow ... \rightsquigarrow c'_k \rightsquigarrow c_2$.*

Proof. We distinguish the four possible rules by which the reduction could have been derived. For two of those, which are already present in the language without let/cc (i.e. the ones for contraction in a context and for *return*) we finish the proof exactly as for Lemma 5.2.

For the reduction rule for let/cc, we know that $t_1 = \mathcal{E}'[(\text{let/cc}\ k.t[k])]$ and $t_2 = \mathcal{E}'[t[\text{cont}(\mathcal{E}')]]$. In particular, this means that the possible machine configurations $c_1 \fallingdotseq t_1$ are of the form $(\mathcal{E}, \mathcal{E}_0[(\text{let/cc}\ k.t[k])])$, where $\mathcal{E}[\mathcal{E}_0] = \mathcal{E}'$. We continue by induction on the structure of $\mathcal{E}_0$, where for the inductive case we proceed in the same way as for the other reduction rule cases. In the base case, i.e. $\mathcal{E}_0 = []$, we have: $c_1 = (\mathcal{E}, (\text{let/cc}\ k.t[k])) \rightsquigarrow (\mathcal{E}, t(\text{cont}(\mathcal{E}))) \fallingdotseq t_2$.

Finally, for the reduction rule for a continuation call, we know that

$$t_1 = \mathcal{E}'[\text{call}(\text{cont}(\mathcal{E}''), v)], t_2 = \mathcal{E}''[\mathit{return}(v)],$$

as well as $c_1 = (\mathcal{E}', \mathcal{E}_0[\text{call}(\text{cont}(\mathcal{E}''), v)])$, where $\mathcal{E}[\mathcal{E}_0] = \mathcal{E}'$. We finish by induction on the structure of $\mathcal{E}_0$, where we again only need to consider the base case $\mathcal{E}_0 = []$:

$$c_1 = (\mathcal{E}, \text{call}(\text{cont}(\mathcal{E}''), v)) \rightsquigarrow (\mathcal{E}'', \mathit{return}(v)) \fallingdotseq t_2$$ □

To conclude, we prove the correspondence theorem for the language with let/cc, where $\mathcal{T}$ is the translation for machine configurations defined in the same way as for the language without let/cc (see Fig. 5.5).

Theorem C.1. *For any two machine configurations with $c_1 \rightsquigarrow c_2$ it is $\mathcal{T}(c_1) \rhd \mathcal{T}(c_2)$.*

Just as with the correspondence theorem for the simple language without let/cc (Theorem 5.1), this can be proven by separately considering the rules for $\rightsquigarrow$. For the rules (A.1-3) taken over from the language without let/cc, the proofs are identical[1] ; thus we only need to consider the new rules (B.1) and (B.2).

Proof for Theorem C.1, rule (B.1). By inversion we know the form of c_1, and it is:

$$\mathcal{T}(c_1) = \mathcal{M}'((\text{let/cc } k.t[k]))[\mathcal{T}(\mathcal{E})] = (\mathcal{M}(t)[\bar{k}] \gg \texttt{sh}(\bar{k}))[\mathcal{T}(\mathcal{E})] \triangleright \mathcal{M}'(t)[\mathcal{T}(\mathcal{E})]$$

and

$$\mathcal{T}(c_2) = \mathcal{M}'(t[\text{cont}(\mathcal{E})])[\mathcal{T}(\mathcal{E})] = \mathcal{M}'(t)[\mathcal{M}(\text{cont}(\mathcal{E}))] = \mathcal{M}'(t)[\mathcal{T}(\mathcal{E})]. \qquad \square$$

Proof for Theorem C.1, rule (B.2). By inversion we know the form of c_1, and it is:

$$\mathcal{T}(c_1) = \mathcal{M}'(\text{call}(\text{cont}(\mathcal{E}'), v))[\mathcal{T}(\mathcal{E})] = \mathcal{M}(v) \gg \mathcal{T}(\mathcal{E}')$$

and

$$\mathcal{T}(c_2) = \mathcal{M}'(\textit{return}(v))[\mathcal{T}(\mathcal{E}')] = \mathcal{M}(v) \gg \mathcal{T}(\mathcal{E}'). \qquad \square$$

[1] As is easily checked, the auxiliary lemmas (Lemma 5.3, Lemma 5.4) used in these proofs also hold for the extended specification of evaluation contexts, with the continuation call context translated as follows: $\mathcal{M}^{\text{ctx}}(\text{call}(t_k, [])) := \tilde{\mu}\{n \Rightarrow n \gg t_k\}$.

Bibliography

Abel, Andreas, Brigitte Pientka, David Thibodeau, and Anton Setzer (2013). "Copatterns: Programming infinite structures by observations". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '13)*. New York, NY, USA: ACM, pp. 27–38.

Abramsky, Samson (1994). "Proofs as processes". In: *Theoretical computer science* 135.1, pp. 5–9.

Andreoli, Jean-Marc (2001). "Focussing and proof construction". In: *Annals of pure and applied logic* 107.1-3, pp. 131–163.

Ariola, Zena M. and Paul Downen (2020). "Compiling With Classical Connectives". In: *Logical methods in computer science* 16.3, 13:1–13:57.

Aschieri, Federico and Francesco A. Genco (2019). "Par Means Parallel: Multiplicative Linear Logic Proofs as Concurrent Functional Programs". In: *Proceedings of the ACM on Programming Languages* 4.POPL, 18:1–18:28.

Barendregt, Henk (1991). "Introduction to generalized type systems". In: *Journal of functional programming* 1.2, pp. 125–154.

Bellin, Gianluigi and Philip J. Scott (1994). "On the π-calculus and linear logic". In: *Theoretical computer science* 135.1, pp. 11–65.

Binder, David, Julian Jabs, Ingo Skupin, and Klaus Ostermann (2019). "Decomposition diversity with symmetric data and codata". In: *Proceedings of the ACM on Programming Languages* 4.POPL, 30:1–30:28.

Blass, Andreas (1992). "A game semantics for linear logic". In: *Annals of pure and applied logic* 56.1-3, pp. 183–220.

Caires, Luís and Frank Pfenning (2010). "Session types as intuitionistic linear propositions". In: *Proceedings of the 21st international conference on concurrency theory (CONCUR 2010)*. Springer Berlin Heidelberg, pp. 222–236.

Cartwright, Robert and Matthias Felleisen (1994). "Extensible denotational language specifications". In: *International Symposium on Theoretical Aspects of Computer Software 1994*. Springer Berlin Heidelberg, pp. 244–272.

Cook, William R. (1990). "Object-oriented programming versus abstract data types". In: *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer Berlin Heidelberg, pp. 151–178.

Curien, Pierre-Louis and Hugo Herbelin (2000). "The duality of computation". In: *Proceedings of the fifth ACM SIGPLAN international conference on functional programming (ICFP '00)*. New York, NY, USA: ACM, pp. 233–243.

Curry, Haskell B. (1934). "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11, pp. 584–590.

Danvy, Olivier (1994). "Back to direct style". In: *Science of computer programming* 22.3, pp. 183–195.

Danvy, Olivier, Jacob Johannsen, and Ian Zerny (2011). "A walk in the semantic park". In: *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '11)*. New York, NY, USA: ACM, pp. 1–12.

Danvy, Olivier and Julia L. Lawall (1992). "Back to direct style II: First-class continuations". In: *Proceedings of the 1992 ACM conference on LISP and functional programming*. New York, NY, USA: ACM, pp. 299–310.

Danvy, Olivier and Kevin Millikin (2009). "Refunctionalization at work". In: *Science of computer programming* 74.8, 534—549.

Danvy, Olivier and Lasse R. Nielsen (2001). "Defunctionalization at work". In: *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP '01)*. New York, NY, USA: ACM, pp. 162–174.

Downen, Paul (2017). "Sequent Calculus: A Logic and a Language for Computation and Duality". PhD thesis. University of Oregon.

Downen, Paul and Zena M. Ariola (2014). "The duality of construction". In: *Programming languages and systems: 23rd European symposium on programming (ESOP 2014)*. Springer Berlin Heidelberg, pp. 249–269.

Downen, Paul, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones (2016). "Sequent calculus as a compiler intermediate language". In: *Proceedings of the 21st ACM SIGPLAN international conference on functional programming (ICFP '16)*. New York, NY, USA: ACM, pp. 74–88.

Downen, Paul, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones (2019). "Codata in action". In: *Programming Languages and Systems: 28th European Symposium on Programming (ESOP 2019)*. Springer, Cham, pp. 119–146.

Dummett, Michael (1991). *The logical basis of metaphysics. The William James lectures, 1976*. Harvard University Press.

Ernst, Erik, Klaus Ostermann, and William R. Cook (2006). "A virtual class calculus". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*. New York, NY, USA: ACM, pp. 270–282.

Felleisen, Matthias (1991). "On the expressive power of programming languages". In: *Science of computer programming* 17.1-3, pp. 35–75.

Felleisen, Matthias and Daniel P. Friedman (1987). "Control operators, the SECD-machine, and the λ-calculus". In: *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts*. North-Holland, pp. 193–222.

Felleisen, Matthias and Robert Hieb (1992). "The revised report on the syntactic theories of sequential control and state". In: *Theoretical computer science* 103.2, pp. 235–271.

Gentzen, Gerhard (1935). "Untersuchungen über das logische Schließen. I". In: *Mathematische Zeitschrift* 39.1, pp. 176–210.

Girard, Jean-Yves (1987). "Linear logic". In: *Theoretical computer science* 50.1, pp. 1–101.

— (2001). "Locus solum: From the rules of logic to the logic of rules". In: *Mathematical structures in computer science* 11.3, pp. 301–506.

Gregor, Shirley (2006). "The Nature of Theory in Information Systems". In: *MIS Quarterly* 30.3, pp. 611–642.

Griffin, Timothy G. (1989). "A formulae-as-type notion of control". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 47–58.

Hagino, Tatsuya (1989). "Codatatypes in ML". In: *Journal of symbolic computation* 8.6, pp. 629–650.

Harper, Robert (2016). *Practical foundations for programming languages*. Cambridge University Press.

Herbelin, Hugo (2005). "C'est maintenant qu'on calcule: au cœur de la dualité". Habilitation thesis. Université Paris 11.

Hindley, J. Roger (1969). "The principle type-scheme of an object in combinatory logic". In: *Transactions of the American Mathematical Society* 146, pp. 29–60.

Honda, Kohei (1993). "Types for Dyadic Interaction". In: *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*. Springer Berlin Heidelberg, pp. 509–523.

Honda, Kohei, Vasco T. Vasconcelos, and Makoto Kubo (1998). "Language primitives and type discipline for structured communication-based programming". In: *Programming languages and systems: 7th European symposium on programming (ESOP '98)*. Springer Berlin Heidelberg, pp. 122–138.

Howard, William A. (1980). "The formulae-as-types notion of construction". In: *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism* 44, pp. 479–490.

Hughes, John (1989). "Why functional programming matters". In: *The computer journal* 32.2, pp. 98–107.

Jacobs, Bart (1996). "Objects and classes, co-algebraically". In: *Object orientation with parallelism and persistence*. Springer, Boston, MA, pp. 83–103.

Kennedy, Andrew and Claudio V. Russo (2005). "Generalized algebraic data types and object-oriented programming". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. New York, NY, USA: ACM, pp. 21–40.

Kiczales, Gregor and Mira Mezini (2005). "Aspect-oriented programming and modular reasoning". In: *Proceedings of the 27th international conference on software engineering (ICSE '05)*. New York, NY, USA: ACM, pp. 49–58.

Kleene, Stephen Cole, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen (1952). *Introduction to metamathematics*. Vol. 483. Van Nostrand, New York.

Krishnamurthi, Shriram, Matthias Felleisen, and Daniel P Friedman (1998). "Synthesizing object-oriented and functional design to promote re-use". In: *European Conference on Object-Oriented Programming 1998*. Springer Berlin Heidelberg, pp. 91–113.

Lämmel, Ralf and Ondrej Rypacek (2008). "The Expression Lemma". In: *Proceedings of the 9th international conference on Mathematics of Program Construction (MPC 2008)*. Springer Berlin Heidelberg, pp. 193–219.

Levy, Paul Blain (2001). "Call-by-push-value". PhD thesis. Queen Mary, University of London.

Lorenzen, Paul (1967). "Ein Dialogisches Konstruktivitätskriterium". In: *Journal of symbolic logic* 32.4, pp. 516–516. DOI: 10.2307/2270181.

Martin-Löf, Per (1996). "On the meanings of the logical constants and the justifications of the logical laws". In: *Nordic journal of philosophical logic* 1.1, pp. 11–60.

Meijer, Erik, Maarten Fokkinga, and Ross Paterson (1991). "Functional programming with bananas, lenses, envelopes and barbed wire (FPCA 1991)". In: *Conference on Functional Programming Languages and Computer Architecture*. Springer Berlin Heidelberg, pp. 124–144.

Milner, Robin (1978). "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3, pp. 348–375.

Oliveira, Bruno C. d. S. and William R. Cook (2012). "Extensibility for the masses". In: *European Conference on Object-Oriented Programming (ECOOP 2012)*. Springer Berlin Heidelberg, pp. 2–27.

Ostermann, Klaus and Julian Jabs (2018). "Dualizing Generalized Algebraic Data Types by Matrix Transposition". In: *Programming Languages and Systems: 28th European Symposium on Programming (ESOP 2019)*. Springer, Cham, pp. 60–85.

Ostermann, Klaus, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel (2011). "Revisiting information hiding: Reflections on classical and nonclassical modularity". In: *European Conference on Object-Oriented Programming (ECOOP 2011)*. Springer Berlin Heidelberg, pp. 155–178.

Palsberg, Jens and C. Barry Jay (1998). "The essence of the visitor pattern". In: *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98) (Cat. No. 98CB 36241)*. IEEE, pp. 9–15.

Parigot, Michel (1992). "$\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR '92)*. Springer Berlin Heidelberg, pp. 190–201.

Parnas, David L. (1972). "On the criteria to be used in decomposing systems into modules". In: *Pioneers and Their Contributions to Software Engineering*. Springer Berlin Heidelberg, pp. 479–498.

Pfenning, Frank and Rowan Davies (2001). "A judgmental reconstruction of modal logic". In: *Mathematical structures in computer science* 11.4, pp. 511–540.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT press.

Pottier, François and Nadji Gauthier (2004). "Polymorphic typed defunctionalization". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '04)*. New York, NY, USA: ACM, pp. 89–98.

Rendel, Tillmann, Jonathan Immanuel Brachthäuser, and Klaus Ostermann (2014). "From object algebras to attribute grammars". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM New York, NY, USA, pp. 377–395.

Rendel, Tillmann, Julia Trieflinger, and Klaus Ostermann (2015). "Automatic refunctionalization to a language with copattern matching: with applications to the expression problem". In: *Proceedings of the 20th ACM SIGPLAN international conference on functional programming (ICFP '15)*. New York, NY, USA: ACM, pp. 269–279.

Reynolds, John C. (1972). "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM annual conference – Volume 2 (ACM '72)*. New York, NY, USA: ACM, pp. 717–740.

Robinson, John Alan (1965). "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM (JACM)* 12.1, pp. 23–41.

Stillwell, John (2019). *Reverse mathematics: proofs from the inside out*. Princeton University Press.

Swierstra, Wouter (2008). "Data types à la carte". In: *Journal of functional programming* 18.4, p. 423.

Tarr, Peri, Harold Ossher, William Harrison, and Stanley M. Sutton (1999). "N degrees of separation: Multi-dimensional separation of concerns". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE, pp. 107–119.

Torgersen, Mads (2004). "The expression problem revisited". In: *European Conference on Object-Oriented Programming (ECOOP 2004)*. Springer Berlin Heidelberg, pp. 123–146.

Troelstra, Anne S. (1973). *Metamathematical investigation of intuitionistic arithmetic and analysis*. Vol. 344. Springer Science & Business Media.

Wadler, Philip (1990). "Linear types can change the world!" In: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*. North-Holland, pp. 561–581.

— (1998). "The expression problem". In: *Posted on the Java Genericity mailing list*.

— (2003). "Call-by-value is dual to call-by-name". In: *Proceedings of the eighth ACM SIGPLAN international conference on functional programming (ICFP '03)*. New York, NY, USA: ACM, pp. 189–201.

— (2012). "Propositions as sessions". In: *Proceedings of the 17th ACM SIGPLAN international conference on functional programming (ICFP '12)*. New York, NY, USA: ACM, pp. 273–286.

Wang, Yanlin and Bruno C. d. S. Oliveira (2016). "The expression problem, trivially!" In: *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. New York, NY, USA: ACM, pp. 37–41.

Zeilberger, Noam (2008a). "Focusing and higher-order abstract syntax". In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '08)*. New York, NY, USA: ACM, pp. 359–369.

— (2008b). "On the unity of duality". In: *Annals of pure and applied logic* 153.1-3, pp. 66–96.

— (2010). "Polarity and the logic of delimited continuations". In: *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, pp. 219–227.

Zenger, Matthias and Martin Odersky (2001). "Extensible algebraic datatypes with defaults". In: *Proceedings of the sixth ACM SIGPLAN international conference on functional programming (ICFP '01)*. New York, NY, USA: ACM, pp. 241–252.

— (2004). *Independently extensible solutions to the expression problem*. Tech. rep. EPFL.