Programming with Symmetric Data and Codata Types

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
David Binder
aus Göppingen

Tübingen
2024

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

Tag der mündlichen Qualifikation:	7.10.2024
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Klaus Ostermann
2. Berichterstatter:	Prof. Dr. Peter Schroeder-Heister

Contents

Acknowledgements

First, I would like to thank my advisors Klaus Ostermann and Peter Schroeder-Heister without whom I wouldn't be sitting in front of my computer, typing in these acknowledgements. I will always be grateful that I got the chance to spend a considerable part of my life in the best possible way: reading papers and pursuing my intellectual interests. I do not like to work in isolation; most of my ideas that found their way into papers and ultimately this dissertation were formed during lunch break discussions, with a cup of coffee in one hand and a whiteboard marker in the other. For those discussions I thank my friends and colleagues Jonathan Brachthäuser, Philipp Schuster, Ingo Skupin, Julian Jabs, Marius Müller, Jiří Beneš and Marcial Gaißert. And then there are my once collaborators and now friends, Tim Süberkrüb and David Läwen: Thank you for all the hours in front of a whiteboard, and the excitement in those last stressful few hours we spent before a submission deadline in a Zoom call. But without the love of my family, my parents and my brother, I would not be the person that I am. This dissertation is therefore dedicated to you.

Abstract

Two fundamental theories serve as the pillars on which statically typed functional programming languages are constructed: The lambda calculus and the theory of algebraic data types. We recognize these pillars in the core grammar of lambda abstractions and function applications, constructors and pattern matching that all functional programming languages share. Functional programmers are skeptical of extending this core grammar with concepts such as interfaces from object-oriented programming or exceptions from imperative programming languages. This is because these concepts seem to lack a similar relation to logic and mathematics that the lambda calculus and algebraic data types enjoy through the Curry-Howard isomorphism. Advancements in logic and proof theory, however, challenge this skepticism. The analysis of logical connectives in terms of polarity gives a logical interpretation to object-oriented interfaces in the type-theoretic guise of codata types, and the improved understanding of classical logic shows that control operators which are used to implement exception mechanisms correspond to reasoning principles in classical logic.

This dissertation shows that functional programming languages can be made more symmetric and functional by taking these advancements into account. Instead of basing the language on the lambda calculus and algebraic data types we start with algebraic data and codata types which subsume the lambda calculus as a special case. Instead of designing these two complementary language fragments in an ad-hoc manner, we systematically use the program transformations defunctionalization and refunctionalization as a methodology to derive the properties of both fragments. In fully symmetric languages defunctionalization can transform any codata type into a data type, and refunctionalization can transform any codata type into a data type. Since we also want to benefit from the expressive power of control operators, we switch from term assignment systems for natural deduction in the first part of this dissertation to a term assignment system for the sequent calculus in the second part. We demonstrate why the sequent calculus is a better setting than natural deduction for developing symmetric programming languages which require a first-class treatment of control effects and exceptions. During their lifespan, most programming languages turn into a pile of unrelated features; applying duality to their design allows us to turn back the clock a little bit and bring at least some of these accumulated features into a coherent framework governed by symmetry.

Zusammenfassung

Statisch getypte funktionale Programmiersprachen sind auf zwei grundlegenden Theorien aufgebaut: Auf dem Lambdakalkül und auf der Theorie algebraischer Datentypen. Wir erkennen diese beiden Theorien in der Kerngrammatik von Lambdaabstraktionen, Funktionsanwendungen, Konstruktoren und Pattern-Matching die allen funktionalen Programmiersprachen gemeinsam ist. Funktionale Programmierer begegnen dem Versuch diese Kerngrammatik mit anderen Konstrukten wie Schnittstellen aus der objektorientierten oder Ausnahmen aus der imperativen Programmierung zu erweitern häufig mit Skepsis. Dies ist der Fall da diesen Konzepten die enge Verbindung zur Logik und Mathematik zu fehlen scheint welche dem Lambdakalkül und algebraischen Datentypen vermittels des Curry-Howard Isomorphismus zueigen ist. Fortschritte in der Logik und Beweistheorie fordern jedoch dazu heraus diese Skepsis zu überdenken. Die Analyse von logischen Konnektiven in Begriffen der Polarität erlaubt es objektorientierten Schnittstellen eine logische Interpretation als Kodatentypen zu geben, und das verbesserte Verständnis der klassischen Logik zeigt dass Kontrolloperatoren welche zur Implementierung von Ausnahmebehandlung verwendet werden Prinzipien der klassischen Logik entsprechen.

Diese Dissertation zeigt dass funktionale Programmiersprachen symmetrischer und funktionaler gemacht werden können wenn diesen Fortschritten Rechnung getragen wird. Anstatt auf dem Lambdakalkül und algebraischen Datentypen aufzubauen beginnen wir mit der Theorie von algebraischen Daten- und Kodatentypen, welche den Lambdakalkül als Spezialfall einschließen. Anstatt diese zwei komplementären Sprachfragmente unabhängig voneinander zu entwerfen verwenden wir auf systematische Weise die Programmtransformationen Defunktionalisierung und Refunktionalisierung als Methodologie um die Eigenschaften beider Fragmente herzuleiten. In vollständig symmetrischen Sprachen kann Defunktionalisierung jeden Kodatentyp in einen Datentypen transformieren, und Refunktionalisierung jeden Datentyp in einen Kodatentypen. Da wir auch von der Ausdrucksstärke von Kontrolloperatoren Gebrauch machen wollen wechseln wir im zweiten Teil dieser Dissertation zu einem Termzuweisungssystem für den Sequenzenkalkül, nachdem wir im ersten Teil ein Termzuweisungssystem für das System des natürlichen Schließens verwendet haben. Wir zeigen warum der Sequenzenkalkül ein besseres System als das natürliche Schließen ist um symmetrische Programmiersprachen zu entwickeln, welche eine Behandlung von Kontrolleffekten und Ausnahmen als Konstrukte erster Klasse erfordern. Im Laufe der Zeit entwickeln sich die meisten Programmiersprachen zu einer Anhäufung unzusammenhängender Funktionalitäten; indem wir das Prinzip der Dualität anwenden können wir die Uhr ein Stück weit zurückdrehen und zumindest einige dieser angehäuften Features in ein von Symmetrie geprägtes kohärentes System bringen.

1. Introduction

The lambda calculus and algebraic data types are the two pillars on which the theory of statically typed functional programming languages is built. The lambda calculus was introduced by Church (1936), both as a foundational system for mathematics and as a simple model for computable functions. Because it is designed as a minimalist foundational system, the lambda calculus lacks support for important types like natural numbers. Natural numbers are necessary to show that all computable functions on numbers can be represented, but since they are not provided directly they have to be expressed using so-called functional encodings. Examples of such functional encodings are the Church, Scott and Parigot encodings of data types. However, these are not practical when it comes to designing usable functional programming languages, since programmers should not have to understand functional encodings in order to program with inductive types like natural numbers, lists and trees. To model numbers and other inductive types directly, language designers introduced the theory of algebraic data types. Algebraic data types are a principled framework which lets users of functional programming languages extend the set of available types by providing data type declarations. They were first introduced in the functional programming languages Hope and Epigramm in the 1960s and gained popularity through their inclusion in the report on StandardML (cf. MacQueen, Harper, and Reppy (2020)).

The influence of these two foundational theories is still visible in the design of popular functional programming languages like Haskell, OCaml, F#, Idris, Agda or Lean. While these languages differ significantly in how they treat side effects, the order in which they evaluate expressions, or whether they are compiled to a managed runtime or machine code, they nevertheless share a common core by which one can identify them as members of the same family. The family resemblance stems from a shared fundamental grammar of expressions which are used to structure the data and control flow of programs. This shared fundamental grammar can be expressed as the following syntax of expressions:

$$e, t \coloneqq x \mid \lambda x.e \mid e\ e \mid K(e_1, \ldots, e_n) \mid e.\textbf{case}\ \{\ldots, K(x_1, \ldots, x_n) \Rightarrow e, \ldots\}$$

We can see the influence of both the lambda calculus and of algebraic data types in this grammar. The lambda calculus contributes lambda abstractions $\lambda x.e$ and function applications $e\ e$, whereas constructors $K(e_1, \ldots, e_n)$ and case expressions $e.\textbf{case}\ \{\ldots\}$ come from the theory of algebraic data types[1]. These terms for functions and data types can be further split into introduction and elimination forms: Lambda abstractions and

[1]We use the concrete syntax $e.\textbf{case}\ \{\ldots\}$ instead of the more familiar **case** e **of** $\{\ldots\}$ since the former is more uniform with the elimination form $e.d(e_1, \ldots, e_n)$ on codata types which we will introduce later.

constructors are introduction forms, and function applications and case expressions are elimination forms. The language defined by this grammar is of course still quite minimal, but a lot of other expression forms found in functional programming languages are just syntactic sugar for this simple core.

Functional programmers are often skeptical about extending this core grammar with constructs from other programming paradigms such as classes and objects from object-oriented programming or exceptions and exception handlers from imperative programming. What is the reason for this skepticism? In contrast to the the lambda calculus and algebraic datatypes which have a clear and well-known relation to logic and mathematics through the Curry-Howard isomorphism, these other constructs are not perceived to be similarly grounded in logic. If we want to extend functional programming languages with such new features, then we have to demonstrate to functional programmers that these features are useful in practice and that they are also related to logic and mathematics through the Curry-Howard isomorphism.

To show that these features are useful we can rely on examples from the existing programming practice in object-oriented and imperative languages, but in order to show that the Curry-Howard isomorphism applies to them we have to rely on advances in logic, proof theory and type theory. Two advancements in those fields are particularly important: The analysis of logical connectives in terms of polarity and the improved understanding of classical logic. These two developments make it possible to extend the Curry-Howard isomorphism to areas that were previously thought to be out of its reach. The study of polarity showed that there are two fundamentally different kinds of types for which various different names are used in the literature. Sometimes they are called positive and negative, synchronous and asynchronous, or inductive and coinductive types, but we will consistently call them data and codata types. Codata types allow us to extend the Curry-Howard isomorphism to object-oriented programs by giving a logical interpretation to interfaces and the improved understanding of the proof theory of classical logic allows us to give a logical account of control operators such as exceptions and exception handlers since these correspond to classical reasoning principles.

No practical programming language has so far been designed which reflects this deepened understanding of the Curry-Howard isomorphism. What should such a programming language look like? It should reflect our understanding of polarity by providing means for declaring both data and codata types instead of just providing means to declare data types. It should also provide mechanisms for handling exceptions, continuations and complex control flow which are based on our improved understanding of the proof theory of classical logic. This thesis shows how the design of functional programming languages can take these advancements in our understanding of the Curry-Howard isomorphism into account. In the rest of the introduction I will give a high-level introduction to these two big ideas: I will introduce codata types in section 1.1 and the proof theory of classical logic in the sequent calculus in section 1.2. I will then describe the contributions and structure of this dissertation in section 1.3.

1.1. Data and Codata Types

There are many ways to motivate the introduction of codata types to programming languages. Common to all of them is that the polarity of a type (i.e. whether it is a data or a codata type) influences other properties which are directly relevant for programmers. In the next subsections, I give three different examples of this phenomenon. I show in section 1.1.1 how pairs can be defined as both data and codata types, and how the difference can be observed in programming languages that use linear types to reason about resources. In section 1.1.2 I show how data and codata types precisely capture the dilemma at the core of the expression problem. Data types can easily be extended with new consumers and codata types can easily be extended with new producers, but the inverse is not the case. Finally, in section 1.1.3 I show that data and codata types provide a principled answer to the question whether programming languages should be strict or lazy: Instead of making one global decision, we should choose an evaluation order depending on the type. Data types should be evaluated eagerly and codata types should be evaluated on demand.

1.1.1. Two Ways to Define Pairs

In order to illustrate the difference between data and codata types we start with a type that can be presented in both ways. The simplest example of such a type is the pair type $\tau_1 \times \tau_2$ for which one can find the following typing and conversion rules in many articles and books:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash t : \tau_2}{\Gamma \vdash [e, t] : \tau_1 \times \tau_2}\ \times\text{-I} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.\pi_1 : \tau_1}\ \times\text{-E}_1 \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.\pi_2 : \tau_2}\ \times\text{-E}_2$$

$$[e, t].\pi_1 \equiv_\beta e \qquad [e, t].\pi_2 \equiv_\beta t \qquad e \equiv_\eta [e.\pi_1, e.\pi_2]$$

In this version of the rules the pair type $\tau_1 \times \tau_2$ is defined by the pairing constructor $[\_, \_]$ which is used in the introduction rule and the projections π_1 and π_2 which are used in the elimination rules[2]. However, it is not clear if the introduction or the elimination rules are more fundamental; the type is defined simultaneously by all of its rules. We can take this *unpolarized* pair type and *polarize* it by choosing either the introduction or elimination rules as the rules that define its meaning. These two different choices result in two different types: If we pick the constructor $[\_, \_]$ as fundamental we obtain the data type $\tau_1 \otimes \tau_2$ (pronounced "tensor") and if we take the projections π_1 and π_2 as fundamental we obtain the codata type $\tau_1 \& \tau_2$ (pronounced "with"). Let us now look at those two alternatives in more detail.

The rules for the data type $\tau_1 \otimes \tau_2$ are more familiar to functional programmers. An OCaml programmer, for example, might write the following type declaration and swap function for pairs:

[2]Instead of the more familiar prefix syntax $\pi_1\ e$ I use the postfix notation $e.\pi_1$ since projections are a special instance of destructors on codata types.

```
type ('a, 'b) pair = MkPair of 'a * 'b;;
let swap_pair p = match p with MkPair (x,y) -> MkPair (y,x);;
```

A Haskell programmer expresses the same program with only slightly different syntax:

```
data Pair a b = MkPair a b
swap_pair p = case p of { MkPair x y -> MkPair y x }
```

In both of these examples, we can see that the introduction form is more fundamental since it is introduced as part of the data type declaration. A proof theorist might express the same idea by saying that this type is characterized by the fact that we have a notion of canonical proof: a canonical proof of $\tau_1 \otimes \tau_2$ consists of a proof of τ_1 and a proof of τ_2. The elimination form, i.e. pattern matching, on the other hand, is derived or justified based on all available canonical proofs. In this case, there is only one form of canonical proof which corresponds to the fact that we only have to pattern match on one constructor.

Instead of relying on the concrete syntax of OCaml or Haskell we can also present the type $\tau_1 \otimes \tau_2$ using formal typing rules. For the introduction rule, we use the constructor $[e, t]$ while we use a pattern match $e.\mathbf{case}\ \{[x, y] \Rightarrow c\}$ for the elimination rule.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Delta \vdash t : \tau_2}{\Gamma, \Delta \vdash [e, t] : \tau_1 \otimes \tau_2}\ \otimes\text{-I} \qquad \frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash c : \tau_3}{\Gamma \vdash e.\mathbf{case}\ \{[x, y] \Rightarrow e'\} : \tau_3}\ \otimes\text{-E}$$

$$[e, t].\mathbf{case}\ \{[x, y] \Rightarrow e'\} \equiv_\beta e'[e/x, t/y] \qquad e \equiv_\eta e.\mathbf{case}\ \{[x, y] \Rightarrow [x, y]\}$$

What is less familiar to functional programmers is that we can also define pairs using their elimination rules. This means that a pair is defined by the two observations or methods `first` and `second` that we can invoke on it. One way to define types in this way is through interfaces or similar mechanisms in object-oriented languages. In Java, for example, we can define the following generic interface for pairs:

```
public interface Pair<A,B> {
    public A first();
    public B second();
}
```

Every object that implements this interface behaves as a pair since we can invoke both projections as methods on that object. Object-oriented programming languages like Java unfortunately add a lot of additional complexity to this simple idea by also allowing mutation and inheritance for which we cannot easily give a Curry-Howard interpretation. But we can nevertheless observe that types in object-oriented languages are characterized by the methods that we can invoke on them. This means that types in object-oriented languages are defined by their elimination rules. If we focus on this central idea and formalize it we obtain the type-theoretic notion of codata types. The proof assistant Agda, for example, supports codata types and allows to write the pair type in the following way:

```
record Pair (A B : Set) : Set where
  coinductive
  field
    first : A
    second : B
```

This codata type declaration directly tells us that we can invoke the `first` and `second` observation on any pair. However, we also need a way to introduce terms of these codata types. The introduction rule allows to introduce a term by specifying how it behaves on each of the possible observations. This term-forming operation is called *copattern matching* (Abel, Pientka, Thibodeau, and Setzer (2013)) since it is the dual of pattern-matching: pattern matching specifies how to behave on each of the possible canonical proofs of a data type and copattern matching specifies how to behave on each of the possible canonical refutations. We use **cocase** $\{\ldots\}$ to write terms introduced using copattern matching. An example of this can be seen in the following formal typing rules for the codata type $\tau_1 \mathbin{\&} \tau_2$.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash t : \tau_2}{\Gamma \vdash \textbf{cocase}\ \{\pi_1 \Rightarrow e, \pi_2 \Rightarrow t\} : \tau_1 \mathbin{\&} \tau_2}\ \&\text{-I}$$

$$\frac{\Gamma \vdash e : \tau_1 \mathbin{\&} \tau_2}{\Gamma \vdash e.\pi_1 : \tau_1}\ \&\text{-E}_1 \qquad\qquad \frac{\Gamma \vdash e : \tau_1 \mathbin{\&} \tau_2}{\Gamma \vdash e.\pi_2 : \tau_2}\ \&\text{-E}_2$$

$$\textbf{cocase}\ \{\pi_1 \Rightarrow e_1, \pi_2 \Rightarrow e_2\}.\pi_1 \equiv_\beta e_1 \qquad \textbf{cocase}\ \{\pi_1 \Rightarrow e_1, \pi_2 \Rightarrow e_2\}.\pi_2 \equiv_\beta e_2$$

$$e \equiv_\eta \textbf{cocase}\ \{\pi_1 \Rightarrow e.\pi_1, \pi_2 \Rightarrow e.\pi_2\}$$

We have now seen two alternative ways to define pairs $\tau_1 \times \tau_2$ as either a data type $\tau_1 \otimes \tau_2$ or a codata type $\tau_1 \mathbin{\&} \tau_2$. A natural question is whether a language should provide both mechanisms to define pairs or if this redundancy is useless in practice. The difference turns out to be important in substructural programming languages which leverage the type system to track the use of resources in the typing rules since there is an important difference between the types $\tau_1 \otimes \tau_2$ and $\tau_1 \mathbin{\&} \tau_2$[3]. In a substructural type theory, we treat variables and terms as resources that cannot be arbitrarily duplicated or discarded. This is useful, for example, if we want to write programs that manipulate file handles since an open file handle should not be freely duplicated and shared between different functions, but it also cannot be freely discarded since we have to close the file handle once we have finished reading from it. And within such a theory that cares about resource usage, we can distinguish the two pair types. When we eliminate the pair type $\tau_1 \otimes \tau_2$ by pattern matching we have access to *both* elements τ_1 and τ_2. For the pair type $\tau_1 \mathbin{\&} \tau_2$, on the other hand, we can only invoke one projection to obtain either an element of type τ_1 or an element of τ_2. This shows that we do want to have both pair types available in the programming language if our underlying system is linear.

[3]In section 1.1.3 we will see another difference; the type $\tau_1 \otimes \tau_2$ corresponds to *strict* pairs whose elements are evaluated eagerly whereas the type $\tau_1 \mathbin{\&} \tau_2$ corresponds to *lazy* pairs whose elements are evaluated upon the invocation of the first or second projection.

I already mentioned several times that constructors encode canonical proofs and that pattern matching corresponds to elimination rules which are justified based on canonical proofs. I will now provide some context in order to make this relationship between programming constructs such as constructors and pattern matching on the one hand, and proof-theoretic concepts such as canonical and justified proofs on the other hand, clearer. The distinction between canonical proofs and justified proofs can be traced back to a remark by Gentzen in his foundational paper which introduced the natural deduction calculus. In that calculus every logical connective is characterized by both introduction and elimination rules. He observes that these introduction and elimination rules observe some sort of internal coherence and writes:

> The introductions represent, as it were, the "definitions" of the symbols concerned, and the eliminations are no more, in the final analysis, than the consequences of these definitions. This fact may be expressed as follows: In eliminating a symbol, we may use the formula with whose terminal symbol we are dealing only "in the sense afforded it by the introduction of that symbol". (Gentzen (1935a) and Gentzen (1935b), cited via Schroeder-Heister (2018))

This small remark was the beginning of an entire strand of research that tries to explain the meaning of logical constants in terms of the valid inference rules that characterize them. Examples of this development can be found by Prawitz (1985), Martin-Löf (1996) and Dummett (1991), and a good overview of this field is presented by Schroeder-Heister (2018). If we try to apply the remark by Gentzen to the theory of data and codata types that we started to introduce above, then we can see that data types are types whose meaning is defined by the canonical introduction forms which are mentioned in the data type declaration, and that pattern matching is a form of justified elimination rule. This relationship, and how it extends to codata types and copattern matching, was made completely explicit by Zeilberger (2009).

How the meaning of logical constants can be explained in terms of the role they play in inferences was also investigated by another influential philosopher: Karl Popper. He developed an inferentialist theory of meaning where the meaning of a logical constant is not defined in terms of denotational models but through the role it plays in logical inferences. This is described in several of my articles on the historical development of logic (Binder, Piecha, and Schroeder-Heister (2022), Binder and Piecha (2017), and Binder and Piecha (2021)) which influenced the general outlook developed in this thesis but did not become a part of its text.

1.1.2. Data Types, Codata Types and the Expression Problem

One way to present the essence of the *expression problem* (Wadler (1998)) is to show how the extensibility properties of a type depend on its polarity, i.e. whether it is a data or codata type. It is easy to extend a data type with new consumers but difficult to extend it with new producers. Dually, it is easy to add new producers to a codata type, but difficult to extend it with new consumers. This problem can be illustrated with the canonical example of arithmetic expressions. Arithmetic expressions consist of either natural number literals or the addition of two arithmetic expressions. The only observation we consider at first is evaluation, which allows us to compute the value of an arithmetic expression. We can consider two possible extensions of this program: If we extend arithmetic expressions with multiplication nodes then we have to add a new producer, and if we also want to print arithmetic expressions then we have to extend the program with a new consumer. Whether each of these extensions is easy or hard depends on the representation that we have chosen. Let us look at the two different representations in turn.

Arithmetic Expressions as a Data Type

In functional programming languages we usually model syntax trees as a data type and implement the `eval` function by pattern matching on all available syntax nodes. For the simple example of arithmetic expressions this looks as follows:

```
data AST {
    Lit(Nat),
    Add(AST,AST)
}
def eval(x) = x.case {
    Lit(n) => n,
    Add(e1,e2) => eval(e1) + eval(e2)
}
```

It is easy to extend this program with a `print` function, where we use `++` for string concatenation and `print_nat` as a primitive function for printing natural numbers.

```
def print(x) = x.case {
    Lit(n) => print_nat(n),
    Add(e1,e2) => "(" ++ print(e1) ++ "+" ++ print(e2) ++ ")"
}
```

If we want to extend the language with multiplication nodes, however, then we have to add a clause to all the existing parts of the program which pattern match on ASTs.

```
data AST {
    Lit(Nat),
    Add(AST,AST),
    Mult(AST,AST),
}
def eval(x) = x.case {
    Lit(n) => n,
    Add(e1,e2) => eval(e1) + eval(e2)
    Mult(e1,e2) => eval(e1) * eval(e2)
}
```

In this example, there is only one such definition that we have to modify, but in a larger project, there might be many more definitions that we have to revisit. There is a similar situation in mathematics where it is very easy to add another lemma or proof to the existing corpus of mathematical knowledge, but if we change a definition of a mathematical concept then we have to revisit all existing proofs and check that they are still correct. This vague analogy between data types and definitions of mathematical objects can be made precise for proof assistants which usually use data types to model mathematical objects. I have studied the extension of the expression problem to proof assistants in more detail in a recent publication (Binder, Skupin, Süberkrüb, and Ostermann (2024b)).

Arithmetic Expressions as a Codata Type

Using codata types we can model a syntax tree by specifying all the methods that every syntax node has to implement. If we only consider evaluation, then we can write the initial program as follows.

```
codata Ast {
    eval : Nat
}
def Lit(n) = cocase {
    eval => n
}
def Add(e1,e2) = cocase {
    eval => eval(e1) + eval(e2)
}
```

It is easy to extend this program with a new syntax node for multiplication, because we only have to add a new definition that implements the `eval` observation.

```
def Mult(e1,e2) = cocase {
    eval => eval(e1) * eval(e2)
}
```

but it is hard to extend the codata type with a new `print` observation since we have to modify all existing copattern matches:

```
codata Ast {
    eval : Nat,
    print: String
}
def Lit(n) = cocase {
    eval => n,
    print => print_nat(n)
}
def Add(e1,e2) = cocase {
    eval => eval(e1) + eval(e2),
    print => "(" ++ print(e1) ++ "+" ++ print(e2) ++ ")"
}
```

The expression problem is considered a genuine problem since we can usually not foresee whether we want to extend a program by new producers or consumers in the future, and we usually want to extend programs by both. Many solutions of varying complexity have been suggested over the years. Providing both data and codata types in a language doesn't solve the expression problem by itself, but it provides at least a choice for each individual type. A much more thorough discussion of the expression problem, and a partial solution involving the defunctionalization and refunctionalization algorithms, is presented in chapter 3 and chapter 4 of this thesis.

1.1.3. Streams and Non-Strict Evaluation

The distinction between data and codata types is also essential for better understanding the difference between producer-driven and demand-driven computation. Most programs are producer-driven, but Hughes (1989) showed that in many domains we can use laziness, which is one way to implement demand-driven computation, to write programs more modularly as the composition of smaller functions. One such domain concerns the computation on infinite streams, which we want to decompose into smaller functions that take streams as arguments and return streams as results. It is obviously not possible to represent all elements of an infinite stream in finite memory. We therefore have to find a representation that allows us to only compute those elements of a stream which are demanded by the surrounding program. One possible representation is to use a data type with one "`Cons`" constructor in a lazy programming language like Haskell. We can use this data type declaration to define a stream which consists of infinitely many ones:

```
data Stream { Cons(Nat, Stream) }

def ones = Cons(1, ones)
```

This definition crucially only works in a lazy programming language, since it would create an infinite loop in a strict programming language. While this definition works in a lazy programming language, it is still not an optimal representation, as we will discuss below. Instead, it is much better to model streams as a codata type which makes the demand-driven nature explicit. More concretely, we can use the following codata type and definition:

```
codata Stream { head: Nat, tail: Stream }

def ones = cocase { head => 1, tail => ones }
```

One of the main reasons why it is preferable to formulate such demand-driven types as codata types concerns the validity of η-laws. Let us take a look at the η-laws for a stream e in its data and codata variant.

$$t[e/z] \equiv_\eta e.\textbf{case}\ \{\texttt{Cons}(x,y) \Rightarrow t[\texttt{Cons}(x,y)/z]\} \qquad (\eta\text{-Data})$$
$$e \equiv_\eta \textbf{cocase}\ \{\texttt{head} \Rightarrow e.\texttt{head}, \texttt{tail} \Rightarrow e.\texttt{tail}\} \qquad (\eta\text{-Codata})$$

The rule η-Data says that if e occurs in t then we can first pattern-match on e and substitute the canonical form $Cons(x,y)$ for the original occurrence of e in t. The rule η-Codata, on the other hand, says that e can be replaced by a copattern match which "forwards" all the observations to the term e; this form of the η-law is a straightforward generalization of the familiar η-law for functions. These η-laws, however, are in general not valid for every evaluation strategy. The rule η-Data is only valid for strict evaluation strategies, and the rule η-Codata is only valid for non-strict evaluation strategies; let us demonstrate why this is the case.

Let us first show that the rule η-Data is only valid for strict evaluation orders. From now on we assume that Ω is some arbitrary non-terminating term. The following two terms are η-equivalent, but they behave differently when we use a non-strict evaluation order. In that case, the term on the left diverges, whereas the term on the right evaluates to the value 5. Using a strict evaluation order, on the other hand, both terms diverge.

$$\Omega.\textbf{case}\ \{\texttt{Cons}(x,xs) \Rightarrow (\lambda x.5)\texttt{Cons}(x,xs)\} \equiv_\eta (\lambda x.5)\Omega$$

Vice versa, the rule η-Codata is only valid for non-strict evaluation orders, as the following example shows. Here the terms on either side of the equation evaluate to the value 5 under a non-strict evaluation order, but only the term on the right evaluates to 5 under a strict evaluation order.

$$(\lambda x.5)\Omega \equiv_\eta (\lambda x.5)(\textbf{cocase}\ \{\texttt{head} \Rightarrow \Omega.\texttt{head}, \texttt{tail} \Rightarrow \Omega.\texttt{tail}\})$$

The moral of this observation is that if we care about the validity of η-laws, then we should use strict evaluation for data types and non-strict evaluation for codata types. Programmers should care about the validity of η-laws since many common refactorings are only valid if a corresponding η-law is valid. And since it follows directly that we cannot use the data representation of streams any longer, this means that we do need support for the codata representation of streams in a language. Chapter 4 is mainly concerned with this interaction between the polarity (data or codata) and the evaluation order (call-by-value or call-by-name) chosen for a type. In that chapter I present a system with only one form of type declaration which is parameterized by both the polarity and the evaluation order. Separate algorithms allow to change the polarity and the evaluation order of a type independently from each other, without changing the meaning of the program.

1.2. Formats of Reasoning

Many different calculi are used for proving propositions in various logical systems such as propositional logic, predicate logic or modal logic. Among these different calculi, we can identify families whose members resemble each other in the way that the rules and axioms are organized. Examples of such families are axiomatic calculi, natural deduction systems and sequent calculi. We call these families of calculi different *formats of reasoning*. The central point of this section is to show that these different formats of reasoning also correspond to different programming styles and that the programming style that corresponds to the classical sequent calculus is still mostly unexplored. I focus on the following three correspondences to illustrate the point[4]:

Reasoning Format	Programming Language
Axiomatic Calculus	Combinatory Logic
Natural Deduction	Lambda Calculus
Classical Sequent Calculus	$\lambda\mu\tilde{\mu}$-Calculus

How are properties of the logical calculus reflected in the corresponding programming language? The central property of an axiomatic calculus, for example, is that it does not allow to make temporary assumptions. Combinatory logic, the programming language

[4]The excellent book by Troelstra and Schwichtenberg (2000) contains a much more thorough introduction to the differences between axiomatic calculi, natural deduction and the sequent calculus.

which corresponds to axiomatic calculi, correspondingly does not make use of variables. The most important innovation in natural deduction is the use of temporary assumptions. These temporary assumptions make reasoning in those systems more *natural*, and they correspond to the use of variables in typed lambda calculi. What, then, is the property of the sequent calculus which distinguishes it from natural deduction, and how does this manifest itself when we look at its corresponding programming language? The core difference is that in the classical sequent calculus, we can also make a temporary assumption that a proposition is *false*, whereas in natural deduction we can only assume that a proposition is true. Translated into the vocabulary of programming this means that a language built on the sequent calculus has first-class support for continuations. The following three subsections will explain these correspondences in more detail.

1.2.1. Axiomatic Calculi and Combinatory Logic

The early modern formalisms for writing proofs in a fully rigorous manner are almost all axiomatic calculi. For example, when Frege (1879) wrote down the rules for the propositional fragment of his Begriffsschrift he used a collection of logical axioms but only one rule of inference: *modus ponens*. Later authors often followed this model. Whitehead and Russell, for example, built upon Frege's work and specified the rules for *Principia Mathematica* as an axiomatic system. Many other textbooks on logic that appeared in the first half of the twentieth century, such as the very influential "Grundzüge der theoretischen Logik" by Hilbert and Ackermann, also followed this approach.

We can illustrate the general form of an axiomatic calculus by giving an example for the implicational fragment of propositional logic. The formulas of this fragment consist only of propositional variables and implications. First, we need a collection of axioms:

Definition 1.2.1 (Axioms of Implicational Logic)**.** The following formulas are axioms of the implicational fragment of propositional logic:

1. $\phi \rightarrow \phi$
2. $\phi \rightarrow \psi \rightarrow \phi$
3. $(\phi \rightarrow \psi \rightarrow \chi) \rightarrow (\phi \rightarrow \psi) \rightarrow \phi \rightarrow \chi$

We then have to define what we mean by a valid proof. The following definition uses the axioms and one inference rule: modus ponens.

Definition 1.2.2 (Proofs of Implicational Logic)**.** A proof of a formula ϕ is a finite list of formulas ending with ϕ, where every formula in the list satisfies one of the following two conditions:

1. The formula is an instance of one of the axioms.
2. The formula is of the form ψ, and there is some ϕ such that the formulas $\phi \rightarrow \psi$ and ϕ occur earlier in the list.

This definition already suffices to give a fully formal proof of the tautology $\phi \to \psi \to \psi$.

Example 1.2.3. The list $[(\psi \to \psi) \to \phi \to \psi \to \psi, \psi \to \psi, \phi \to (\psi \to \psi)]$ is a proof of the proposition $\phi \to \psi \to \psi$ according to definition 1.2.2. We can annotate each element of the list with its corresponding justification:

$$\begin{aligned} (\psi \to \psi) \to \phi \to \psi \to \psi &\qquad \text{(1: by Axiom 2)} \\ \psi \to \psi &\qquad \text{(2: by Axiom 1)} \\ \phi \to (\psi \to \psi) &\qquad \text{(3: By 1 and 2)} \end{aligned}$$

This simple proof calculus is sound and complete, but the proofs that we have to write are quite unnatural. The reason why these proofs feel unnatural is that the calculus does not allow us to make temporary assumptions; we have to proceed from one tautological formula to the next. If we turn this simple proof system into a programming language by assigning terms to derivations, then we obtain combinatory logic:

Definition 1.2.4 (Combinatory Logic)**.** The terms of combinatory logic are constructed according to the following grammar:

$$e \;::=\; S \;\mid\; K \;\mid\; I \;\mid\; e\; e$$

Here is how combinatory logic arises as the term assignment system for the calculus defined by the axioms and inference rules introduced above. For each of the three axioms of definition 1.2.1 we have a term constructor S, K and I, and we use application $e\ e$ for the second clause of definition 1.2.2. The proof of example 1.2.3, for example, can be represented by the combinatory logic term KI.

Writing functional programs without the use of variables and lambda abstractions is known as “point-free programming”, and standard techniques from the theory of combinatory logic[5] can be used to translate any functional program into this point-free style. Whether programs written in this style are readable, however, is a separate question. Compiling to combinatory logic has also been used as an implementation technique by some compilers for functional languages. An early example of this is Turner, 1979's compiler for the programming language SASL.

1.2.2. Natural Deduction and the Lambda Calculus

In the previous section we observed that if we only proceed from one tautological formula to the next without making temporary assumptions, then our logical calculus does not

[5]Hindley and Seldin (2008) and Barendregt (1981), for example, provide various translations from the lambda calculus to combinatory logic.

properly reflect how people usually construct proofs. This was also observed by logicians who then developed new calculi which capture the way mathematicians actually construct proofs; these novel calculi were called "natural deduction calculi". They were independently discovered by Gentzen (1935a) and Gentzen (1935b) and by Jaśkowski (1934)[6].

The calculus of natural deduction has two characteristic properties. The first property which distinguishes it from the earlier axiomatic calculi is the use of temporary assumptions which are discharged during the course of the proof. The second characteristic property is that each rule only mentions one principal connective and that each rule can be characterized as an introduction or an elimination rule, depending on whether the principal connective appears in a premiss or the conclusion.

Definition 1.2.5 (Inference Rules of Natural Deduction)**.** There are two rules for the implicational fragment of natural deduction: One introduction rule and one elimination rule.

$$\dfrac{\begin{matrix}[\phi]^1\\ \mathcal{D}\\ \psi\end{matrix}}{\phi \to \psi}\ \to\text{-Intro} : 1 \qquad\qquad \dfrac{\begin{matrix}\mathcal{D}_1\\ \phi \to \psi\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ \phi\end{matrix}}{\psi}\ \to\text{-Elim}$$

Using these inference rules we can now construct proofs in a much more natural way, which can be seen if we write down the proof of the example contained in the previous section.

Example 1.2.6. The proof of the theorem from example 1.2.3 in natural deduction is:

$$\dfrac{\dfrac{[\psi]^1}{\psi \to \psi}\ \to\text{-Intro} : 1}{\phi \to \psi \to \psi}\ \to\text{-Intro} : 2$$

In this proof, we start with the temporary assumption ψ which we discharge in the first inference step to obtain a closed proof of the proposition $\psi \to \psi$. Note that we do not discharge any assumptions in the second inference step; this is nevertheless a valid instance of the introduction rule.

The calculus of natural deduction is especially important for programming language designers due to its term assignment system: the lambda calculus. The lambda calculus was first invented in its untyped and typed form by Alonzo Church (Church, 1936; Church, 1940). The relationship between natural deduction and the (typed) lambda calculus, which is nowadays known as the Curry-Howard correspondence or "propositions-as-types"(cp. Sørensen and Urzyczyn, 2006), was not known from the beginning. It was first observed by Howard (1980) as a correspondence between typed combinatory logic and the axioms of propositional logic, and only later extended to the lambda calculus.

[6]A superficial difference between these two calculi is that Gentzen used trees to write proofs whereas Jaśkowski uses a tabular format.

Definition 1.2.7 (Lambda Calculus)**.** The terms of the lambda calculus are constructed according to the following grammar:

$$e_\lambda ::= x \mid \lambda x.e_\lambda \mid e_\lambda\ e_\lambda.$$

For example, we can annotate the proof from the example above and see that this proof can be encoded by the lambda term $\lambda x.\lambda y.y$.

$$\dfrac{\dfrac{y : \psi \vdash y : \psi}{\vdash \lambda y.y : \psi \to \psi}\ \to\text{-Intro}}{\vdash \lambda x.\lambda y.y : \phi \to \psi \to \psi}\ \to\text{-Intro}$$

Here we use a context Γ of temporary assumptions on the left of the turnstile to record those assumptions that we have not discharged yet; this is sometimes called natural deduction in sequent-calculus style.

1.2.3. Sequent Calculus and the $\lambda\mu\tilde{\mu}$-Calculus

We have seen that by making temporary assumptions we can write more natural proofs in natural deduction than in axiomatic calculi, and that these temporary assumptions correspond to the use of variables in the lambda calculus. In the sequent calculus we can go one step further because we can not only assume that a proposition is *true*, but also that a proposition is *false*. This new kind of assumption that a proposition is false corresponds to *covariables* which are used in the $\lambda\mu\tilde{\mu}$-calculus, the term assignment system for the sequent calculus that I will now introduce.

The classical sequent calculus operates on sequents $\Gamma \vdash \Delta$, where both Γ and Δ are ordered lists of formulas. The meaning of a sequent $\phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_n$ is ordinarily explained by saying that if all the propositions ϕ_1 to ϕ_n are true, then at least one of the propositions ψ_1 to ψ_n has to be true as well. Equivalently, we can say that if all the propositions ϕ_1 to ϕ_n are true and all the propositions ψ_1 to ψ_n are false then we have a contradiction. Put differently, we assume the *truth* of all the ϕ_i and the *falsity* of all the ψ_i. While these two explanations are logically equivalent, the latter of the two is more appropriate to understand the term assignment system of the sequent calculus, which uses variables to encode assumptions of truth and covariables to encode assumptions of falsity.

Every inference rule of the sequent calculus has a finite set of sequents as premisses, and exactly one sequent as its conclusion. There are two kinds of rules which are used to construct proofs: structural rules and logical rules. Structural rules do not mention any logical connectives and only govern the behavior of sequents which is independent of the logical form of the propositions that occur in it[7]The logical rules, on the other hand, determine the meaning of the logical constants.

[7]The natural deduction calculus does not use explicit structural rules; weakening and contraction are instead implicitly handled through the convention that assumptions may be used arbitrarily often in a derivation.

Definition 1.2.8 (Structural Rules of the Sequent Calculus)**.** The classical sequent calculus is characterized by the following structural rules:

$$\frac{}{\phi \vdash \phi}\ \text{Axiom} \qquad \frac{\Gamma_1 \vdash \Delta_1, \phi \qquad \phi, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \text{Cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta}\ \text{Weakening-L} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta}\ \text{Weakening-R}$$

$$\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta}\ \text{Contraction-L} \qquad \frac{\Gamma \vdash \phi, \phi, \Delta}{\Gamma \vdash \phi, \Delta}\ \text{Contraction-R}$$

$$\frac{\Gamma_1, \phi, \psi, \Gamma_2 \vdash \Delta}{\Gamma_1, \psi, \phi, \Gamma_2 \vdash \Delta}\ \text{Exchange-L} \qquad \frac{\Gamma \vdash \Delta_1, \phi, \psi, \Delta_2}{\Gamma \vdash \Delta_1, \psi, \phi, \Delta_2}\ \text{Exchange-R}$$

In natural deduction, the rules for a logical connective came as introduction and elimination rules. In the sequent calculus, by contrast, there are only introduction rules, but every logical constant is governed by left-introduction and right-introduction rules. We illustrate this with the example of the function type.

Definition 1.2.9 (Logical Rules of the Sequent Calculus)**.** The implicational fragment of the classical sequent calculus is characterized by the following right and left introduction rule:

$$\frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta}\ \to\text{-RI} \qquad \frac{\Gamma_1 \vdash \phi, \Delta_1 \qquad \Gamma_2, \psi \vdash \Delta_2}{\Gamma_1, \Gamma_2, \phi \to \psi \vdash \Delta_1, \Delta_2}\ \to\text{-LI}$$

The calculus that we obtain from definitions 1.2.8 and 1.2.9 is strictly more powerful than the calculi introduced in sections 1.2.1 and 1.2.2, because we can prove propositions that are not intuitionistically valid. An example of this is the following proof:

Example 1.2.10 (Peirce's Law)**.** Consider Peirce's law $((\phi \to \psi) \to \phi) \to \phi$, for which we can construct the following proof:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vdash \phi}\ \text{Axiom}}{\phi \vdash \psi, \phi}\ \text{Weakening-R}}{\vdash \phi \to \psi, \phi}\ \to\text{-RI} \qquad \dfrac{}{\phi \vdash \phi}\ \text{Axiom}}{(\phi \to \psi) \to \phi \vdash \phi, \phi}\ \to\text{-LI}}{(\phi \to \psi) \to \phi \vdash \phi}\ \text{Contraction-R}}{\vdash ((\phi \to \psi) \to \phi) \to \phi}\ \to\text{-RI}$$

That Peirce's law is valid in classical logic can also easily be verified by constructing the corresponding truth table.

That we can prove arbitrary classically valid propositions might be concerning from a programming language perspective. The point of constructive logic, after all, is to ensure that all proofs have a computational meaning, and it is not obvious that a computational interpretation can be given to classical proofs. But, as Griffin (1989) discovered, we can give a computational interpretation to classical proofs using *control operators*. Examples of such control operators are Landin (1965)'s operator **J**, Scheme's **call/cc** or the operator $\mathcal{C}$ introduced by Felleisen, Friedman, Kohlbecker, and B. Duba (1987). While the details are different, all these control operators have in common that they allow the program to access the *continuation* from within an expression.

Griffin's insight made it possible to then develop a well-behaved term assignment system for the sequent calculus. One of the first steps was the $\lambda\mu$-calculus by Parigot (1992a), which introduced the control operator μ. Curien and Herbelin (2000) completed this control operator μ with the dual $\tilde{\mu}$ construct to obtain the $\lambda\mu\tilde{\mu}$ calculus:

Definition 1.2.11 (The $\lambda\mu\tilde{\mu}$ Calculus (Syntax))**.** The terms of the $\lambda\mu\tilde{\mu}$-calculus are constructed according to the following grammar:

$$\begin{array}{rcll} p & ::= & x \mid \mu\alpha.s \mid \lambda(x\cdot\alpha).s & \textit{Producers} \\ c & ::= & \alpha \mid \tilde{\mu}x.s \mid p\cdot c & \textit{Consumers} \\ s & ::= & \langle p \mid c \rangle & \textit{Statements} \end{array}$$

In contrast to most term-assignment systems for natural deduction, like the lambda calculus, which only have one syntactic category, the $\lambda\mu\tilde{\mu}$-calculus has three syntactic categories: producers, consumers and statements. From the perspective of the Curry-Howard isomorphism producers correspond to proofs that a proposition is true, consumers correspond to refutations which show that a proposition is false, and statements are used to prove a contradiction. This split is also reflected in the typing rules, which use three different judgments. The judgment $\Gamma \vdash p : \sigma$ says that the producer has type σ, the judgment $\Gamma \vdash c \overset{\mathsf{con}}{:} \sigma$ says that the consumer c has type σ, and the judgment $\Gamma \vdash s$ says that the statement s is well-typed. Type environments Γ contain both variable assignments $x : \sigma$ which correspond to an assumption that σ is true, and covariable assignments $\alpha \overset{\mathsf{con}}{:} \sigma$ which correspond to the assumption that σ is false.

Definition 1.2.12 (The $\lambda\mu\tilde{\mu}$ Calculus (Typing Rules))**.** The typing rules of the $\lambda\mu\tilde{\mu}$-calculus are

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\textsc{Var} \qquad \frac{\alpha \overset{\mathsf{con}}{:} \tau \in \Gamma}{\Gamma \vdash \alpha \overset{\mathsf{con}}{:} \tau}\textsc{Covar} \qquad \frac{\Gamma, \alpha \overset{\mathsf{con}}{:} \tau \vdash s}{\Gamma \vdash \mu\alpha.s : \tau}\mu \qquad \frac{\Gamma, x : \tau \vdash s}{\Gamma \vdash \tilde{\mu}x.s \overset{\mathsf{con}}{:} \tau}\tilde{\mu}$$

$$\frac{\Gamma_1 \vdash p : \tau \qquad \Gamma_2 \vdash c : \tau}{\Gamma_1, \Gamma_2 \vdash \langle p \mid c \rangle}\textsc{Cut}$$

$$\frac{\Gamma, x : \tau_1, \alpha \overset{\mathsf{con}}{:} \tau_2 \vdash s}{\Gamma \vdash \lambda(x\cdot\alpha).s : \tau_1 \to \tau_2}\to\text{-R} \qquad \frac{\Gamma_1 \vdash p : \tau_1 \qquad \Gamma_2 \vdash c : \tau_2}{\Gamma_1, \Gamma_2 \vdash p\cdot c \overset{\mathsf{con}}{:} \tau_1 \to \tau_2}\to\text{-L}$$

Using these typing rules we can now assign a term to the proof of Peirce's law that we have given above.

Example 1.2.13 (Peirce's Law, Continued)**.** The term that we can assign to the proof of example 1.2.10 is $\lambda(x \cdot \alpha).\langle x \mid (\lambda(y \cdot \beta).\langle y \mid \alpha \rangle) \cdot \alpha \rangle$. It can be observed that the covariable α that is bound in the outer lambda abstraction is used twice, and that the covariable β that is bound in the inner lambda abstraction is discarded. This non-linear use of covariables is the characteristic property of terms that we assign to proofs of propositions that are classically, but not intuitionistically, valid.

This strong relationship between classical reasoning principles and control operators makes the $\lambda\mu\tilde{\mu}$ a good basis for compiler intermediate languages. Such a sequent calculus-based compiler intermediate language was already proposed by Downen, Maurer, Zena M Ariola, and Peyton Jones (2016), and we make the same point in our article Binder, Tzschentke, Müller, and Ostermann (2024). To develop the metatheory of the $\lambda\mu\tilde{\mu}$-calculus is the second main thread of this thesis.

1.3. Structure and Contributions

This thesis is divided into two parts. Part I discusses the more familiar term systems and languages based on natural deduction and the lambda calculus. Part II then goes beyond languages based on natural deduction and is concerned with symmetric systems based on the sequent calculus and the $\lambda\mu\tilde{\mu}$-calculus. The duality of data and codata types is a central theme in both parts.

Part I is called "A Natural Deduction Calculus for Data and Codata" and is concerned with systems whose term language is most similar to ordinary functional programming languages. I introduce an untyped system **ND** in chapter 2, together with its conversion and reduction theory, and present a full proof of the confluence of this system. I introduce a number of normal forms such as the normal form **NF**, the weak normal form **WNF**, the head normal form **HNF**, and the weak head normal form **WHNF**. These different normal forms correspond to different evaluation and normalization strategies, and I introduce the call-by-value evaluation strategy for weak normal forms and the call-by-name strategy for weak head normal forms explicitly.

The following chapter 3 introduces a type assignment system for simple types and the de- and refunctionalization algorithms which transpose between the data and codata fragment. That chapter is based on a publication that was accepted at the conference on the principles of programming languages (POPL); the core contribution of that publication and chapter 3 consists in proving, for the first time, that it is possible to define total, inverse and semantics-preserving defunctionalization and refunctionalization algorithms for languages which have local pattern and copattern matches. We only prove this theorem for the type system described in that chapter, which only has simple types. I solved the problem of extending the defunctionalization and refunctionalization algorithms to dependent types in the paper Binder, Skupin, Süberkrüb, and Ostermann (2024a), which, however, did not become a part of this thesis.

Part II is called "Sequent Calculi for Data and Codata"; the chapters included in that part introduce and discuss various aspects of the symmetric $\lambda\mu\tilde{\mu}$-calculus. At the core of the sequent calculus is a deep duality of proofs and refutations, producers and consumers, values and continuations that is hidden from view if we work in the framework of natural deduction and the lambda calculus. In a recent paper (Binder, Tzschentke, Müller, and Ostermann, 2024) I give a much more thorough introduction to the languages presented in part II. That paper is directed at compiler writers in particular, since it introduces the $\lambda\mu\tilde{\mu}$-calculus as a possible intermediate representation for programming languages with control effects.

Chapter 4 solves two problems that were left open at the end of chapter 3: The first problem is that in the natural deduction setting we have to specify two similar, but separate, defunctionalization and refunctionalization algorithms. In the symmetric setting of the $\lambda\mu\tilde{\mu}$-calculus this is no longer the case, and we can instead define one algorithm which subsumes both transformations. The second problem concerns the interaction of defunctionalization and refunctionalization with evaluation order. The system presented in chapter 3 assumes the call-by-value evaluation order for all types. As we have already seen, this is disadventageous if we are interested in a rich equational theory, since only the eta-laws for data types are valid under call-by-value. Chapter 4 solves this problem by introducing a nominal evaluation strategy: We can define for each type individually whether it should be evaluated strictly or on demand. It is then possible to define a defunctionalization and refunctionalization algorithm which changes both the polarity (i.e. data or codata) and the evaluation order of the type. In fact, this algorithm can be decomposed into two individual algorithms: one algorithm changes the polarity and the other algorithm changes the evaluation order by introducing shifts in the program.

Chapter 5 discusses the relationship between the format of reasoning and the corresponding programming style that we have already started to introduce in section 1.2. That chapter is based on the article Ostermann, Binder, Skupin, Süberkrüb, and Downen (2022a), which was published at the international conference on functional programming (ICFP). We start with the observation that there are four different kinds of rules in derivation systems based on sequents: left-introduction rules, right-introduction rules, left-elimination rules and right-elimination rules. The difference between left and right rules can be seen by the location of the prinicpal connective: if the connective appears on the left side of the sequent we have a left rule, and vice versa for right rules. The difference between introduction and elimination rules can be explained in the same way. In introduction rules the principal connective appears in the conclusion of the rule, while in elimination rules the principal connective appears in one of the premisses. We present all kinds of rules for each connective, but various subsets of the rules are functionally complete. We call these functionally complete subsets the introduction, elimination, left and right calculus, respectively. In that chapter we show how these different calculi correspond to different ways to write and structure programs, and that it can therefore be beneficial to support all these possible rules in a programming language.

Chapter 6 is of a very technical nature since it compares two different but related program transformations: the ANF transformation for the lambda calculus and the focusing transformation for the $\lambda\mu\tilde{\mu}$-calculus. People familiar with both transformations are aware that they are roughly similar, but there has been no formal proof yet which makes this correspondence precise. Making the relationship between the ANF transformation and focusing completely precise is the central contribution of chapter 6. In order to prove the correspondence we have to split the ANF transformation into two different stages which can be composed to obtain the original transformation. The first stage uses let-bindings to lift subcomputations into a position where they can be evaluated, and the second stage linearizes these resulting let-bindings. The first stage corresponds exactly to the focusing transformation for the $\lambda\mu\tilde{\mu}$-calculus, and the second stage corresponds to simple μ-reductions. These correspondences are then proved and all the details are spelled out.

Some of the chapters of this thesis are based on previous publications; a note at the beginning of each such chapter mentions this separately. I shall also shortly mention two other articles that were written during the time of my PhD and inspired by many of the themes I develop here. In the article Binder, Skupin, Läwen, and Ostermann (2022), accepted and published at the workshop for type driven development (TyDe), I developed a system of *structural refinement types*. The type system is based on the algebraic subtyping approach that was introduced by Dolan and Mycroft (2017) and Dolan (2017) and later simplified by Parreaux (2020). Structural refinement types combine properties of nominal data types and polymorphic variants, and allow to express and infer subtypes which are refinements of a given data type declaration. For example, we can infer the type of non-empty lists, which is a refinement of the type of lists. The article Bhanuka, Parreaux, Binder, and Brachthäuser (2023), accepted at OOPSLA'23, is also based on algebraic subtyping. In that article we show that the often inscrutable error messages that arise from constraint-based type inference can be significantly improved by using insights from the theory of algebraic subtyping. If we read any subtyping constraint that is generated during type inference as a statement about data that flows through the program, then we can explain errors that arise during typechecking by using the flow of data through the program as an explanatory device, a concept the programmer and user is likely familiar with.

Part I.

A Natural Deduction Calculus for Data and Codata

2. The Untyped Calculus ND

In this chapter, I am going to introduce the untyped calculus **ND** for data and codata types. In section 2.1 I introduce terms and the basic syntactic machinery needed to work with them. In section 2.2 I show how to faithfully embed the untyped lambda calculus into the calculus **ND**. Section 2.3 introduces the symmetric *conversion* relation $e_1 \equiv e_2$ between terms, which is then specialized to the directed *reduction* relation $e_1 \triangleright e_2$ in section 2.4. In section 2.5 I show that reduction satisfies the Church-Rosser property, and that conversion is, therefore, a consistent relation (i.e. not all terms of the language are convertible). Section 2.6 defines four different normal forms: normal forms, head normal forms, weak normal forms and weak head normal forms. In section 2.7 the calculus is extended with let bindings and a control operator, the resulting calculus is no longer confluent. Finally, in section 2.8 I introduce two deterministic evaluation orders, call-by-value evaluation **cbv** and call-by-name evaluation **cbn**.

2.1. Syntax of Terms

The syntax of terms is given in definition 2.1.1 and consists of variables x, constructors applied to arguments $K(\overline{e})$, the application of a destructor d with arguments $\overline{e}$ on a term e, which is written as $e.d(\overline{e})$, and two constructs for pattern and copattern matching. A pattern match $e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\}$ analyses a term e and tests it against a list of clauses. Each clause consists of a pattern made from a constructor applied to variables and an expression on the right-hand side. A copattern match $\textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\}$ consists of a list of clauses, each of which consists of a destructor applied to variables and an expression on the right-hand side.

We use the notation $\overline{X}$ to represent a (possibly empty) sequence $X_1, \ldots, X_i, \ldots, X_n$ and follow the Featherweight Java (Igarashi, Benjamin C Pierce, and Wadler, 2001) conventions for this notation. In this convention, multiple occurrences of such sequences should be read as being indexed simultaneously, e.g. the judgement "$\Gamma, \overline{\Pi} \vdash \overline{c}$" should be read as a list of judgements "$\Gamma, \Pi_i \vdash c_i$". We also sometimes omit the last element in a sequence and write $X_1, \ldots$ instead of $X_1, \ldots, X_n$ in situations where there is no ambiguity and writing out the complete sequence would make the examples excessively verbose. This is particularily the case in section 2.5.

Definition 2.1.1 (Terms of the Calculus ND)**.** The terms of the untyped natural deduction system are:

$$
\begin{array}{rcll}
e & ::= & x & \textit{Variables} \\
 & | & K(\overline{e}) & \textit{Constructor} \\
 & | & e.d(\overline{e}) & \textit{Destructor} \\
 & | & e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\} & \textit{Pattern match} \\
 & | & \textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\} & \textit{Copattern match}
\end{array}
$$

In pattern and copattern matches every constructor or destructor must occur at most once in the list of clauses.

Let us consider some examples which illustrate the use of constructors and pattern matches. The boolean values `True` and `False` are constructors which are applied to an empty list of arguments. A list which contains the two boolean truth values can similarly be represented using only constructors: $\texttt{Cons}(\texttt{True}, \texttt{Cons}(\texttt{False}, \texttt{Nil}))$. An if-then-else expression "**if** e_1 **then** e_2 **else** e_3" can be represented using pattern matching as $e_1.\textbf{case}\ \{\texttt{True} \Rightarrow e_2, \texttt{False} \Rightarrow e_3\}$, and an expression which tests whether a given list e is empty can be written as $e.\textbf{case}\ \{\texttt{Nil} \Rightarrow \texttt{True}, \texttt{Cons}(x, xs) \Rightarrow \texttt{False}\}$.

Destructors and copattern matching are less familiar, but we have already seen some examples in the introduction: A lazy pair which consists of the two expressions e_1 and e_2 is written $\textbf{cocase}\ \{\pi_1 \Rightarrow e_1, \pi_2 \Rightarrow e_2\}$. This pair waits for an observation of the form $e.\pi_1$ or $e.\pi_2$ which will force its evaluation and yield one of the expressions e_1 or e_2. Another example we have seen in the introduction is the stream type: a stream is defined by the two destructors `head` and `tail`. Given the stream $e = \textbf{cocase}\ \{\texttt{head} \Rightarrow e_1, \texttt{tail} \Rightarrow e_2\}$ we can either observe the first element e_1 of the stream with $e.\texttt{head}$ or request the remainder e_2 of the stream with $e.\texttt{tail}$. Another important example that uses the possibility for destructors to take arguments will be introduced in section 2.2: lambda abstractions and function applications.

2.1.1. Free Variables, Substitutions, Contexts

Before we come to the interesting conversion and reduction theory of the untyped calculus, we first have to put some basic bureaucracy into place. This bureaucracy concerns the treatment of free and bound variables, the concept of substitutions and the formal definition of subterm occurrences. We start with the most simple concept: the set of free variables of a term.

Definition 2.1.2 (Free Variables)**.** The set of free variables of a term e is written $\mathrm{FV}(e)$, and a term is called *closed* if this set is empty. The set $\mathrm{FV}(e)$ is defined recursively on the structure of terms:

$$
\begin{aligned}
\mathrm{FV}(x) &\coloneqq \{x\} \\
\mathrm{FV}(K(e_1, \ldots, e_n)) &\coloneqq \mathrm{FV}(e_1) \cup \ldots \cup \mathrm{FV}(e_n) \\
\mathrm{FV}(e.d(e_1, \ldots, e_n)) &\coloneqq \mathrm{FV}(e) \cup \mathrm{FV}(e_1) \cup \ldots \cup \mathrm{FV}(e_n) \\
\mathrm{FV}(e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\}) &\coloneqq \mathrm{FV}(e) \cup (\mathrm{FV}(e_1) \setminus \overline{x}) \cup \ldots \cup (\mathrm{FV}(e_n) \setminus \overline{x}) \\
\mathrm{FV}(\textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\}) &\coloneqq (\mathrm{FV}(e_1) \setminus \overline{x}) \cup \ldots \cup (\mathrm{FV}(e_n) \setminus \overline{x})
\end{aligned}
$$

The meaning of a closed term is independent of its context while the meaning of a term that contains free variables depends on some assignment of terms or values for the free variables it contains. In order to make this assignment explicit, we need to introduce the concept of substitutions. In the lambda calculus we usually only substitute one expression for one variable at a time. For the calculus **ND**, however, we frequently have to use a simultaneous substitution which substitutes many expressions for many variables[1]. When we reduce the expression $\texttt{Cons}(2, \texttt{Nil}).\mathbf{case}\ \{\texttt{Cons}(x, xs) \Rightarrow e_1, \texttt{Nil} \Rightarrow e_2\}$, for example, we substitute 2 for the variable x and Nil for the variable xs in the expression e_1. We write $e_1[2, \texttt{Nil}/x, xs]$ for this simultaneous substitution.

We will now define substitutions σ and the action of a substitution on a term, which is written $e\,\sigma$, separately. When we define the action of a substitution on a term, we have to take care to avoid the unintended capture of free variables in the terms we substitute.

Definition 2.1.3 (Substitution)**.** A simultaneous substitution σ of the terms $e_1, \ldots, e_n$ for the variables $x_1, \ldots, x_n$ is defined by the following grammar. We also require that all the x_i are distinct.

$$\sigma ::= [e_1, \ldots, e_n/x_1, \ldots, x_n]$$

Every substitution has both a domain and a range. The domain is the set of variables for which the substitution is defined whereas the range is the set of free variables of the expressions that appear in the substitution.

Definition 2.1.4 (Domain and Range of a Substitution)**.** We define the domain and the range of a substitution as the following sets of variables:

$$\begin{aligned}\texttt{dom}([e_1, \ldots, e_n/x_1, \ldots, x_n]) &\coloneqq \{x_1, \ldots, x_n\}\\ \texttt{rng}([e_1, \ldots, e_n/x_1, \ldots, x_n]) &\coloneqq \mathrm{FV}(e_1) \cup \ldots \cup \mathrm{FV}(e_n)\end{aligned}$$

These definitions just show what a substitution *is* but we are usually more interested in what a substitution *does*, i.e. what happens when we apply it to an expression. This is called the *action* of the substitution on a term.

Definition 2.1.5 (Action of a Substitution)**.** The action of a substitution σ on a term e is written $e\sigma$ and defined by the following clauses:

$$\begin{aligned}x[e_1, \ldots, e_n/x_1, \ldots, x_n] &\coloneqq e_i \quad (\text{if } x = x_i)\\ y\sigma &\coloneqq y \quad (\text{if } y \notin \texttt{dom}(\sigma))\\ (K(e_1, \ldots, e_n))\sigma &\coloneqq K(e_1\sigma, \ldots, e_n\sigma)\\ (e.d(e_1, \ldots, e_n))\sigma &\coloneqq (e\sigma).d(e_1\sigma, \ldots, e_n\sigma)\\ (e.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\})\sigma &\coloneqq (e\sigma).\mathbf{case}\ \{\overline{K(\overline{y}) \Rightarrow (e\sigma')\sigma}\}\\ (\mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\})\sigma &\coloneqq \mathbf{cocase}\ \{\overline{d(\overline{y}) \Rightarrow (e\sigma')\sigma}\}\end{aligned}$$

[1]As Stoughton (1988) showed, simultaneous substitutions also allow us to make the definition structurally recursive in the case where we have to rename bound variables.

In the last two clauses, we have to perform a renaming of bound variables with the help of the substitution σ'. This substitution σ' has the form $[y_1, \dots, y_n/x_1, \dots, x_n]$, where the y_i are fresh for both the domain and the range of σ.

When we have two substitutions σ_1 and σ_2 then we can either apply one after the other, or we can compute the composition of the two substitutions and apply the resulting substitution $\sigma_2 \circ \sigma_1$ instead.

Definition 2.1.6 (Composition of Substitutions)**.** Given the two substitutions

$$\sigma_1 \coloneqq [e_1, \dots, e_n/x_1, \dots, x_n] \qquad \sigma_2 \coloneqq [t_1, \dots, t_m/y_1, \dots, y_m]$$

we define their composition as

$$\sigma_2 \circ \sigma_1 \coloneqq [e_1\sigma_2, \dots, e_n\sigma_2, t_j, \dots, t_k/x_1, \dots, x_n, y_j, \dots, y_k]$$

where $j, \dots, k$ is the greatest sub-range of indices $1, \dots, m$ such that none of the variables y_j to y_k is in the domain of σ_1.

An important lemma that we need later states how to permute one substitution over another:

Lemma 2.1.7 (Permutation of Substitutions)**.** *For any terms $e, e_1, \dots, t_1, \dots$ we have:*

$$e[e_1, \dots/x_1, \dots][t_1, \dots/y_1, \dots] = e[t_1, \dots/y_1, \dots][e_1[t_1, \dots/x_1, \dots], \dots/x_1, \dots]$$

if all the variables $x_1, \dots$ are distinct from all the variables $y_1, \dots$.

Proof. By induction on the structure of e. □

We often have to decompose a term into a *subexpression* and its surrounding *context*. This is often done informally by underlining the subexpression that we want to highlight. For example, when we want to talk about the second element in a list we can mark it in the following way:

$$\texttt{Cons}(1, \texttt{Cons}(\underline{2}, \texttt{Cons}(3, \texttt{Nil})))$$

The subexpression 2 is just an ordinary term, but the surrounding context has to be specified using its own grammar. In this example, the context is written as follows, where we use the symbol □ to mark the place where we have to insert the subexpression.

$$\texttt{Cons}(1, \texttt{Cons}(\square, \texttt{Cons}(3, \texttt{Nil})))$$

We are only interested in contexts with one hole, which can be defined by the following grammar.

Definition 2.1.8 (Contexts)**.** Contexts with one hole are defined by the following grammar:

$$\begin{aligned} C ::= {} & \square \mid K(\overline{e}, C, \overline{e}) \mid C.d(\overline{e}) \mid e.d(\overline{e}, C, \overline{e}) \mid C.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\} \\ & \mid e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}, K(\overline{x}) \Rightarrow C, \overline{K(\overline{x}) \Rightarrow e}\} \\ & \mid \textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}, d(\overline{x}) \Rightarrow C, \overline{d(\overline{x}) \Rightarrow e}\} \end{aligned}$$

Since contexts have a hole there is a corresponding operation to plug an expression e into the hole $\Box$ of a context C. This plugging operation is written $C[e]$ and is different from the substitution of a term for a variable, since plugging a term into a hole can capture free variables of that term. For example, plugging the variable x into the hole of the context $e.\textbf{case}\ \{\texttt{Tup}(x, y) \Rightarrow \Box\}$ results in the expression $e.\textbf{case}\ \{\texttt{Tup}(x, y) \Rightarrow x\}$, where the variable x is now bound.

2.2. Embedding Lambda Calculus

In the examples we have seen so far we have not used functions, which are usually one of the first constructs which are introduced into a term language. In particular, the calculus **ND** does not contain lambda abstractions $\lambda x.e$ and function applications $e\ e$. This is not because we have forgotten to add them, but because the destructors and copattern matches that are available in **ND** subsume the lambda abstractions and function applications that are found in the lambda calculus. We show this by embedding the lambda calculus defined in definition 2.2.1 into our calculus.

Definition 2.2.1 (Terms of the Lambda Calculus)**.** There are three kinds of *lambda terms* Λ:

$$e_\lambda ::= x \mid \lambda x.e_\lambda \mid e_\lambda\ e_\lambda.$$

That is, we have *variables* x, *lambda abstractions* $\lambda x.e_\lambda$ and *function applications* $e_\lambda\ e_\lambda$.

For the embedding, we assume that there is a destructor $\texttt{Ap}(-)$ that we can use to encode lambda abstractions and function applications.

Definition 2.2.2 (Embedding of Lambda Terms)**.** We define an embedding of lambda terms $[\![-]\!] : \Lambda \to \textbf{ND}$ by the following clauses:

$$\begin{aligned}
[\![x]\!] &:= x \\
[\![\lambda x.e]\!] &:= \textbf{cocase}\ \{\texttt{Ap}(x) \Rightarrow [\![e]\!]\} \\
[\![e_1\ e_2]\!] &:= [\![e_1]\!].\texttt{Ap}([\![e_2]\!])
\end{aligned}$$

For example, the embedding of the term $(\lambda x.x)(\lambda y.z)$ is the term

$$(\textbf{cocase}\ \{\texttt{Ap}(x) \Rightarrow x\}).\texttt{Ap}(\textbf{cocase}\ \{\texttt{Ap}(y) \Rightarrow z\}).$$

Lambda terms are both familiar and syntactically more compact than the result of embedding them in **ND**. For this reason, we will use the syntax of lambda terms and function applications in the remainder of this thesis, especially in examples, but whenever we write $\lambda x.e$ or $e\ e$ we implicitly intend those terms to denote their embedding according to definition 2.2.2.

2.3. Conversion

So far, we have only used the syntactic equality of terms which we have written $e_1 = e_2$. Syntactic equality, however, is usually not what we are interested in when we investigate the metatheory of an untyped term language; we want an equivalence relation that equates more terms. This equivalence relation is called *conversion*, written $e_1 \equiv e_2$, and combines three ways in which terms from **ND** can be equated. Alpha renaming allows to identify terms that only differ in the choice of names for bound variables, beta conversion allows to perform computation, and eta conversion allows to identify extensionally equivalent terms. We will now introduce them in turn.

Let us first consider alpha renaming. The name of bound variables should not influence the meaning we assign to terms, that is, we do consider the two terms $\lambda x.x$ and $\lambda y.y$ to be the same. We can make this intuition explicit by allowing bound variables to be renamed, and we equate any two terms that can be made equal by such alpha-renamings.

Definition 2.3.1 (Alpha-Renaming)**.** A single step of alpha-renaming, written $e_1 \equiv^1_\alpha e_2$, is specified by the following two equations, where we assume that the y_i do not occur free in e.

$$e.\mathbf{case}\ \{\ldots, K(\overline{x}) \Rightarrow e, \ldots\} \equiv^1_\alpha e.\mathbf{case}\ \{\ldots, K(\overline{y}) \Rightarrow e[\overline{y}/\overline{x}], \ldots\} \tag{α-Data}$$
$$\mathbf{cocase}\ \{\ldots, d(\overline{x}) \Rightarrow e, \ldots\} \equiv^1_\alpha \mathbf{cocase}\ \{\ldots, d(\overline{y}) \Rightarrow e[\overline{y}/\overline{x}], \ldots\} \tag{α-Codata}$$

Next, we also want to equate two terms if one of the terms can be obtained from the other by a series of computation steps. These computation steps happen when we apply a destructor on a comatch which contains a clause for it, or when we pattern match on a constructor.

Definition 2.3.2 (Beta-Conversion)**.** A single step of beta-conversion, written $e_1 \equiv^1_\beta e_2$, is specified by the following two equations:

$$K(\overline{e}).\mathbf{case}\ \{\ldots, K(\overline{x}) \Rightarrow e\} \equiv^1_\beta e[\overline{e}/\overline{x}] \tag{β-Data}$$
$$\mathbf{cocase}\ \{\ldots, d(\overline{x}) \Rightarrow e, \ldots\}.d(\overline{e}) \equiv^1_\beta e[\overline{e}/\overline{x}] \tag{β-Codata}$$

These rules require that the constructor and destructor is applied to the same number of expressions and variables, respectively.

Example 2.3.3. The following two examples illustrate beta conversion for data and codata types:

$$\mathbf{cocase}\ \{\pi_1 \Rightarrow e_1, \pi_2 \Rightarrow e_2\}.\pi_1 \equiv^1_\beta e_1$$
$$[1,2].\mathbf{case}\ \{[x,y] \Rightarrow [y,x]\} \equiv^1_\beta [2,1]$$

Let us also consider some edge cases of beta conversion. In definition 2.1.1 we required that every constructor or destructor appears at most once in the list of clauses. This requirement rules out degenerate cases such as **cocase** $\{\pi_1 \Rightarrow e_1, \pi_1 \Rightarrow e_2\}.\pi_1$ where the copattern match contains two clauses for the destructor π_1. Excluding these kinds of degenerate cases is necessary to ensure that the calculus is confluent. What we cannot do in the untyped system, however, is to exclude terms such as **cocase** $\{\pi_1 \Rightarrow e_1\}.\pi_2$ where the copattern match does not contain a clause for the destructor that is invoked on it. Such expressions are called *stuck*, and it is the task of the type system to exclude the possibility of stuck terms.

The last component that we have to discuss is eta conversion which equates expressions with their corresponding canonical form. Applied to functions, it says that an expression e can be equated with its canonical form $\lambda x.e\ x$. Applied to booleans, it says that every expression e which contains a free variable x which stands for a boolean can be specialized to the two cases where x is either true or false: $x.\textbf{case}\ \{\texttt{True} \Rightarrow e[\texttt{True}/x], \texttt{False} \Rightarrow e[\texttt{False}/x]\}$. Definition 2.3.4 formally defines eta conversion for data and codata types, but we have specialized the η rule for data types.

Definition 2.3.4 (Eta-Conversion)**.** A single step of eta-conversion, written $e_1 \equiv^1_\eta e_2$, is specified by the following two equations:

$$e.\textbf{case}\ \{\overline{K(\overline{x}) \Rightarrow K(\overline{x})}\} \equiv^1_\eta e \qquad (\eta\text{-Data})$$

$$\textbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e.d(\overline{x})}\} \equiv^1_\eta e \quad (\text{if } \overline{x} \notin \text{FV}(e)) \qquad (\eta\text{-Codata})$$

Our definitions of alpha renaming and beta and eta conversion correspond exactly to the respective notions for the lambda calculus:

Remark 2.3.5 (Conversion in Lambda Calculus)**.**

$$\lambda x.e \equiv^1_\alpha \lambda y.e[y/x] \quad (\text{if } y \notin \text{FV}(e)) \qquad (\alpha\text{-}\lambda)$$

$$(\lambda x.e_1)e_2 \equiv^1_\beta e_1[e_2/x] \qquad (\beta\text{-}\lambda)$$

$$(\lambda x.e\,x) \equiv^1_\eta e \quad (\text{if } x \notin \text{FV}(e)) \qquad (\eta\text{-}\lambda)$$

So far, we have only defined how to perform a single step of alpha renaming, beta conversion and eta conversion. In order to obtain a sensible equivalence relation from these individual steps we have to compute their transitive, symmetric, reflexive closure. We also have to ensure that conversion is a congruence, i.e. that we are allowed to replace equivalent terms in arbitrary positions in a term. We obtain the closure by adding the rules Refl, Sym and Trans, and we ensure that the relation is a congruence by allowing to perform the individual steps inside of an arbitrary context C in the rules α, β and η.

Definition 2.3.6 (Conversion)**.** Two terms e_1 and e_2 are *convertible*, written $e_1 \equiv e_2$, if we can derive it using the following rules:

$$\frac{e_1 \equiv^1_\alpha e_2}{C[e_1] \equiv C[e_2]}\,\alpha \qquad \frac{e_1 \equiv^1_\beta e_2}{C[e_1] \equiv C[e_2]}\,\beta \qquad \frac{e_1 \equiv^1_\eta e_2}{C[e_1] \equiv C[e_2]}\,\eta$$

$$\frac{}{e \equiv e}\,\textsc{Refl} \qquad \frac{e_1 \equiv e_2}{e_2 \equiv e_1}\,\textsc{Sym} \qquad \frac{e_1 \equiv e_2 \quad e_2 \equiv e_3}{e_1 \equiv e_3}\,\textsc{Trans}$$

Whenever we write $e_1 \equiv e_2$, we mean the unrestricted use of the alpha, beta and eta rules. If we want to restrict ourselves only to a subset of the rules, then we indicate this in the subscripts by writing $\equiv_\beta$ or $\equiv_\alpha$ etc.

Conversion relations can be ordered by how many terms they equate. In that respect, syntactic equality equates the least amount of terms, since any term is only equated to itself and to no other term. If we include alpha renamings and beta conversion we obviously equate more terms, and if we also include eta-equivalence we are equating even more. What we have to guard against, however, is that we equate too many terms. Consistency is a minimal requirement that ensures that the conversion relation has not become trivial:

Definition 2.3.7 (Consistency)**.** A conversion relation $\equiv$ is *consistent* if there exist terms e_1 and e_2 such that $e_1 \not\equiv e_2$.

Luckily, we can prove that β-conversion is consistent and that the theory induced by alpha renamings and computation steps is therefore not trivial.

Theorem 2.3.8 (Beta-Conversion is Consistent)**.** *The relation $\equiv_{\alpha\beta}$ defined in definition 2.3.6 is consistent.*

Proof. This is a corollary of the Church-Rosser theorem that we will prove in section 2.5. □

The same does not hold for $\equiv_\eta$, however, which is inconsistent.

Theorem 2.3.9 (Eta-Conversion is Inconsistent)**.** *The relation $\equiv_\eta$ defined in definition 2.3.6 is not consistent, i.e. for any two terms e_1 and e_2 we have $e_1 \equiv_\eta e_2$.*

Proof. Consider the following derivation:

$$e_1 \equiv^1_\eta \textbf{cocase}\ \{\} \equiv^1_\eta e_2$$

□

The situation is different in the lambda calculus, where the addition of η-equality does not make conversion inconsistent. Let us first analyze why the situation is different, and then how eta-equality can be handled consistently in a calculus with data and codata types. Every term of the lambda calculus stands for a function. This justifies to read the η-equality $\lambda x.e\,x \equiv_\eta e$ from right to left and expand any term e to the term $\lambda x.e\,x$, which is the canonical form of a function. In a calculus with data and codata types the assumption that every term stands for a function (or any other concrete data or codata type for that matter) no longer holds. But in the proof of theorem 2.3.9 we implicitly used an assumption that every term stands for the negative unit type, i.e. a codata type with no destructors, and can hence be equated to its η-normal form **cocase** $\{\}$. The solution to this problem is simple: We only consider *typed* η-equivalence which requires that we check the type of a term before η-expanding it in order to see which η-equivalences are valid.

2.4. Reduction

The conversion relation $\equiv$ that we introduced in the previous section is symmetric, but the underlying one-step beta and eta-conversion have a natural direction. In this section, we introduce the reduction relation $\triangleright$ to capture this directionality. The rules for beta reduction are:

Definition 2.4.1 (Beta-Reduction)**.** A single step of beta-reduction, written $e_1 \triangleright^1_\beta e_2$, is specified by the following two rules:

$$K(\overline{e}).\mathbf{case}\ \{\ldots, K(\overline{x}) \Rightarrow e\} \triangleright^1_\beta e[\overline{e}/\overline{x}] \qquad (\beta\text{-Data})$$
$$\mathbf{cocase}\ \{\ldots, d(\overline{x}) \Rightarrow e, \ldots\}.d(\overline{e}) \triangleright^1_\beta e[\overline{e}/\overline{x}] \qquad (\beta\text{-Codata})$$

And similarly, we can specify the rules for eta reduction as follows:

Definition 2.4.2 (Eta-Reduction)**.** A single step of eta-reduction, written $e_1 \triangleright^1_\eta e_2$, is specified by the following two rules:

$$e.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow K(\overline{x})}\} \triangleright^1_\eta e \qquad (\eta\text{-Data})$$
$$\mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e.d(\overline{x})}\} \triangleright^1_\eta e \quad (\text{if } \overline{x} \notin \mathrm{FV}(e)) \qquad (\eta\text{-Codata})$$

In order to obtain the reduction relation $e_1 \triangleright e_2$ we form the reflexive transitive closure and ensure that the relation is a congruence, i.e. closed under contexts. Reduction is directed, so we do not include symmetry in those rules.

Definition 2.4.3 (Reduction)**.** The term e_1 reduces to e_2, written $e_1 \triangleright e_2$, if we can derive it using the following rules:

$$\frac{e_1 \triangleright^1_\beta e_2}{C[e_1] \triangleright C[e_2]}\ \beta \qquad \frac{e_1 \triangleright^1_\eta e_2}{C[e_1] \triangleright C[e_2]}\ \eta \qquad \frac{}{e \triangleright e}\ \textsc{Refl} \qquad \frac{e_1 \triangleright e_2 \quad e_2 \triangleright e_3}{e_1 \triangleright e_3}\ \textsc{Trans}$$

Similarly to conversion, we write $\triangleright$ whenever we mean both beta-reduction and eta-reduction and will use $\triangleright_\beta$ and $\triangleright_\eta$ whenever we only mean beta-reduction or eta-reduction.

Lemma 2.4.4. *For all e_1 and e_2, if $e_1 \triangleright e_2$, then $e_1 \equiv e_2$.*

Proof. By simple inspection of the rules. □

The Church-Rosser theorem, proved in the next section, gives a tool to go in the other direction: Whenever two terms e_1 and e_2 are convertible, then there exists a term e_3 such that both e_1 and e_2 reduce to it.

2.5. The Church-Rosser Theorem

The Church-Rosser theorem says that if a term e_1 reduces to the two terms e_2 and e_3, then we will always find a term e_4 such that both e_2 and e_3 reduce to it. This is usually depicted as the following diagram.

$$\begin{tikzcd}
e_1 \arrow[r, "\triangleright"] \arrow[d, "\triangleright"] & e_2 \arrow[d, dotted, "\triangleright"] \\
e_3 \arrow[r, dotted, "\triangleright"] & e_4
\end{tikzcd}$$

In this subsection I am going to prove the Church-Rosser theorem for β-reduction, following the proof strategy of Barendregt (1981) for the lambda-calculus. That proof strategy relies on the following important lemma: It is sufficient to prove the confluence of some relation whose transitive closure is the desired relation in order to prove that the relation is confluent.

Lemma 2.5.1. *The transitive closure of a confluent relation is confluent.*

Proof. We can prove this lemma pictorially by observing that we can paste together small confluent squares to obtain a larger confluent square.

$$\begin{tikzcd}
e_1 \arrow[r] \arrow[d] & \cdot \arrow[r, dashed] \arrow[d, dotted] & \cdot \arrow[r] \arrow[d, dotted] & e_2 \arrow[d, dotted] \\
\cdot \arrow[r, dotted] \arrow[d, dashed] & \cdot & \cdot \arrow[r, dotted] & \cdot \arrow[d, dashed] \\
\cdot \arrow[r, dotted] \arrow[d] & \cdot \arrow[d, dotted] & \cdot \arrow[r, dotted] \arrow[d, dotted] & \cdot \arrow[d, dotted] \\
e_3 \arrow[r, dotted] & \cdot \arrow[r, dashed] & \cdot \arrow[r, dotted] & e_4
\end{tikzcd}$$

More formally, we prove the theorem by induction on the number of steps taken in each direction. □

There is no obvious direct way to show that the relation $\rhd_\beta$ is confluent. But lemma 2.5.1 presents us with a blueprint for how we can proceed: We first have to find some other relation that is confluent, and whose transitive closure is exactly $\rhd_\beta$. The first candidate that comes to mind is the relation $\rhd^1_\beta$ which does only one step of beta-reduction. This relation, however, is not confluent, as the following example shows. (We use the translation from definition 2.2.2 and the defined term $I = \lambda x.x$ to keep the example readable.)

$$\begin{tikzcd}
(\lambda x.x\,x)(I\,I) \arrow[r, "\rhd^1_\beta"] \arrow[d, "\rhd^1_\beta"] & (\lambda x.x\,x)I \arrow[d, dotted, "\rhd^1_\beta"] \\
(I\,I)(I\,I) \arrow[r, dotted, "\rhd^1_\beta"] & ?
\end{tikzcd}$$

There is no term to which both $(I\,I)(I\,I)$ and $(\lambda x.x\,x)I$ reduce in one step, so the relation $\rhd^1_\beta$ cannot be confluent. What we have to do instead is to define a notion of *parallel reduction*[2] which is confluent and whose transitive closure is precisely the relation $\rhd_\beta$.

Definition 2.5.2 (Parallel Reduction)**.** We write $e_1 \blacktriangleright e_2$ for the parallel reduction of redexes in e_1. This relation is defined by the following derivation rules:

$$\frac{}{e \blacktriangleright e}\ \textsc{Refl} \qquad \frac{e_1 \blacktriangleright e_1' \ldots}{K(e_1,\ldots) \blacktriangleright K(e_1',\ldots)}\ \textsc{Cong}_1 \qquad \frac{e \blacktriangleright e' \qquad e_1 \blacktriangleright e_1' \ldots}{e.d(e_1,\ldots) \blacktriangleright e'.d(e_1',\ldots)}\ \textsc{Cong}_2$$

$$\frac{e_1 \blacktriangleright e_1' \ldots}{\textbf{cocase}\ \{d_1(x,\ldots) \Rightarrow e_1,\ldots\} \blacktriangleright \textbf{cocase}\ \{d_1(x,\ldots) \Rightarrow e_1',\ldots\}}\ \textsc{Cong}_3$$

$$\frac{e \blacktriangleright e' \qquad e_1 \blacktriangleright e_1' \ldots}{e.\textbf{case}\ \{K_1(x,\ldots) \Rightarrow e_1,\ldots\} \blacktriangleright e'.\textbf{case}\ \{K_1(x,\ldots) \Rightarrow e_1',\ldots\}}\ \textsc{Cong}_4$$

$$\frac{e \blacktriangleright e' \qquad e_1 \blacktriangleright e_1' \ldots}{\textbf{cocase}\ \{d(x_1,\ldots) \Rightarrow e,\ldots\}.d(e_1,\ldots) \blacktriangleright e'[e_1',\ldots/x_1,\ldots]}\ \beta\text{-}\textsc{Codata}$$

$$\frac{e \blacktriangleright e' \qquad e_1 \blacktriangleright e_1' \ldots}{K(e_1,\ldots).\textbf{case}\ \{K(x_1,\ldots) \Rightarrow e,\ldots\} \blacktriangleright e'[e_1',\ldots/x_1,\ldots]}\ \beta\text{-}\textsc{Data}$$

In order to keep this definition and the following proofs readable and compact we use some syntactic abbreviations. Instead of always specifying both ends of a list of arguments (i.e. $e_1,\ldots,e_n$) we often omit the right end and simply write $e_1,\ldots$. And instead of specifying all the premisses $e_1 \blacktriangleright e_1'$ to $e_n \blacktriangleright e_n'$ we simply write $e_1 \blacktriangleright e_1' \ldots$

[2] Parallel reductions were introduced by Takahashi (1995) to give an inductively defined relation that is confluent. Previous versions of the confluence proof by Tait and Martin-Löf relied on a notion of residuals that was not defined inductively on the structure of terms.

The intuition behind this relation is that it allows to reduce all redexes in a term in parallel. In the example above the term $(\lambda x.x\,x)(I\,I)$ contains the (overlapping) redexes $(\lambda x.x\,x)(I\,I)$ and $I\,I$. We can reduce them in parallel to obtain the result $(\lambda x.x\,x)(I\,I) \blacktriangleright I\,I$. Parallel reduction is strictly weaker than beta reduction $\rhd_\beta$ since it does not allow to reduce redexes that are not contained in the original term. For example, the relation $(\lambda x.x\,x)(I\,I) \blacktriangleright I$ does not hold, since the term $x\,x$ is not initially a redex. Parallel reduction is not a deterministic relation, since the rule Refl allows not to reduce a redex.

The first lemma states that terms in head-normal form[3] keep their outer shape, i.e. outermost constructors or copattern matches are not changed under parallel reduction.

Lemma 2.5.3 (Properties of Parallel Reduction)**.** *The following statements hold:*

1. *If $K(e_1, \ldots) \blacktriangleright e$ then e has the form $K(e'_1, \ldots)$ and $e_1 \blacktriangleright e'_1, \ldots$.*
2. *If* **cocase** $\{d_1(x, \ldots) \Rightarrow e_1, \ldots\} \blacktriangleright e$ *then e has the form* **cocase** $\{d_1(x, \ldots) \Rightarrow e'_1, \ldots\}$ *and $e_1 \blacktriangleright e'_1, \ldots$.*

Proof. In the first case only the rules Refl and Cong$_1$ apply, and in the second case only the rules Refl and Cong$_3$ apply. □

The next lemma that we have to prove is that parallel reduction is compatible with substitution. This means that for a substitution $e[e_1, \ldots/x_1, \ldots]$ we can always reduce the terms e and e_i before substituting them, since the resulting term can also be obtained using parallel reductions.

Lemma 2.5.4 (Parallel Reduction is Compatible with Substitution)**.** *If $e \blacktriangleright e'$ and $e_1 \blacktriangleright e'_1 \ldots$, then $e[e_1, \ldots/x_1, \ldots] \blacktriangleright e'[e'_1, \ldots/x_1, \ldots]$.*

Proof. The proof proceeds by induction on the size of e and a case analysis of the possible derivations of $e \blacktriangleright e'$. We have to distinguish the following cases:

- Case Refl: We have to prove that $e[e_1, \ldots/x_1, \ldots] \blacktriangleright e[e'_1, \ldots/x_1, \ldots]$. We do this by induction on the structure of e.
- Case Cong: We show this for Cong$_1$, since the other congruence cases are similar. In this case, e has the form $K(t_1, \ldots)$ and e' has the form $K(t'_1, \ldots)$. We therefore have to show that $K(t_1, \ldots)[e_1, \ldots/x_1, \ldots] = K(t_1[e_1, \ldots/x_1, \ldots], \ldots)$ reduces to $K(t'_1, \ldots)[e'_1, \ldots/x_1, \ldots] = K(t'_1[e'_1, \ldots/x_1, \ldots], \ldots)$. This follows from an application of the rule Cong$_1$ and the induction hypothesis applied to the smaller terms t_i.
- Case β-Codata: In this case e has the form **cocase** $\{d(y_1, \ldots), \ldots \Rightarrow e_r\}.d(t_1, \ldots)$ and e' has the form $e'_r[t'_1, \ldots/y_1, \ldots]$, where $e_r \blacktriangleright e'_r$ and $t_1 \blacktriangleright t'_1, \ldots$. We assume without loss of generality that the variables $x_1, \ldots$ and $y_1, \ldots$ are distinct, and that

[3]Head normal forms are formally defined in section 2.6.

we don't have to perform alpha renamings when we substitute in the copattern match. In that case, the term

$$\textbf{cocase } \{d(y_1, \ldots), \ldots \Rightarrow e_r\}.d(t_1, \ldots)[e_1, \ldots / x_1, \ldots]$$

is the same as

$$\textbf{cocase } \{d(y_1, \ldots) \Rightarrow e_r[e_1, \ldots / x_1, \ldots]\}.d(t_1[e_1, \ldots / x_1, \ldots], \ldots).$$

We have to show that this reduces to $e'_r[t'_1, \ldots / y_1, \ldots][e'_1, \ldots / x_1, \ldots]$. By lemma 2.1.7 and our variable assumption above this term is the same as

$$e'_r[e'_1, \ldots / x_1, \ldots][t'_1[e'_r, \ldots / x_1, \ldots], \ldots / y_1, \ldots].$$

We can finish the proof by using the rule β-Codata and the induction hypothesis applied to the smaller formulas e_r and t_i.

- Case β-Data: This case is similar to the case β-Codata.

□

Lemma 2.5.5 (Parallel Reduction is Confluent). *For any e_1, e_2, e_3, if $e_1 \blacktriangleright e_2$ and $e_1 \blacktriangleright e_3$, then there exists an e_4 such that $e_2 \blacktriangleright e_4$ and $e_3 \blacktriangleright e_4$:*

$$\begin{tikzcd} e_1 \arrow[r, "\blacktriangleright"] \arrow[d, "\blacktriangleright"] & e_2 \arrow[d, dotted, "\blacktriangleright"] \\ e_3 \arrow[r, dotted, "\blacktriangleright"] & e_4 \end{tikzcd}$$

Proof. We prove this lemma by induction on the derivation of $e_1 \blacktriangleright e_2$. The cases for Refl, Cong$_1$ and Cong$_3$ are simple:

- Case Refl: Since e_1 is equal to e_2 we can choose e_4 to be e_3. The following diagram can then easily be seen to commute:

$$\begin{tikzcd} e_1 \arrow[r, "\blacktriangleright"] \arrow[d, "\blacktriangleright"] & e_1 \arrow[d, dotted, "\blacktriangleright"] \\ e_3 \arrow[r, dotted, "\blacktriangleright"] & e_3 \end{tikzcd}$$

- Case Cong$_1$: We know that e_1 has the form $K(t_1, \ldots)$ and that it reduces to $K(t'_1, \ldots)$ with $t_1 \blacktriangleright t'_1 \ldots$. By lemma 2.5.3 we also know that e_3 has the form $K(t''_1, \ldots)$ with $t_1 \blacktriangleright t''_1 \ldots$. By the induction hypothesis, we know that there exists an t'''_1 such that $t'_1 \blacktriangleright t'''_1$ and $t''_1 \blacktriangleright t'''_1$. We can thus choose e_4 to be $K(t'''_1, \ldots)$:

$$\begin{tikzcd} K(e_1, \ldots) \arrow[r, "\blacktriangleright"] \arrow[d, "\blacktriangleright"] & K(e_1', \ldots) \arrow[d, dotted, "\blacktriangleright"] \\ K(e_1'', \ldots) \arrow[r, dotted, "\blacktriangleright"] & K(e_1''', \ldots) \end{tikzcd}$$

- Case Cong$_3$: This case is similar to the case for Cong$_1$.

The remaining cases are a bit more difficult. We will present the cases involving the rules Cong$_2$ and β-Codata, that is those cases where e_1 is of the form $e.d(t_1, \ldots)$. The cases involving the rules Cong$_4$ and β-Data where e_1 has the form $e.\textbf{case}\ \{\ldots\}$ are similar. We have to distinguish the following three cases:

- Case Cong$_2$-Cong$_2$: If e_1 has been reduced to e_2 and e_3 by the rule Cong$_3$, then we know that $e \blacktriangleright e'$, $e \blacktriangleright e''$, $t_1 \blacktriangleright t_1' \ldots$ and $t_1 \blacktriangleright t_1''$. By the induction hypothesis, we know that appropriate e''' and $t_1''' \ldots$ exist, and we can choose $e'''.d(t_1''', \ldots)$ for e_4.

$$\begin{tikzcd} e.d(t_1, \ldots) \arrow[r, "\blacktriangleright"] \arrow[d, "\blacktriangleright"] & e'.d(t_1', \ldots) \arrow[d, dotted, "\blacktriangleright"] \\ e''.d(t_1'', \ldots) \arrow[r, dotted, "\blacktriangleright"] & e'''.d(t_1''', \ldots) \end{tikzcd}$$

- Case Cong$_2$-β-Codata: In this case we know that e_1 must have the outer form $\textbf{cocase}\ \{d(x_1, \ldots) \Rightarrow e, \ldots\}.d(t_1, \ldots)$. Without loss of generality we assume that e_2 has the form $\textbf{cocase}\ \{d(x_1, \ldots) \Rightarrow e', \ldots\}.d(t_1', \ldots)$ where where $e \blacktriangleright e'$ and $t_1 \blacktriangleright t_1'$, and that e_3 has the form $e''[t_1'', \ldots / x_1, \ldots]$ with $e \blacktriangleright e''$ and $t_1 \blacktriangleright t_1''$. The induction hypothesis ensures that there exists a term e''' with $e' \blacktriangleright e'''$ and $e'' \blacktriangleright e'''$, and a term t_1''' with $t_1' \blacktriangleright t_1'''$ and $t_1'' \blacktriangleright t_1'''$. We choose e_4 to be the term $e'''[t_1''', \ldots / x_1, \ldots]$; the reduction along the bottom of the resulting square holds by lemma 2.5.4, and the reduction along the right side holds according to rule β-Codata and lemma 2.5.4.

$$\begin{tikzcd} \textbf{cocase}\ \{d(x_1, \ldots) \Rightarrow e, \ldots\}.d(t_1, \ldots) \arrow[r, "\blacktriangleright"] \arrow[d, "\blacktriangleright"] & \textbf{cocase}\ \{d(x_1, \ldots) \Rightarrow e', \ldots\}.d(t_1', \ldots) \arrow[d, dotted, "\blacktriangleright"] \\ e''[t_1'', \ldots / x_1, \ldots] \arrow[r, dotted, "\blacktriangleright"] & e'''[t_1''', \ldots / x_1, \ldots] \end{tikzcd}$$

- Case β-Codata-β-Codata: In this case we know that e_1 must have the form $\textbf{cocase}\ \{d(x_1, \ldots) \Rightarrow e, \ldots\}.d(t_1, \ldots)$, and that this outer redex is reduced in both cases. The term e_2 must have the form $e'[t_1', \ldots / x_1, \ldots]$ where $e \blacktriangleright e'$ and $t_1 \blacktriangleright t_1'$. Similarly, the term e_3 must have the form $e''[t_1'', \ldots / x_1, \ldots]$ where $e \blacktriangleright e''$ and

$t_1 \blacktriangleright t_1''$. The induction hypothesis ensures that there exists a term e''' with $e' \blacktriangleright e'''$ and $e'' \blacktriangleright e'''$, and a term t_1''' with $t_1' \blacktriangleright t_1'''$ and $t_1'' \blacktriangleright t_1'''$. We choose e_4 to be the term $e'''[t_1''', \dots / x_1, \dots]$; the lower and rightmost reduction hold by lemma 2.5.4.

$$\begin{tikzcd}
\textbf{cocase } \{d(x_1, \dots) \Rightarrow e, \dots\}.d(t_1, \dots) \arrow[r, "\blacktriangleright"] \arrow[d, "\blacktriangleright"] & e'[t_1', \dots / x_1, \dots] \arrow[d, dotted, "\blacktriangleright"] \\
e''[t_1'', \dots / x_1, \dots] \arrow[r, dotted, "\blacktriangleright"] & e'''[t_1''', \dots / x_1, \dots]
\end{tikzcd}$$

□

Lemma 2.5.6 (Transitive Closure of Parallel Reduction is Beta Reduction)**.** *The transitive closure of the relation* $\blacktriangleright$ *is the relation* $\triangleright_\beta$*.*

Proof. To prove that the transitive closure of $\blacktriangleright$ is contained in $\triangleright_\beta$ it is sufficient to verify that $\triangleright_\beta^1$ is contained in $\blacktriangleright$. To prove that $\triangleright_\beta$ is contained in the transitive closure of $\blacktriangleright$ it is sufficient to verify that one step of parallel reduction can be expressed by multiple steps of $\triangleright_\beta^1$ reduction. Both of these facts can easily be seen by inspecting the definition of parallel reduction. □

Theorem 2.5.7 (Confluence of Beta Reduction)**.** *For any* e_1, e_2, e_3*, if* $e_1 \triangleright_\beta e_2$ *and* $e_1 \triangleright_\beta e_3$*, then there exists an* e_4 *such that* $e_2 \triangleright_\beta e_4$ *and* $e_3 \triangleright_\beta e_4$*.*

$$\begin{tikzcd}
e_1 \arrow[r, "\triangleright_\beta"] \arrow[d, "\triangleright_\beta"] & e_2 \arrow[d, dotted, "\triangleright_\beta"] \\
e_3 \arrow[r, dotted, "\triangleright_\beta"] & e_4
\end{tikzcd}$$

Proof. This follows from lemmas 2.5.1, 2.5.5 and 2.5.6. □

The Church-Rosser theorem has some important corollaries. The first corollary says that we can use reduction to check for convertibility:

Corollary 2.5.8 (Conversion Implies Reduction)**.** For all terms e_1 and e_2, if $e_1 \equiv_\beta e_2$, then there exists a term e_3 such that $e_1 \triangleright_\beta e_3$ and $e_2 \triangleright_\beta e_3$.

Proof. See Hindley and Seldin (2008, Theorem 1.41). □

From this corollary we can deduce that beta-conversion is consistent:

Corollary 2.5.9 (Beta Conversion is Consistent)**.** The conversion relation $\equiv_\beta$ is consistent, i.e. there exist terms e_1 and e_2 such that $e_1 \equiv_\beta e_2$ is not the case.

Proof. In order to prove this we just have to pick two different beta normal forms. Assume that they are convertible; by corollary 2.5.8 there must be a third term such that both normal forms reduce to it. This is absurd, since normal forms do not contain redexes. □

2.6. Normal Forms

In the previous section we have already referred twice to the concept of a *normal form*; in this section we make this concept formal. There are many different normal forms that can be defined for terms. Here, we are only considering normal forms with respect to the presence of β-redexes[4].

The most important normal form is the β-normal form **NF** which does not contain any β-redexes. But this restriction can be relaxed, and we can allow redexes to occur at specific places. If we allow for the presence of redexes under binders, then we obtain a weak normal form **WNF**. These weak normal forms are important for programming languages, since it is often not possible to efficiently evaluate under a binder, such as in the body of a local higher-order function. If we allow for the presence of redexes in arguments to constructors and destructors, then we obtain a head normal form **HNF**. If we combine both, i.e. if we allow for redexes both under binders and in arguments of constructors and destructors, then we obtain a weak head normal form **WHNF**. Weak head normal forms are especially important for lazy programming languages such as Haskell. The following table, inspired by a similar presentation of Sestoft (2001), summarizes the different normal forms.

	Reduce arguments	Don't reduce arguments
Reduce under binder	**NF**	**HNF**
Don't reduce under binder	**WNF**	**WHNF**

How can these various normal forms be specified syntactically? We can use the concept of *neutral terms n*, which are terms that can not be reduced since they contain a variable in a position which is scrutinized.

Definition 2.6.1 (Normal Form)**.** The normal form **NF** is defined by:

$$
\begin{aligned}
n &::= x \mid n.d(\overline{v}) \mid n.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow v}\} \\
v &::= n \mid K(\overline{v}) \mid \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow v}\}
\end{aligned}
$$

Definition 2.6.2 (Weak Normal Form)**.** The weak normal form **WNF** is defined by:

$$
\begin{aligned}
n &::= x \mid n.d(\overline{v}) \mid n.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\} \\
v &::= n \mid K(\overline{v}) \mid \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\}
\end{aligned}
$$

Definition 2.6.3 (Head Normal Form)**.** The head normal form **HNF** is defined by:

$$
\begin{aligned}
n &::= x \mid n.d(\overline{e}) \mid n.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow v}\} \\
v &::= n \mid K(\overline{e}) \mid \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow v}\}
\end{aligned}
$$

[4]In contrast to normal forms which also consider η-redexes, such as η-long or η-short normal forms.

Definition 2.6.4 (Weak Head Normal Form)**.** The weak head normal form **WHNF** is defined by:

$$
\begin{aligned}
n &::= x \mid n.d(\overline{e}) \mid n.\mathbf{case}\ \{\overline{K(\overline{x}) \Rightarrow e}\} \\
v &::= n \mid K(\overline{e}) \mid \mathbf{cocase}\ \{\overline{d(\overline{x}) \Rightarrow e}\}
\end{aligned}
$$

These different normal forms are important once we consider different evaluation strategies in section 2.8. Call-by-value evaluation strategies reduce to weak normal forms, whereas call-by-name evaluation strategies reduce to weak head normal forms. Normal forms which require that redexes don't occur under binders are less relevant for programming languages, but essential for proof assistants, since they have to perform complete normalization of terms during typechecking.

2.7. Let Bindings and Control Operators

In this section we extend the language with two new constructs: let bindings and control operators. It might, at first sight, seem strange to group them in one subsection, but there is an underlying reason why we present them together. That reason becomes clear once we consider the symmetric calculus in part II, where let bindings and control operators can be seen to be perfectly dual to each other. This deep duality, hidden in natural deduction and the lambda calculus but completely obvious in the sequent calculus and the $\lambda\mu\tilde{\mu}$-calculus, can be intuitively explained as follows: Let bindings allow to name a *proof* and bind it to a variable, whereas control operators allow to name a *refutation* or *continuation* and bind it to a covariable. We have discussed this duality in more detail in our recent article Binder, Tzschentke, Müller, and Ostermann, 2024.

2.7.1. Let Bindings

Let bindings allow to name a term and bind it to a variable. This is usually written as $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$, where we name the term e_1 and bind it to the variable x in the term e_2. Most functional programming languages support some form of local let bindings, since using let bindings usually makes code more readable. But there are also more theoretical reasons why we should study let bindings explicitly:

Let bindings can be used to give a more fine-grained analysis of beta-reduction. In those more fine-grained analyses we replace the usual rule of beta reduction $(\lambda x.e_1)\ e_2 \triangleright_\beta e_1[e_2/x]$ with the rule $(\lambda x.e_1)\ e_2 \triangleright_\beta \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$. This technique is used, for example, by Zena M. Ariola, Maraist, Odersky, Felleisen, and Wadler (1995, Section 5.1) to give an equational presentation of the call-by-need lambda calculus.

Another important use of let bindings is in compiler intermediate languages. Some compilers use the ANF transformation introduced by Sabry and Felleisen (1992) to make the evaluation order explicit. The nested arithmetic expression $(2 + 4) * (3 + 7)$, for example, would be translated into the form $\mathbf{let}\ x = 2 + 4\ \mathbf{in}\ (\mathbf{let}\ y = 3 + 7\ \mathbf{in}\ x * y)$. I will discuss the ANF transformation in much more detail in chapter 6.

Definition 2.7.1 (Let Bindings)**.** We extend the syntax of expressions (Definition 2.1.1) and contexts (Definition 2.1.8) as follows:

$$\begin{aligned} e &::= \ldots \mid \textbf{let } x = e \textbf{ in } e \\ C &::= \ldots \mid \textbf{let } x = C \textbf{ in } e \mid \textbf{let } x = e \textbf{ in } C \end{aligned}$$

We also extend the definition of free variables and the action of a substitution in the obvious way.

We can define alpha renaming, beta-conversion and eta-conversion for let bindings, similar to how we defined them for data and codata types.

Definition 2.7.2 (Conversion for Let-Bindings)**.**

$$\textbf{let } x = e_1 \textbf{ in } e_2 \equiv^1_\alpha \textbf{let } y = e_1 \textbf{ in } e_2[y/x] \quad (\text{if } y \notin \mathrm{FV}(e_2)) \tag{α-Let}$$

$$\textbf{let } x = e_1 \textbf{ in } e_2 \equiv^1_\beta e_2[e_1/x] \tag{β-Let}$$

$$\textbf{let } x = e \textbf{ in } x \equiv^1_\eta e \tag{η-Let}$$

The rule of eta conversion is interesting, since it seems that we can also express its content through beta-conversion, but there are two important caveats. First, once we consider evaluation order the rule of beta conversion is restricted to values and has the form $\textbf{let } x = v \textbf{ in } e \equiv^1_\beta e[v/x]$, but the eta-rule is still valid in its unrestricted form. And second, the corresponding rule in the sequent calculus states that $\tilde{\mu}x.\langle\, x \mid c \,\rangle \equiv_\eta c$, which cannot be expressed using beta-conversion alone.

The reduction rules for let bindings are just the directed variants of the rules of conversion in definition 2.7.2.

Definition 2.7.3 (Reduction for Let-Bindings)**.**

$$\textbf{let } x = e_1 \textbf{ in } e_2 \triangleright^1_\beta e_2[e_1/x] \tag{β-Let}$$

$$\textbf{let } x = e \textbf{ in } x \triangleright^1_\eta e \tag{η-Let}$$

The addition of let expressions to a language is mostly harmless, but the addition of control operators presented in the next subsection is more challenging.

2.7.2. Control Operators

There are many reasons to introduce control operators to programming languages. One of those reasons is performance; the following example from our recent article Binder, Tzschentke, Müller, and Ostermann (2024) illustrates this point. Consider this small program which multiplies all the numbers contained in a list[5]:

$$\textbf{def } \mathrm{mul}(xs) \coloneqq xs.\textbf{case } \{\texttt{Nil} \Rightarrow 0; \texttt{Cons}(x, xs) \Rightarrow x * \mathrm{mul}(xs)\}$$

This function works correctly, but it is inefficient if the list contains a zero. In that case it would be better to abort the computation which always traverses the entire list and directly return with zero as a final result. Control operators allow us to implement this behaviour in the following way:

$$\begin{array}{l}\textbf{def } \mathrm{mul}(xs) \coloneqq \textbf{label } \alpha\ \{\mathrm{mul'}(xs; \alpha)\} \\ \textbf{def } \mathrm{mul'}(xs; \alpha) \coloneqq xs.\textbf{case } \{\texttt{Nil} \Rightarrow 1; \texttt{Cons}(y, ys) \Rightarrow \textbf{ifz}(y, \textbf{goto}(0; \alpha), y * \mathrm{mul'}(xs; \alpha))\}\end{array}$$

The control operators that are used in this example are **label** and **goto**, and they allow to jump with the goto expression to a label α in a surrounding context. We use the expression **label** α $\{\mathrm{mul'}(xs; \alpha)\}$ in order to introduce a label α that we can jump to in the recursive helper function mult'. In the helper function mult' we check with the function **ifz** if the element y is zero, and either jump to the outer label with **goto**$(0; \alpha)$ or otherwise perform a normal recursive call $y * \mathrm{mul'}(xs; \alpha)$. This implementation shows the desired behaviour: We do not traverse the tail of the list once we have encountered a zero. We will see another example, the parsimonious filter function, where classical reasoning principles allow to write the efficient variant of a function more easily in chapter 5.

Definition 2.7.4 (Control Operator)**.** We extend the syntax of expressions (Definition 2.1.1) and contexts (Definition 2.1.8) as follows:

$$\begin{aligned} e &::= \ldots \mid \textbf{label } \alpha\ \{e\} \mid \textbf{goto}(e; \alpha) \\ C &::= \ldots \mid \textbf{label } \alpha\ \{C\} \mid \textbf{goto}(C; \alpha) \end{aligned}$$

We will not give a fully formal definition of the conversion and reduction rules for **label** and **goto**, because it is much simpler to study the metatheory of control operators in the $\lambda\mu\tilde{\mu}$-calculus, which is one of the main reasons to use the $\lambda\mu\tilde{\mu}$-calculus as a basis for compiler intermediate languages. Instead, we can define the meaning of these two control operators by translating them to the $\lambda\mu\tilde{\mu}$-calculus; this translation can be found in Binder, Tzschentke, Müller, and Ostermann, 2024, Section 2.6. Here we will only present some approximate rules which allow to get an intuitive understand of their operational behaviour. This operational behaviour is specified by the following two equations:

[5]We assume some primitives for number literals and arithmetic operations in this example.

$$\textbf{label } \alpha \ \{\ldots \textbf{goto}(e;\alpha)\ldots\} \triangleright e \tag{2.1}$$
$$\textbf{label } \alpha \ \{e\} \triangleright e \tag{2.2}$$

Equation (2.1) handles the case where we want to jump to the label α with the expression e. The problem with making this rule fully formal is that we have to ensure that the intermediate context between the label and the goto expression, here indicated by an ellipsis, does not contain another label which brings the same covariable α into scope. Equation (2.2) handles the case where the inner expression e does not contain a goto expression, and where we can therefore drop the label that surrounds the expression e. In order to make this formal we have to define a function which computes the free covariables contained in an expression, and require that the expression e does not contain the free covariable α.

We discuss the intricacies of the control operators **label** and **goto**, their translation into the sequent calculus and how they compare to other control operators in more detail in Binder, Tzschentke, Müller, and Ostermann (2024, Section 5.3).

2.7.3. Loss of confluence

In section 2.5 we showed that the system which only contains the expressions from definition 2.1.1 is confluent. The system that we obtain by adding let bindings and control operators is no longer confluent, as the following example shows:

$$\textbf{label } \alpha \ \{\textbf{let } x = \textbf{goto}(1;\alpha) \textbf{ in } 2\}$$

If we reduce the inner let expression first then we obtain the expression **label** α $\{2\}$ which can be further reduced to the value 2 by eq. (2.2). On the other hand, we can use eq. (2.1) to reduce the expression in one step to the value 1. These two values are clearly unrelated, and there is no third term to which both reduce.

What are we to make of this situation? The solution to this problem can be found by analyzing it in terms of different evaluation orders. Call-by-value evaluation orders only allow to substitute values for variables, and the expression $\textbf{goto}(1;\alpha)$ is clearly not a value; so under call-by-value we should obtain the final value 1. Using the call-by-name evaluation order, on the other hand, requires us to substitute the expression $\textbf{goto}(1;\alpha)$ for x in 2, so we end up with the end result 2.

2.8. Evaluation Orders

In section 2.7.3 we saw that the addition of control operators led to the loss of confluence. The loss of confluence in effectful languages leads us to consider *evaluation strategies* or *evaluation orders*. Evaluation orders solve the confluence problem by brute force; they don't allow any divergent computations, since there is always at most one redex that can be evaluated. Let us first formally define what an evaluation strategy is.

Definition 2.8.1 (Deterministic Relation)**.** A binary relation $R \subseteq \mathcal{P}(A \times B)$ is called *deterministic* if for any $a \in A$ and $b_1, b_2 \in B$, $(a, b_1) \in R$ and $(a, b_2) \in R$ imply $b_1 = b_2$.

Definition 2.8.2 (Evaluation Strategy)**.** An evaluation strategy is a deterministic subset of the reduction relation $\triangleright_\beta$.

There are two evaluation orders which are widely used: the call-by-value evaluation strategy $\triangleright_{\mathbf{cbv}}$ and the call-by-name evaluation strategy $\triangleright_{\mathbf{cbn}}$. We will now introduce them in turn for the language defined in definition 2.1.1.

2.8.1. Call-by-Value

The call-by-value evaluation strategy is at the core of *strict* programming languages such as OCaml. The characteristic property of the call-by-value evaluation strategy is that arguments to a function are evaluated to their weak normal form **WNF** (Definition 2.6.2) before they are substituted for a variable in the function body. Similarly, the arguments to constructors are fully evaluated before we pattern match on the constructor. These two restrictions are modelled in the following definition, which specializes the general reduction relation of definition 2.4.1.

Definition 2.8.3 (Call-by-value Reduction Step)**.** A single step of call-by-value reduction is specified by the following two rules, where values v refer to the weak normal form **WNF** of section 2.6.

$$\begin{aligned} K(v_1, \ldots, v_n).\mathbf{case}\ \{K(x_1, \ldots, x_n) \Rightarrow e, \ldots\} &\triangleright^1_{\mathbf{cbv}} e[v_1, \ldots, v_n/x_1, \ldots, x_n] \\ \mathbf{cocase}\ \{d(x_1, \ldots, x_n) \Rightarrow e, \ldots\}.d(v_1, \ldots, v_n) &\triangleright^1_{\mathbf{cbv}} e[v_1, \ldots, v_n/x_1, \ldots, x_n] \end{aligned}$$

Restricting the rules of definition 2.4.1 so that only values are substituted for variables is not enough to make call-by-value reduction deterministic. We also have to restrict *where* we can apply these rewrites, and we can do this by restricting the contexts from definition 2.1.8.

Definition 2.8.4 (Call-by-value Evaluation Context)**.** The call-by-value evaluation contexts are defined by the following grammar, where values v refer to the weak normal form **WNF** of section 2.6.

$$E ::= \square \mid K(\overline{v}, E, \overline{e}) \mid E.\mathbf{case}\ \{\ldots\} \mid E.d(\overline{e}) \mid v.d(\overline{v}, E, \overline{e})$$

There are some freedoms in how call-by-value evaluation contexts are defined. In this case, for example, we choose to evaluate the arguments to constructors and destructors from left to right, and in the application of a destructor to a scrutinee we choose to evaluate the scrutinee first. Call-by-value reduction is then defined as the reflexive transitive closure of the reduction steps from definition 2.8.3 in evaluation contexts from definition 2.8.4.

2.8.2. Call-by-Name

The call-by-name evaluation order[6] is the foundation for *non-strict* programming languages such as Haskell. These languages are called non-strict because applying a function to a diverging computation does not necessarily lead to a diverging computation. Under the call-by-name evaluation order we only perform the computation steps that are necessary in order to make progress towards a value in weak head normal form. For example, when we have to evaluate a term of the form $e.d(e_1, \ldots, e_n)$ we only evaluate the term e to its weak head normal form **WHNF** (cp. definition 2.6.4) and don't evaluate the arguments of the destructor d. Similarly, when we have to evaluate a term of the form $e.\mathbf{case}\ \{\ldots\}$ we evaluate the destructee e to its weak head normal form, which for a well-typed expression is a constructor K applied to arbitrary expressions. These two restrictions motivate the following definition of evaluation contexts for call-by-name evaluation order:

Definition 2.8.5 (Call-by-name Evaluation Context)**.** The call-by-name evaluation contexts are defined by the following grammar:

$$E ::= \Box \mid E.d(\overline{e}) \mid E.\mathbf{case}\ \{\ldots\}$$

This definition of of call-by-name evaluation contexts is already sufficient to specify call-by-name evaluation. We don't have to also restrict the beta reduction rules of definition 2.4.1, like we had to do in definition 2.8.3 for the call-by-value evaluation order. Reduction strategies which also reduce to head normal forms **HNF** or normal forms **NF** can be found in the article by Sestoft (2001), but restricted to just the untyped lambda calculus.

[6] As well as the call-by-need evaluation order (cp. Zena M. Ariola, Maraist, Odersky, Felleisen, and Wadler (1995),Launchbury (1993)), which is an optimization of call-by-name reduction which ensures that arguments are evaluated at most once.

3. Defunctionalization, Refunctionalization and Local Closures

The content of this chapter is based on the following peer-reviewed publication. The notation and terminology has been changed to be consistent with the rest of this thesis.

> David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann (2019). “Decomposition Diversity with Symmetric Data and Codata”. In: *Proc. ACM Program. Lang.* 4.POPL. DOI: `10.1145/3371098`. URL: `https://doi.org/10.1145/3371098`

This chapter further expands on a problem that I have already introduced in section 1.1.2 of the introduction: the expression problem. In this chapter I now introduce the *defunctionalization* and *refunctionalization* algorithms that are a partial solution to the expression problem. These two algorithms allow to transform data types into codata types and vice versa, but they only work as a whole-program transformation. In this chapter I do not yet discuss how these algorithms interact with evaluation order, but I will come back to this problem in chapter 4. This chapter also introduces typing rules for the untyped calculus that I have introduced in chapter 2. The type system only supports simple types, but I discuss the extension to more expressive type systems such as dependent types at the end. The extension of defunctionalization and refunctionalization to dependent types poses significant challenges. These challenges, and how we solved them, are described in a recent separate publication (Binder, Skupin, Süberkrüb, and Ostermann, 2024a), but they are not discussed in this thesis.

The central contribution of the paper on which this chapter is based is that we provided the first symmetric programming language with support for local (co)pattern matching, which includes local anonymous function or object definitions, that allows an automatic translation as described above. The paper also presented the first mechanical formalization of such a language and proved i) that the type system is sound, that the translations between data and codata types are ii) type-preserving, iii) behavior-preserving and iv) inverses of each other.

3.1. The Expression Problem

The expression problem, which I started to describe in section 1.1.2, describes a fundamental trade-off in program design: Should a program’s primary decomposition be determined by the way its domain objects are constructed (“functional” decomposition), or by the way they are destructed (“object-oriented” decomposition)? In the introduction I already argued that a programming languages should not force one of these decompositions on the programmer; rather, a programming language should support both ways of decomposing a program.

In this chapter I show that a programmer's decision between a functional and object-oriented decomposition does not have to be final; there is an automatic translation between these two decompositions. This bijective translation turns a data type into a codata type ("refunctionalization") or vice versa ("defunctionalization").

Programmers are confronted with the expression problem in many situations. Should you `fold` over the input or `unfold` over the output? Should a program be structured according to how its input is constructed or how its output is destructed? Should one use algebraic data types with pattern matching as available in functional languages or classes and methods as available in object-oriented languages? Should a program be extensible in its set of constructors or in its set of destructors?

In section 1.1.3 we have already seen that infinite streams can be represented as both data and codata types. The standard *map* function on infinite streams can therefore be written by pattern-matching on the constructors of the input (assuming that streams are defined as a data type):

```
map(f,Cons(x,xs)) = Cons(f(x),map(f,xs))
```

or it can be written by copattern-matching (Abel, Pientka, Thibodeau, and Setzer, 2013) on the destructors of the output (assuming that streams are defined as a codata type):

```
map(f,s).head = f(s.head)
map(f,s).tail = map(f,s.tail)
```

In the constructor-centric first version, it is easy to add new consumers (pattern-matching functions) but hard to add new producers (constructors); in the destructor-centric version it is the other way around. This trade-off has been discussed in many forms (John C. Reynolds, 1975; Cook, 1990; Krishnamurthi, Felleisen, and Friedman, 1998) and is today widely known as the *expression problem* (Wadler, 1998), which has resulted in a long string of works on programming techniques and programming language design to support extensibility of both constructors and destructors.

We want to analyze the problem on a more fundamental level. We want to understand the exact relation between the two different decompositions described by the expression problem. A promising first step in that direction is an analysis of two traditional global program transformations, defunctionalization (John Charles Reynolds, 1972; Danvy and Nielsen, 2001) and refunctionalization (Danvy and Millikin, 2009). These transformations change the modularity and extensibility of a program. For instance, defunctionalization collects all function definitions in a program and arranges them into a single pattern match. Defunctionalization also changes a destructor (function application) of a codata type (functions) — into constructors of a data type. We now show how to generalize both defunctionalization and refunctionalization. Furthermore, we use the term *transposition* and the verb *transpose* to refer to either defunctionalization or refunctionalization.

We also present a programming language that is *symmetric* in its support for these decompositions in the sense that any program in constructor-centric form can be mechanically transformed into a unique destructor-centric program and vice versa, and that these transformations are total, type-preserving, behavior-preserving, and inverses of each other. We consider the features of our language, particularly the transformations, to be generally useful in practice, but we are focusing on their use to enable us to explore the relation between the different decompositions.

We believe that a programming language that is symmetric in this sense is relevant for programmers, language designers, and language implementers:

- It is relevant for *programmers* because non-symmetric languages encourage or force one of the decompositions. For instance, in Haskell 98 (without GADTs), constructors of an algebraic data type `data Exp a` always return an expression of type `Exp a`, but functions (=destructors) can have types like `Int -> Exp Int` for a type constructor `Exp`, which has led programmers to use (typed) Church encodings of data types instead of algebraic data types to profit from the more powerful type system on the destructor (in this case: function) side (Carette, Kiselyov, and Shan, 2007). Object-oriented languages strongly encourage a destructor-based decomposition into objects; a constructor-based decomposition requires awkward designs such as the "visitor pattern" (Gamma, Helm, Johnson, and Vlissides, 1995). For constructor-based designs implemented with pattern matching, various features such as linear pattern matching or guards have been developed with no obvious counterpart in destructor-based designs, and vice versa. Even in languages with direct support for codata (Abel, Pientka, Thibodeau, and Setzer, 2013) (to be discussed in detail later in the related work section), symmetry is destroyed by intermingling codata types and function types. These asymmetries needlessly restrict the design choices of the programmer for purely technical reasons that have nothing to do with the problem domain.

- It is relevant for *language designers* because the symmetry can be used to identify language design "holes" (features that are available for one decomposition but not the other). There is also a conceptual "two for the price of one" economy: By identifying a feature for the constructor side to be the exact counterpart to a feature on the destructor side, the design becomes simpler and meta-theoretic properties for one side may by construction carry over to the other side. We also believe that this work can clarify the relation between object-oriented languages and functional data type-oriented languages (Cook, 2009).

- It is relevant for *language implementers* due to the possibility of using the transformations between the decompositions as a compilation technique and hence realizing two corresponding features with just one shared implementation. The transformation itself may also be a guide on how to implement cross-compilers between functional and object-oriented languages systematically.

Concretely, we make the following contributions:

- We present the first full symmetric programming language that allows invertible defunctionalization and refunctionalization.

- We have fully formalized the language in the Coq theorem prover and mechanically verified that the language is type-sound. We have implemented the transposition algorithms in Coq and have proven that they are total, preserve typing and behavior, and are inverses of each other. All “difficult” parts of the proofs have been mechanically verified in Coq, with a few rather obvious but very laborious to mechanize ‘plumbing’ proofs left as ordinary paper proofs.

Related work is discussed in detail in section 3.7, but we want to discuss two related previous lines of work here. Rendel, Trieflinger, and Ostermann (2015) presented an extension of defunctionalization and refunctionalization to arbitrary codata types (not just functions). In Rendel, Trieflinger, and Ostermann’s work, the transformations only work on a language that allows only top-level definitions such that programs can be arranged in a kind of matrix, which can then be transposed to flip the decomposition. Our transformation extends Rendel, Trieflinger, and Ostermann’s algorithm for a “complete” programming language that allows block structure/nesting/local definitions. Another line of work that bears a superficial similarity to this chapter is work on compiling codata to data and vice versa (Downen, Sullivan, Zena M. Ariola, and Jones, 2019; Laforgue and Régis-Gianas, 2017). We defer to the related work section for details, but for now, we want to emphasize that these works aim for a *compositional* encoding of codata in terms of data (or vice versa) that therefore does not change the modularity or extensibility of the program, whereas we are interested in *global* transformations that switch the modular structure in the sense of the expression problem.

The remainder of this chapter is structured as follows: In section 3.2, we present background on de- and refunctionalization and describe the language design issues that need to be addressed to turn these techniques into total transposition algorithms. In section 3.3 we present an informal overview of the solution to the problems discussed in section 3.2. Section 3.4 presents a case study to illustrate how the language works in terms of a useful and realistic example. Section 3.5 contains the formalization of the language on which we base our development. The presentation of the transposition algorithms is contained in section 3.6. Section 3.8 discusses implications and future work, and section 3.7 presents related work.

3.2. Problem Statement

We extend and generalize defunctionalization (John Charles Reynolds (1972) and Danvy and Nielsen (2001)) and refunctionalization (Danvy and Millikin (2009)), respectively. These traditional whole-program transformations turn programs with higher-order functions (that is, with function application as destructors of functions) into first-order programs with constructors of algebraic data types and pattern matching (defunctionalization) or back (refunctionalization) and as such are a useful step towards fully symmetric defunctionalization and refunctionalization between data and codata types. In this section, we revisit these transformations and describe the problems in turning them into transpositions of a full-fledged programming language.

To illustrate defunctionalization, consider this program in Haskell-like syntax which maps two anonymous functions over a list.

```
map :: (Int -> Int) -> [Int] -> [Int]
map f xs = ... f (head xs) ...

let x=7
    y=12
in map (\z.z+x) (map (\z.z*y) [1,2,3])
```

The traditional way to defunctionalize a function type is to first turn all local function declarations of that type into top-level definitions by lambda lifting (Johnsson, 1985). The values for the free variables in the function bodies are passed as parameters to these functions. Top-level declarations need a name, hence we need to synthesize/invent two new names in our example, `plus` and `mult`.

```
plus x = \z.z+x
mult y = \z.z*y

let x=7
    y=12
in map (plus x) (map (mult y) [1,2,3])
```

The next step is to create an algebraic data type with one constructor for each top-level function, whereby each constructor has a parameter for the free variable in the original function body. A special *apply* function is created, which pattern-matches on the synthesized algebraic data type to determine the correct function body for that call and to get access to the values that would have otherwise been stored in the closure. Function definitions are replaced by invocations of the matching synthesized constructor, and function applications are replaced by invocations of the first-order *apply* function.

```
data Int2Int = Plus Int | Mult Int

apply :: Int2Int -> Int -> Int
apply (Plus x) z = z+x
apply (Mult x) z = z*y

map :: Int2Int -> [Int] -> [Int]
map f xs = ... apply f (head xs) ...

let x=7
    y=12
in map (Plus x) (map (Mult y) [1,2,3])
```

Refunctionalization tries to turn first-order data types back into functions, i.e., reverse the process of defunctionalization, but the attempt to make it a total function and the inverse of defunctionalization fails for several reasons:

a) It is only a partial function in traditional functional languages because it is not clear what to do if there is more than one pattern match on the argument type of the function type to be refunctionalized (Danvy and Millikin, 2009). In our example, imagine a second function pattern matching on `Int2Int`, such as

   ```
   isPlus :: Int2Int -> Bool
   isPlus (Plus _) = True
   isPlus (Mult _) = False
   ```

   This extended program can no longer be refunctionalized.

b) Lambda-lifting requires the synthesis of new names not in the original program, which then show up as constructor names in the refunctionalized program, and it is not obvious how to make that process invertible. In our example, we had to invent the names `plus` and `mult`. If we would refunctionalize and then defunctionalize, it is not clear how to guarantee that we get the same names back.

c) It is not clear how to "undo" the lambda-lifting because the defunctionalized program contains no information about which functions ought to be de-lambda-lifted. In our example, it is not possible to reconstruct from the defunctionalized program whether `plus` and `mult` were originally defined locally or as top-level functions.

d) If the `apply` function is changed such that the top-level operation is no longer a pattern match on its first argument, it is not clear how to deal with the function body when refunctionalizing the program.

e) Finally, if the arguments of the newly generated constructors are changed to be not just variable names but general expressions, it is not clear how to preserve the evaluation order when refunctionalizing the program. For instance, if we change `(Plus x)` to `Plus (x + 1)` in the invocation of `map` above, a naive refunctionalization would refunctionalize the first argument of the first `map` call to `(\z.z+(x+1))`, thereby changing the order of evaluation by moving the addition inside a λ-abstraction.[1]

Previous work by Rendel, Trieflinger, and Ostermann (2015) has addressed problem a) by generalizing function types to general codata types with copattern matching (Abel, Pientka, Thibodeau, and Setzer, 2013). Function types are a special case of codata types with a single *apply* destructor. The case of multiple pattern matches on algebraic data types can be solved with codata types by synthesizing one destructor per pattern match; something that was not possible with traditional functions due to the inherent limitation of functions having only one destructor. However, Rendel, Trieflinger, and Ostermann (2015) left open a solution to problems b)-e): their proposal assumed a language in which all definitions and (co)pattern matches were on the level of top-level functions only, that is, no local definitions are possible. In the next section, we review how Rendel, Trieflinger, and Ostermann have addressed a) and describe our novel solutions for b)-e), which together enable fully invertible transposition for a complete functional language with local pattern matching and copattern matching (including λ-abstraction).

3.3. Overview

We now give an informal overview of how we addressed problems a) to e) as outlined in the previous section. We address

a) by generalizing functions to codata

b) by adding names to matches and comatches

[1]We assume a call-by-value language in the remainder of this chapter.

c) by distinguishing local and global constructor and destructor names

d) by adding definitions and codefinitions

e) by adding let-like functionality to matches and comatches

The next subsections describe these ideas in detail. Since all features we describe apply dually to both the data and codata features of the language, we use the prefix x to abstract over which side of the duality we refer to by replacing a concrete (possibly empty) prefix by x. For instance, an $x$tor is a *construc*tor or a *destruc*tor, $x$pattern matching is pattern matching or *co*pattern matching, and so forth.

We propose a language with symmetric data and codata types with (co)pattern matching, together with *first-order* functions.

3.3.1. Generalizing Functions to Codata

While most functional languages support data types and function types, we support data types and codata types, which are strictly more general than function types. This means that even though we do not mention function types, lambda abstraction, and the application of a lambda-defined function to an argument explicitly in the formalization in section 3.5, these could be provided by desugarings into the more fundamental codata type declarations, copattern matches and destructor applications, respectively.

Functions are the special case of codata with just one destructor (typically called `apply`). In our example, we can define such a codata type for functions from `Int` to `Int` as follows:

```
codata Int2Int { apply(Int): Int }
```

This works similarly to how function types are provided in Java, where lambda abstractions are objects which implement the `Function<T,R>` interface, which provides the `R apply(T t)` method.

Codata types are instantiated by copattern matching, and destructors can be called on values of codata types using dot notation. The following example shows how to apply the function `f` on its argument `head xs` by calling the destructor `apply` on `f`, and a copattern match that mimics the λ-abstraction `\z.z+x`.

```
map :: Int2Int -> [Int] -> [Int]
map f xs = ... f.apply(head xs) ...
...
in map (cocase Int2Int { apply(z) => z+x }) ...
```

If we consider the example of the additional `isPlus` function from the previous section, refunctionalization can now be restored by adding an additional destructor to the codata type and corresponding destructor implementations in the copattern matches for that codata type. The result of refunctionalizing `Int2Int` after adding the `isPlus` function to the defunctionalized program from the introduction looks as follows:

```
codata Int2Int { apply(Int): Int,  isPlus() : Bool }

let x=7, y=12 in
  map (cocase Int2Int { apply(z) => z + x, isPlus() => true })
```

```
        (map (cocase Int2Int { apply(z) => z * y, isPlus() => false})
             [1,2,3])
```

3.3.2. Adding Names to Cases and Cocases

We solve the problem of synthesizing names for new constructors/destructors by requiring programmers to give a unique name to each (co)pattern match in the program. For instance, here we give the name `Plus` to the first comatch. This name is then turned into the constructor name `Plus`. In order to visually separate the name of the comatch from the codata type on which we comatch, we use the keyword `on`.

```
... in map (cocase Plus on Int2Int { apply(z) => z+x }) ...
```

Strictly speaking, this step is not required when writing code, since these names are only required when transposing the program. Instead, it would suffice to generate them on the fly during the transformation process by either prompting the user or having a generator for fresh names. However, names specified in the program text have the distinct advantage that these names will be turned into constructor or destructor names, respectively; autogenerated names decrease the readability of the generated program.

3.3.3. Local and Global Constructor and Destructor Names

When refunctionalizing a data type, it is not obvious whether a constructor invocation should be turned into an inlined copattern match (which would lead to code duplication if the same constructor is invoked multiple times) or whether it should be turned into a function call of a function that does the copattern match (which would avoid code duplication if the same constructor is invoked multiple times). Dually, the same problem arises for defunctionalization and destructor invocations. To keep refunctionalization and defunctionalization total and inverses of each other, we distinguish *local* names (denoted by names that start with an underscore `_`) from *global* names (names that do not start with an underscore). Names of generator and consumer functions (introduced in the next subsection) will always be global, while names of matches and comatches will always be local.

Local constructors and destructors can only be invoked in one place in the program; transposing the corresponding data or codata type leads to an inlined match or comatch of the opposite polarity. Conversely, global constructors and destructors can be invoked in many places in the program; transposing the corresponding data or codata type yields a top-level first-order function definition containing the xmatch, which is called in all places that used to invoke the xtor. All xtors that result from local xpattern matches are local, thereby guaranteeing that a transposition roundtrip will again yield the same program.

In our running example, the `Plus` and `Mult` constructors are local constructors.

```
data Int2Int { _Plus(Int), _Mult(Int) }
```

Let us consider an extension of the data type with another constructor that is global (in this case for the identity function):

```
data Int2Int { _Plus(Int), _Mult(Int), Identity() }
...
apply (Identity()) z = z
```

If we refunctionalize `Int2Int`, then the invocations of `_Plus` and `_Mult` are turned into the copattern matches we started with. Since the `Identity` constructor is global, a top-level function

```
Identity = cocase Identity on Int2Int { apply(z) => z }
```

is generated and this function is invoked in all places that invoked the `Identity` constructor.

3.3.4. Definitions and Codefinitions

As we have seen in the previous subsection, global xtors are turned into top-level functions containing an xpattern match. However, how can we know which top-level functions must be turned back into a global xtor in the next round of transposition? And what do we do about top-level functions that do not contain a top-level xpattern match in their body?

We solve both problems by partitioning functions into three different kinds:

- Ordinary functions are not affected by transposition (except that their bodies are transposed).
- *Definitions* are syntactically restricted to contain a top-level pattern match on their first argument. Refunctionalization turns definitions into global destructors; the cases of the pattern match are distributed to the corresponding copattern matches.
- *Codefinitions* are syntactically restricted to contain a top-level copattern match. Defunctionalization turns codefinitions into global constructors; the cases of the copattern match are distributed to the corresponding pattern matches.

All three kinds of functions are not first-class, i.e. they cannot be passed as an argument or returned as a value. In our running example, `apply` is a global destructor of `Int2Int`, hence defunctionalization turns `apply` into a consumer function (denoted by the keyword `def`).

```
def Int2Int.apply(z : Int) : Int {
  Plus(x) => z+x,
  Mult(x) => z*x}
```

Transposing `Int2Int` once more turns the definition `apply` back into a global destructor, as intended.

In a similar fashion, the additional `Identity` constructor from subsection 3.3.3 would now be refunctionalized into a codefinition (denoted by the keyword `codef`).

```
codef Identity() : Int2Int { apply(z) => z }
```

Regarding ordinary functions without a top-level xpattern match, in our language, we keep them as a separate construct because we consider it to be notationally and conceptually convenient to distinguish them from definitions and codefinitions. However, for the sake of completeness we want to point out that they can be easily desugared using a `Unit` data type (with a single no-argument constructor, as usual): An ordinary function with body e becomes a definition of type `Unit` with e in the single pattern match case (a similar desugaring to codefinitions would also be possible).

3.3.5. Let-like Functionality for Matches and Comatches

Let us now reconsider the example from the introduction, changing `(Plus x)` to `Plus (x + 1)` in the invocation of `map`. In our CBV language, `x + 1` is evaluated when the constructor call is evaluated. However, if we were to refunctionalize the constructor invocation to `cocase plus on Int2Int {apply(z)=> z+(x+1)}`, then the `x + 1` would be evaluated only when the `apply` destructor is invoked.

We solve this problem by extending both the pattern match and the copattern match construct with a name binding construct similar to an enclosing `let` binding. These bindings are evaluated when the xpattern match itself is evaluated, which restores the desired evaluation order.

In our example, we add a corresponding binding to the comatch:

```
...cocase Plus on Int2Int using x:=x+1 { apply(z) => z+x }
```

Adding these annotations to xmatches instead of simply using `let`-bindings around them corresponds to the idea that xmatches are essentially a type of local `gfun` or `cfun` and thus should be thought of as *closures*. Furthermore, since they need to be closed in order to apply transpositions on them, this removes the need to search for all relevant `let`s around them which are required for this precondition to hold.

3.4. Case Study

The symmetric design of our language gives programmers the possibility to view the domain they are modeling from different angles, depending on the decompositions that are chosen for the domain objects occurring in the program. Being able to switch decompositions makes it more convenient to change or add functionality, and gives new insights into the structure of the program. To illustrate this point, we present a case study which is inspired by Danvy, Johannsen, and Zerny (2011), who inter-derive reduction-based and reduction-free negational normalization functions. The original case study used de- and refunctionalization at several places to change the perspective on the program, which was done manually. By contrast, we can leverage the symmetric design of our language to perform corresponding transposition mechanically. Thus, this case study focuses on parts of the original case study by Danvy, Johannsen, and Zerny (2011) which hinged on the use of de- and refunctionalization. The goal is to obtain a program which computes the negation normal form of a boolean formula with conjunction, disjunction and negation by repeatedly searching for a redex of the form $\neg(\phi \wedge \psi)$, $\neg(\phi \vee \psi)$ or $\neg\neg\phi$ and replacing it by $\neg\phi \vee \neg\psi$, $\neg\phi \wedge \neg\psi$ and ϕ, respectively.[2] We will semi-mechanically

[2] A simpler, "big-step" style implementation of this problem is of course possible and also described by Danvy; here we focus on the "small-step" reduction-style solution because it is well-suited to

derive this program iteratively after first manually writing a program that searches for one such redex. The program requires only small modifications after the mechanical defunctionalization to adapt this code into a reduction-based evaluator which evaluates a boolean formula to its negation normal form. This approach highlights how the development of the final solution was greatly simplified by switching our *view* on the program by changing to a different decomposition, which was aided by the use of a mechanical transformation.

The definitions of expressions, redexes, values, the embedding of values in expressions, and the reduction of immediate redexes will not change during the subsequent steps, and are given by the following definitions:

```
data Expr { EVar(Id), ENot(Expr), EAnd(Expr, Expr), EOr(Expr, Expr) }
data Redex { RedNot(Expr), RedAnd(Expr, Expr), RedOr(Expr, Expr) }
data Value { ValPosVar(Id),
             ValNegVar(Id),
             ValAnd(Value, Value),
             ValOr(Value, Value) }

def Value.asExpr() : Expr {
  ValPosVar(id) => EVar(id),
  ValNegVar(id) => ENot(EVar(id)),
  ValAnd(e1,e2) => EAnd(e1.asExpr(), e2.asExpr()),
  ValOr(e1,e2)  => EOr(e1.asExpr(), e2.asExpr()) }

def Redex.eval() : Expr {
  RedNot(e)     => e,
  RedAnd(e1,e2) => EOr(ENot(e1), ENot(e2)),
  RedOr(e1,e2)  => EAnd(ENot(e1), ENot(e2)) }
```

As a first step, we write the functions `search`,[3] `searchPos` and `searchNeg`, which search for the leftmost outermost redex in an expression. The function `search` starts the search by calling `searchPos` with the initial continuation; `searchPos` searches for the first negation, recursively building up a continuation along the way and passes the computation to `searchNeg` after the first negation has been encountered. The `Found` data type represents the result of searching for a redex in an expression.

```
data Found { FoundValue(Value), FoundRedex(Redex) }

codata Value2Found { apply(Value) : Found }

/* Start the search with the trivial continuation */
fun search(e : Expr) : Found :=
  e.searchPos(cocase BaseCnt on Value2Found {
    apply(val) => FoundValue(val)})

/* Searching for a negation */
def Expr.searchPos(cnt : Value2Found) : Found {
  EVar(id)     => cnt.apply(ValPosVar(id)),
  ENot(e)      => e.searchNeg(cnt),
  EAnd(e1,e2) => e1.searchPos(
    cocase AndCnt1 on Value2Found using e2:=e2, cnt:=cnt {
      apply(v1) => e2.searchPos(
```

illustrate the features of our language.

[3]The `search` function is actually the result of CPS-transforming and then simplifying a direct-style function; the simplification amounts to only applying a continuation when a value is found.

```
        cocase AndCnt2 on Value2Found using v1:=v1, cnt:=cnt {
          apply(v2) => cnt.apply(ValAnd(v1, v2))})}),
  EOr(e1,e2)  => e1.searchPos(
    cocase OrCnt1 on Value2Found using e2:=e2, cnt:=cnt {
      apply(v1) => e2.searchPos(
        cocase OrCnt2 on Value2Found using v1:=v1, cnt:=cnt {
          apply(v2) => cnt.apply(ValOr(v1, v2))})})}

/* Searching a redex under a negation */
def Expr.searchNeg(cnt : Value2Found) : Found {
  EVar(id)     => cnt.apply(ValNegVar(id)),
  ENot(e)      => FoundRedex(RedNot(e)),
  EAnd(e1,e2) => FoundRedex(RedAnd(e1,e2)),
  EOr(e1,e2)  => FoundRedex(RedOr(e1,e2)) }
```

Following the approach of Danvy, Johannsen, and Zerny (2011), we defunctionalize the codata type `Value2Found`, since it is known that applying defunctionalization results in interesting semantic artifacts, bringing us closer to an abstract machine representation. This results in the following transformed program:

```
data Value2Found { _BaseCnt(),
                   _AndCnt1(Expr, Value2Found),
                   _AndCnt2(Value, Value2Found),
                   _OrCnt1(Expr, Value2Found),
                   _OrCnt2(Value, Value2Found)}

def Value2Found.apply(v : Value) : Found {
  _BaseCnt()       => FoundValue(v),
  _AndCnt1(e,cnt)  => e.searchPos(_AndCnt2(v,cnt)),
  _AndCnt2(v',cnt) => cnt.apply(ValAnd(v',v)),
  _AndCnt2(v',cnt) => cnt.apply(ValAnd(v',v)),
  _OrCnt1(e,cnt)   => e.searchPos(_OrCnt2(v,cnt)),
  _OrCnt2(v', cnt) => cnt.apply(ValOr(v',v))}

fun search(e : Expr) : Found := e.searchPos(_BaseCnt())

def Expr.searchPos(cnt : Value2Found) : Found {
  EVar(id)     => cnt.apply(ValPosVar(id)),
  ENot(e)      => e.searchNeg(cnt),
  EAnd(e1,e2) => e1.searchPos(_AndCnt1(e2,cnt)),
  EOr(e1,e2)  => e1.searchPos(_OrCnt2(e2,cnt))}

def Expr.searchNeg(cnt : Value2Found) : Found {
  EVar(id)     => cnt.apply(ValNegVar(id)),
  ENot(e)      => FoundRedex(RedNot(e)),
  EAnd(e1,e2) => FoundRedex(RedAnd(e1,e2)),
  EOr(e1,e2)  => FoundRedex(RedOr(e1,e2))}
```

Under this decomposition, we realize that `Context` is an appropriate name for the new data type. For example, the term `_OrCnt1(e,_AndCnt2(v,_BaseCnt()))` corresponds to the evaluation context $v \wedge (\Box \vee e)$, where v already is a value but e might contain further redexes (note that `Contexts` compose from the inside outwards, similarly to a stack). We therefore rename the data type `Value2Found` to `Context` and its constructors, which results in the following program: The changed part of the code after renaming and `apply` to `findNext` and modifying `FoundRedex`. The `apply` function takes a context, and returns the next redex if a value is plugged into the hole, we therefore rename it to `findNext`. We also extend the definition of the constructor `FoundRedex` to also return the enclosing context of the redex, and modify the `searchNeg` function accordingly.

```
data Context { _EmptyCtx(),
               _AndCtx1(Expr, Context),
               _AndCtx2(Value, Context),
               _OrCtx1(Expr, Context),
               _OrCtx2(Value, Context)}
data Found { FoundValue(Value), FoundRedex(Redex, Context) }

def Expr.searchNeg(ctx : Context) : Found {
  EVar(id)     => ctx.findNext(ValNegVar(id)),
  ENot(e)      => FoundRedex(RedNot(e), ctx),
  EAnd(e1,e2)  => FoundRedex(RedAnd(e1,e2), ctx),
  EOr(e1,e2)   => FoundRedex(RedOr(e1,e2), ctx)}
```

To evaluate an expression to normal form we need one additional function which substitutes an expression (in our case, the result of reducing a redex) into an evaluation context. Now it is easy to define the evaluation function:

```
def Context.substitute(e : Expr) : Expr {
  _EmptyCtx() => e,
  _AndCtx1(e', ctx) => ctx.substitute(EAnd(e,e')),
  _AndCtx2(v, ctx)  => ctx.substitute(EAnd(v.asExpr(), e)),
  _OrCtx1(e', ctx)  => ctx.substitute(EOr(e, e')),
  _OrCtx2(v, ctx)   => ctx.substitute(EOr(v.asExpr(), e))}

fun evaluate(e : Expr) : Value :=
  (search(e)).case _ {
    FoundValue(v) => v,
    FoundRedex(r,ctx) => evaluate(ctx.substitute(r.eval()))}
```

Bringing the data type `Context` back into destructor form results in the following program, which we might not have originally written, since it corresponds to the addition of a destructor to an existing codata type. Adding an additional definition to the defunctionalized version was easy.

```
codata Context { findNext(Value) : Found, substitute(Expr) : Expr }

def Expr.searchPos(ctx : Context) : Found {
  EVar(id)     => ctx.findNext(ValPosVar(id)),
  ENot(e)      => e.searchNeg(ctx),
  EAnd(e1,e2)  => e1.searchPos(
    cocase AndCtx1 on Context using e2:=e2, ctx:=ctx {
      findNext(v1) => e2.searchPos(
        cocase AndCtx2 on Context using v1:=v1, ctx:=ctx {
          findNext(v2) => ctx.findNext(ValAnd(v1, v2)),
          substitute(ex) => ctx.substitute(EAnd(v1.asExpr(), ex))}),
      substitute(ex) => ctx.substitute(EAnd(ex, e2))}),
  EOr(e1,e2)   => e1.searchPos(
    cocase OrCtx1 on Context using e2:=e2, ctx:=ctx {
      findNext(v1) => e2.searchPos(
        cocase OrCtx2 on Context using v1:=v1, ctx:=ctx {
          findNext(v2) => ctx.findNext(ValOr(v1, v2)),
          substitute(ex) => ctx.substitute(EOr(v1.asExpr(), ex))}),
      substitute(ex) => ctx.substitute(EOr(ex, e2))})}
```

3.5. Formalization

In this section we present the syntax, typing rules and operational semantics of the language on which defunctionalization and refunctionalization operate.

3.5.1. Syntax

The syntax of expressions is given in definition 3.5.1. We use the convention that type names, constructor names, and generator function names start with uppercase letters, whereas function names, destructor names, and consumer function names start with lowercase letters. Names in C and d may be prepended with an underscore to denote local names as described in section 3.3.3.

Definition 3.5.1 (Syntax of expressions)**.**

$$\mathcal{T} \in \textsc{TyName} \quad K \in \textsc{CtorName} \quad d \in \textsc{DtorName}$$

$$\begin{array}{rcll}
e & ::= & x & \textit{Variable} \\
 & \mid & K(\overline{e}) & \textit{Constructor and gfun calls} \\
 & \mid & e.d(\overline{e}) & \textit{Destructor and cfun calls} \\
 & \mid & e.\textbf{case}\ d\ \textbf{using}\ \overline{x = e}\ \{\overline{K(\overline{x}) \Rightarrow e}\} & \textit{Pattern match} \\
 & \mid & \textbf{cocase}\ K\ \textbf{on}\ \mathcal{T}\ \textbf{using}\ \overline{x = e}\ \{\overline{d(\overline{x}) \Rightarrow e}\} & \textit{Copattern match} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \textit{Let expression}
\end{array}$$

The syntax of programs Θ is given in definition 3.5.2. A program consists of a list of declarations δ, and each declaration can declare a data or a codata type, a toplevel definition or a toplevel codefinition.

Definition 3.5.2 (Syntax of programs)**.**

$$\begin{array}{rcll}
\delta & ::= & \textbf{data}\ \mathcal{T}\ \{\overline{K(\overline{\mathcal{T}})}\} & \textit{Data type} \\
 & \mid & \textbf{def}\ \mathcal{T}.d(\overline{\mathcal{T}}) : \mathcal{T}\ \{\overline{K(\overline{x}) \Rightarrow e}\} & \textit{Definition} \\
 & \mid & \textbf{codata}\ \mathcal{T}\ \{\overline{d(\overline{\mathcal{T}}) : \mathcal{T}}\} & \textit{Codata type} \\
 & \mid & \textbf{codef}\ K(\overline{\mathcal{T}}) : \mathcal{T}\{\overline{d(\overline{x}) \Rightarrow e}\} & \textit{Codefinition} \\
\Theta & ::= & \emptyset \mid \delta, \Theta & \textit{Program}
\end{array}$$

To avoid cluttering the definitions, we assume the program to be a global constant. In the remaining definitions, we query the global program via the sets defined in definition 3.5.3.

Definition 3.5.3 (Program queries)**.** Global sets to query a program Θ. Checking the types of expressions depends only on the starred sets, that is, only on declarations.

*		$\textsc{Dt}$	Data types defined in Θ
*		$\textsc{CoDt}$	Codata types defined in Θ
*	$\forall\ \mathcal{T} \in \textsc{Dt}$:	$\textsc{Ctor}(\mathcal{T})$	Constructors of type $\mathcal{T}$
*	$\forall\ \mathcal{T} \in \textsc{Dt}$:	$\textsc{Def}(\mathcal{T})$	Definitions for type $\mathcal{T}$
*	$\forall\ \mathcal{T} \in \textsc{CoDt}$:	$\textsc{Dtor}(\mathcal{T})$	Destructors of type $\mathcal{T}$
*	$\forall\ \mathcal{T} \in \textsc{CoDt}$:	$\textsc{Codef}(\mathcal{T})$	Codefinitions for type $\mathcal{T}$
	$\forall\ (\_.d(\_) : \_) \in \textsc{Def}(-)$:	$\textsc{Cases}(d)$	Body of definition d
	$\forall\ (K(\_) : \_) \in \textsc{Codef}(-)$:	$\textsc{Cocases}(K)$	Body of codefinition K

The sets are mostly self-explanatory, hence we refer to the Coq code for a formal definition and instead suggest to consider the example in examples 3.6.1 and 3.6.2, whose representation in terms of these set functions can be seen in examples 3.6.3 and 3.6.4, as illustration. The only noteworthy aspect of these sets is that $\textsc{Codef}(T)$ and $\textsc{Ctor}(T)$ have been set up in such a way that they have the same codomain, such that we can form the set union $\textsc{Codef}(T) \cup \textsc{Ctor}(T)$. The same holds for $\textsc{Def}(T)$ and $\textsc{Dtor}(T)$.

Together, the first seven items of the bottom half of definition 3.5.3 contain all the static information of a program that is necessary to typecheck expressions.

3.5.2. Typing Rules

Typechecking of expressions is defined in definition 3.5.4. Expressions are typechecked in the context of all function signatures, meaning that arbitrary recursion, including non-termination, between functions is possible.[4]

Definition 3.5.4 (Expression typing)**.** The typing of expressions $\Gamma \vdash e : T$ is defined by the following rules:

$$\frac{\mathbf{lookup}(x, \Gamma) = T}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : T_2}\ \textsc{T-Let}$$

$$\begin{array}{c} K(\overline{T'}) \in \textsc{Ctor}(T) \cup \textsc{Codef}(T) \\ \dfrac{\Gamma \vdash \overline{e : T'}}{\Gamma \vdash K(\overline{e}) : T}\ \textsc{T-Ctor} \end{array} \qquad \begin{array}{c} d(\overline{T'}) : T'' \in \textsc{Dtor}(T) \cup \textsc{Def}(T) \\ \dfrac{\Gamma \vdash e : T \quad \Gamma \vdash \overline{e' : T'}}{\Gamma \vdash e.d(\overline{e'}) : T''}\ \textsc{T-Dtor} \end{array}$$

$$\frac{\begin{array}{c} \Gamma \vdash \overline{e : T'''} \qquad \Gamma \vdash e' : T \qquad T \in \textsc{Dt} \\ \forall\ (K(\overline{T'}) \in \textsc{Ctor}(T)).\ \exists i.\ C = C_i\ \wedge \overline{T'''}, \overline{T'} \vdash e''_i : T'' \end{array}}{\Gamma \vdash e'.\mathbf{case}\ d\ \mathbf{using}\ \overline{e}\ \{\overline{K \Rightarrow e''}\} : T''}\ \textsc{T-Case}$$

$$\frac{\begin{array}{c} \Gamma \vdash \overline{e : T} \qquad T' \in \textsc{CoDt} \\ \forall\ (d(\overline{T''}) : T' \in \textsc{Dtor}(T)).\ \exists i.\ d = d_i\ \wedge\ \overline{T}, \overline{T''} \vdash e_i : T' \end{array}}{\Gamma \vdash \mathbf{cocase}\ K\ \mathbf{on}\ T'\ \mathbf{using}\ \overline{e}\ \{\overline{d \Rightarrow e'}\} : T'}\ \textsc{T-Cocase}$$

[4]Inductive and coinductive types are often used in conjunction with termination/productivity checks (e.g. Atkey and McBride (2013)), but these checks are an orthogonal concern for our purposes.

While the rules for let bindings, xtors and the different kinds of function calls are pretty standard, the rules for xmatches are slightly more involved. Firstly, the rule for matches is set up to ensure that every constructor occurs in exactly one case of the match. While it would be possible to allow non-exhaustive pattern matches, we have restricted ourselves to exhaustive pattern matches for the sake of simplicity.[5] Secondly, it is important to note that the bodies inside the case clauses are typechecked using only the variables bound by the binding lists and the variables provided by the respective case, i.e. it ensures that matches are closed terms. These remarks apply equally to comatches.

Typechecking a full program involves typechecking of function bodies in the context of the types of the arguments, as usual. Typechecking consumer and generator functions is a straightforward extension of typechecking local pattern (copattern) matches. Program well-formedness furthermore involves entirely unsurprising checks that all type names that are used are actually defined, that generator and consumer functions have a branch for each xtor of the corresponding (co)data type (exhaustiveness), and that names in local xpattern matches are globally unique, as explained in section 3.3.2. We have omitted the formal definition of the rule Wf-Prog from the paper because it is not very interesting; the full definition is of course part of our Coq formalization.

3.5.3. Reduction Rules

The following definitions 3.5.5 and 3.5.6 give the small-step operational semantics formulated with evaluation contexts.

Definition 3.5.5 (Values and Evaluation Contexts)**.**

$$\begin{array}{rcl} v & ::= & K(\overline{v}) \mid \textbf{cocase}\ K\ \textbf{on}\ \mathcal{T}\ \textbf{using}\ \overline{v}\ \{\overline{d \Rightarrow e}\} \\ \mathrm{E} & ::= & \square \mid K(\overline{v}, \square, \overline{e}) \mid \square.d(\overline{e}) \mid v.d(\overline{v}, \square, \overline{e}) \mid \textbf{let}\ x = \square\ \textbf{in}\ e \\ & \mid & \square.\textbf{case}\ d\ \textbf{using}\ \overline{e}\ \{\overline{K \Rightarrow e}\} \\ & \mid & v.\textbf{case}\ d\ \textbf{using}\ (\overline{v}, \square, \overline{e})\ \{\overline{K \Rightarrow e}\} \\ & \mid & \textbf{cocase}\ K\ \textbf{on}\ \mathcal{T}\ \textbf{using}\ (\overline{v}, \square, \overline{e})\ \{\overline{d \Rightarrow e}\} \end{array}$$

Definition 3.5.6 (Small-step operational semantics)**.** The small-step operational semantics $e \to e'$ is defined by the following rules:

$$\frac{e_1 \to e_2}{E[e_1] \to E[e_2]}\ \textsc{E-Congr} \qquad \frac{}{\textbf{let}\ x = v\ \textbf{in}\ e \to e[v]}\ \textsc{E-Let}$$

$$\frac{K \Rightarrow e \in \textsc{Cases}(d)}{K(\overline{v}).d(\overline{v'}) \to e[\overline{v}][\overline{v'}]}\ \textsc{E-Def} \qquad \frac{d \Rightarrow e \in \textsc{Cocases}(K)}{K(\overline{v}).d(\overline{v'}) \to e[\overline{v}][\overline{v'}]}\ \textsc{E-Codef}$$

$$\frac{}{K(\overline{v}).\textbf{case}\ \ldots\ \textbf{using}\ \overline{v'}\ \{K \Rightarrow e, \ldots\} \to e[\overline{v}][\overline{v'}]}\ \textsc{E-Case}$$

[5] Refunctionalization, for example, would translate an exception arising from an incomplete pattern match into the invocation of a destructor on a copattern match with a missing cocase. Semantic preservation under transposition might therefore still be possible.

$$\frac{}{(\textbf{cocase } K \textbf{ on } T \textbf{ using } \overline{v} \; \{ \; d \Rightarrow e, \dots \; \}).d(\overline{v'}) \to e[\overline{v}][\overline{v'}]} \; \textsc{E-Cocase}$$

Our language uses call-by-value evaluation, so arguments to functions and terms bound in let expressions and binding lists are evaluated first. Substitution is particularly simple since it only needs to be defined for values, which are closed terms. Continuing the example from above, if e is a value, then we write the evaluation of the let expression as $\textbf{let } x = v \textbf{ in } (\textbf{let } y = e' \textbf{ in } f(1,0)) \to (\textbf{let } x = e' \textbf{ in } f(1,0))[v] = \textbf{let } y = e' \textbf{ in } f(v,0)$. If the body of the function f is the expression e, then we use the suggestive notation $f(\overline{v}) \to e[\overline{v}]$ to indicate the substitution of the arguments into the body of the function.

The expressions in the cocases of a comatch do not have to be evaluated to values for the comatch to be a value. This corresponds to not evaluating expressions under a lambda abstraction when the function type is generalized to arbitrary codata.

3.5.4. Type Soundness

We have proven the type soundness of our language in Coq by the usual preservation and progress theorems.

Theorem 3.5.7 (Preservation)**.** *For all expressions e, if $\Gamma \vdash e : T$ in some program Θ with Θ OK and $e \to e'$, then $\Gamma \vdash e' : T$.*

Theorem 3.5.8 (Progress)**.** *For all programs Θ with Θ OK and expressions e, if $\Gamma \vdash e : T$, then either $e \to e'$, or e is a value.*

Furthermore, we have implemented algorithmic functions implementing the typing and evaluation relations and proven their correctness and completeness with respect to the inductive relations given here. In particular, this shows that the typing and small-step reduction relations are decidable.

3.6. Defunctionalization and Refunctionalization

We implemented defunctionalization and refunctionalization as a two-stage process. In the first step we compute a new program skeleton which consists only of the type signatures. This program skeleton is computed from the given program and the (co)data type T chosen to be transposed. In the new program skeleton, the chosen data type becomes a codata type, or vice versa, with its constructor or destructor signatures collected from the original program, and there are certain changes to the function signatures. The reason to have this stage separate is that it allows us to formulate the statement that typechecking is preserved under transposition. In the second step, the new function bodies are computed from the old program. For this, we use defunctionalization and refunctionalization functions for expressions in a given program.

In subsection 3.6.1 we present the running example for the presentation of the algorithm. In subsections 3.6.2 and 3.6.3 we present the first and second stages of the algorithm, respectively.

3.6.1. A Running Example

We will use the example of examples 3.6.1 and 3.6.2 to illustrate the transposition algorithm. Its formal representation can be found in examples 3.6.3 and 3.6.4.

Example 3.6.1 (Light as a codata type)**.** The following program defines `Light` as a codata type:

```
data Color { Red(), Blue() }
codata Light { color() : Color, next()  : Light }
codef Const(c : Color) : Light {
  color() => c,
  next()  => Const(c) }
codef RedBlue() : Light {
  color() => Red(),
  next()  =>
    cocase _BlueRed on Light {
      color() => Blue(),
      next()  => RedBlue()}}
```

Defunctionalizing the codata type `Light` of the program example 3.6.1 yields the program example 3.6.2. Inversely, refunctionalizing the data type `Light` of example 3.6.2 yields example 3.6.1.

Example 3.6.2 (Light as a data type)**.** The following program defines `Light` as a codata type:

```
data Color { Red(), Blue() }
data Light { Const(Color), RedBlue(), _BlueRed() }
def Light.color() : Color {
  Const(c)      => c,
  RedBlue()     => Red(),
  _BlueRed()    => Blue() }
def Light.next() : Light {
  Const(c)      => Const(c),
  RedBlue()     => _BlueRed(),
  _BlueRed()    => RedBlue() }
```

Example 3.6.3 (Light as a codata type (formal))**.**

$$\begin{aligned}
\textsc{Dt} &= \{\texttt{Color}\}\\
\textsc{CoDt} &= \{\texttt{Light}\}\\
\textsc{Ctor}(\texttt{Color}) &= \{\texttt{Red}(), \texttt{Blue}()\}\\
\textsc{Dtor}(\texttt{Light}) &= \{\texttt{color}() : \texttt{Color}, \texttt{next}() : \texttt{Light}\}\\
\textsc{Def}(\texttt{Light}) &= \{\texttt{Const}(\texttt{Color}), \texttt{RedBlue}()\}\\
\textsc{Cocases}(\texttt{Const}) &= \{\texttt{color}() => \texttt{c}, \texttt{next}() => \texttt{Const(c)}\}\\
\textsc{Cocases}(\texttt{RedBlue}) &= \{\texttt{color}() => \texttt{Red(),}\\
&\qquad \texttt{next() => cocase \_BlueRed on Light } \{\\
&\qquad\qquad\qquad \texttt{color() => Blue(),}\\
&\qquad\qquad\qquad \texttt{next() => RedBlue()}\}
\end{aligned}$$

Example 3.6.4 (Light as a data type (formal))**.**

$$\begin{aligned}
\textsc{Dt}' &= \{\texttt{Color}, \texttt{Light}\}\\
\textsc{CoDt}' &= \emptyset\\
\textsc{Ctor}'(\texttt{Color}) &= \{\texttt{Red}(), \texttt{Blue}()\}\\
\textsc{Ctor}'(\texttt{Light}) &= \{\texttt{Const}(\texttt{Color}), \texttt{RedBlue}(), \texttt{\_BlueRed}()\}\\
\textsc{CoDef}'(\texttt{Light}) &= \{\texttt{color}() : \texttt{Color}, \texttt{next}() : \texttt{Light}\}\\
\textsc{Cases}'(\texttt{color}) &= \{\texttt{Const(c)=> c}, \texttt{RedBlue()=> Red()}, \texttt{\_BlueRed()=> Blue()}\}\\
\textsc{Cases}'(\texttt{next}) &= \{\texttt{Const(c)=> Const(c)}, \texttt{RedBlue()=> \_BlueRed()}, \texttt{\_BlueRed()=> RedBlue()}\}
\end{aligned}$$

3.6.2. Computing the New Program Signatures

The new program skeleton consists of data types, codata types, and the signatures of definitions and codefinitions, which we obtain from the original program by the changes described in definitions 3.6.5 and 3.6.6. The function $\textsc{localCases}(T)$ will return the signatures of all local matches for type T in a program, i.e. their names, each together with a list of the types in the bindings list and the return type. $\textsc{localCocases}$ does the same for local comatches.

This means that we remove `Light` from $\textsc{CoDt}$ and add it to $\textsc{Dt}$ and we set $\textsc{Dtor}(\texttt{Light}) = \textsc{Gfun}(\texttt{Light}) = \emptyset$. Furthermore, with $\textsc{localCocases}(\texttt{Light}) = \{\texttt{\_BlueRed}()\}$, we obtain

$$\begin{aligned}
\textsc{Dt}' =& \textsc{Dt} \cup \{\texttt{Light}\} = \{\texttt{Color}, \texttt{Light}\}\\
\textsc{CoDt}' =& \textsc{CoDt} \setminus \{\texttt{Light}\} = \emptyset\\
\textsc{Ctor}'(\texttt{Light}) =& \textsc{Gfun}(\texttt{Light}) \cup \textsc{localCocases}(\texttt{Light})\\
=& \{\texttt{Const}(\texttt{Color}), \texttt{RedBlue}()\} \cup \{\texttt{\_BlueRed}()\}\\
\textsc{Cfun}'(\texttt{Light}) =& \textsc{Dtor}(\texttt{Light}) = \{\texttt{color}() : \texttt{Color},\ \texttt{next}() : \texttt{Light}\}.
\end{aligned}$$

Observe that the two transformations are entirely symmetric in this first stage; i.e. one can exchange defunctionalization and refunctionalization, data type and codata type, consumer and generator, as well as match and comatch, and the description remains the same. In this and the following definitions, the isGlobal and isLocal predicates check whether a name is global or local, respectively.

Definition 3.6.5 (Defunctionalization of Program Signatures)**.** The algorithm for computing the new typing information, i.e. the new *skeleton*, when defunctionalizing with type T.

$$\begin{aligned}
\textsc{Dt}' &:= \textsc{Dt} \cup \{T\}\\
\textsc{CoDt}' &:= \textsc{CoDt} \setminus \{T\}\\
\textsc{Ctor}(S)' &:= \begin{cases} \textsc{Ctor}(S) & S \neq T\\ \textsc{localCocases}(T) \cup \textsc{Gfun}(T) & S = T \end{cases}\\
\textsc{Dtor}(S)' &:= \begin{cases} \textsc{Dtor}(S) & S \neq T\\ \emptyset & S = T \end{cases}\\
\textsc{Gfun}(S)' &:= \begin{cases} \textsc{Gfun}(S) & S \neq T\\ \emptyset & S = T \end{cases}\\
\textsc{Cfun}(S)' &:= \begin{cases} \textsc{Cfun}(S) & S \neq T\\ \left\{(d(\overline{T'}) : T'') \in \textsc{Dtor}(T) \;\middle|\; \mathsf{isGlobal}(d)\right\} & S = T \end{cases}
\end{aligned}$$

Definition 3.6.6 (Refunctionalization of Programs (Signatures))**.** The algorithm for computing the new typing information, i.e. the new *skeleton*, when refunctionalizing with type T.

$$\begin{aligned}
\textsc{Dt}' &:= \textsc{Dt} \setminus \{T\}\\
\textsc{CoDt}' &:= \textsc{CoDt} \cup \{T\}\\
\textsc{Ctor}(S)' &:= \begin{cases} \textsc{Ctor}(S) & S \neq T\\ \emptyset & S = T \end{cases}\\
\textsc{Dtor}(S)' &:= \begin{cases} \textsc{Dtor}(S) & S \neq T\\ \textsc{localCases}(T) \cup \textsc{Cfun}(T) & S = T \end{cases}\\
\textsc{Gfun}(S)' &:= \begin{cases} \textsc{Gfun}(S) & S \neq T\\ \left\{C(\overline{T'}) \in \textsc{Ctor}(T) \;\middle|\; \mathsf{isGlobal}(C)\right\} & S = T \end{cases}\\
\textsc{Cfun}(S)' &:= \begin{cases} \textsc{Cfun}(S) & S \neq T\\ \emptyset & S = T \end{cases}
\end{aligned}$$

3.6.3. Computing the New Program Bodies

We obtain the new function bodies from the original bodies by the transformations shown in definitions 3.6.9 and 3.6.10.

Definition 3.6.7 (Defunctionalization of Program Bodies)**.** The algorithm for computing the new function bodies when transposing with type T. Here $\textsc{Cfun}(T)$ (resp. $\textsc{Gfun}(T)$ is the set of consumer functions (resp. generator functions) of the *old* program, while $\textsc{Cfun}(T)'$ (resp. $\textsc{Gfun}(T)'$) is the set of consumer functions (generator functions) in the *new* program.

$$\textsc{Cocases}(f)' := \begin{cases} \mathcal{D}[\textsc{Cocases}(f)] & f \notin \textsc{Gfun}(T) \\ \emptyset & f \in \textsc{Gfun}(T) \end{cases}$$

$$\textsc{Cases}(f)' := \begin{cases} \mathcal{D}[\textsc{Cases}(f)] & f \notin \textsc{Cfun}(T)' \\ \{\mathcal{D}[\text{cocases}(d)] \mid (d(\overline{T_p}) : T_r) \in \textsc{Dtor}(T), \mathsf{isGlobal}(d)\} & f \in \textsc{Cfun}(T)' \end{cases}$$

Definition 3.6.8 (Refunctionalization of Program Bodies)**.** The algorithm for computing the new function bodies when transposing with type T. Here $\textsc{Cfun}(T)$ (resp. $\textsc{Gfun}(T)$) is the set of consumer functions (resp. generator functions) of the *old* program, while $\textsc{Cfun}(T)'$ (resp. $\textsc{Gfun}(T)'$) is the set of consumer functions (generator functions) in the *new* program.

$$\textsc{Cocases}(f)' := \begin{cases} \mathcal{R}[\textsc{Cocases}(f)] & f \notin \textsc{Gfun}(T)' \\ \{\mathcal{R}[\text{cases}(C)] \mid C(\overline{T_p}) \in \textsc{Ctor}(T), \mathsf{isGlobal}(C)\} & f \in \textsc{Gfun}(T)' \end{cases}$$

$$\textsc{Cases}(f)' := \begin{cases} \mathcal{R}[\textsc{Cases}(f)] & f \notin \textsc{Cfun}(T) \\ \emptyset & f \in \textsc{Cfun}(T) \end{cases}$$

We refer to the result of defunctionalization of a program Θ as $\mathcal{D}[\Theta]$, and to the refunctionalization result as $\mathcal{R}[\Theta]$. We also use $\mathcal{D}$ and $\mathcal{R}$ for the defunctionalization and refunctionalization of expressions, respectively, which we will define in the next paragraph. We denote the collection of cocases for destructor d from all over the original program (i.e. from all generator functions and all local comatches) as cocases(d), and similarly the collection of cases for constructor c as cases(c). It is this collection step that makes the transformation a whole-program transformation, as it requires searching through the entire program for the relevant (co)case bodies. In our running defunctionalization example, new consumer functions `color` and `next` are added. For instance, the cases of `color` are the following collected cocases(`color`):

```
color() => c
color() => red()
color() => blue()
```

which stem from the codefinitions `const`, `RedBlue` and the local comatch `_BlueRed`, respectively.

Defunctionalization and Refunctionalization of Expressions

Defunctionalization $\mathcal{D}$ of an expression in a given program and with respect to a type T is shown in definition 3.6.9.

Definition 3.6.9 (Defunctionalization of expressions)**.** Defunctionalization of expressions in a given program w.r.t. a codata type $\mathcal{T}$. The interesting cases are above the horizontal line and the congruence cases are below.

$$
\begin{array}{lll}
\mathcal{D}[e.d(\overline{e})] & := & \mathcal{D}[e].d(\overline{\mathcal{D}[e]}),\ \text{if } d \in \textsc{Dtor}(T) \text{ and } \mathsf{isGlobal}(d) \\
\mathcal{D}[e.d(\overline{e})] & := & \mathcal{D}[e].\mathbf{case}\ d\ \mathbf{using}\ \overline{\mathcal{D}[e]}\ \{\mathcal{D}[\text{cocases}(d)]\}, \\
 & & \quad \text{if } d \in \textsc{Dtor}(T) \text{ and } \mathsf{isLocal}(d) \\
\mathcal{D}[K(\overline{e})] & := & K(\overline{\mathcal{D}[e]}),\ \text{if } K \in \textsc{Codef}(\mathcal{T}) \\
\multicolumn{3}{l}{\mathcal{D}[\mathbf{cocase}\ K\ \mathbf{on}\ \mathcal{T}\ \mathbf{using}\ \overline{e:T}\ \{\ldots\}] := K(\overline{\mathcal{D}[e]})} \\
\hline
\mathcal{D}[x] & := & x \\
\mathcal{D}[K(\overline{e})] & := & K(\overline{\mathcal{D}[e]}) \\
\mathcal{D}[e.d(\overline{e})] & := & \mathcal{D}[e].d(\overline{\mathcal{D}[e]}),\ \text{if } d \notin \textsc{Dtor}(T) \\
\mathcal{D}[K(\overline{e})] & := & K(\overline{\mathcal{D}[e]}),\ \text{if } K \notin \textsc{Codef}(\mathcal{T}) \\
\mathcal{D}[e.d(\overline{e})] & := & \mathcal{D}[e].d(\overline{\mathcal{D}[e]}) \\
\multicolumn{3}{l}{\mathcal{D}\big[e.\mathbf{case}\ d\ \mathbf{using}\ \overline{e}\ \{\overline{C \Rightarrow e}\}\big] :=} \\
\multicolumn{3}{l}{\qquad \mathcal{D}[e].\mathbf{case}\ d\ \mathbf{using}\ \overline{\mathcal{D}[e]}\ \{\overline{C \Rightarrow \mathcal{D}[e]}\}} \\
\multicolumn{3}{l}{\mathcal{D}\big[\mathbf{cocase}\ C\ \mathbf{on}\ S\ \mathbf{using}\ \overline{e}\ \{\overline{d \Rightarrow e}\}\big] :=} \\
\multicolumn{3}{l}{\qquad \mathbf{cocase}\ C\ \mathbf{on}\ S\ \mathbf{using}\ \overline{\mathcal{D}[e]}\ \{\overline{d \Rightarrow \mathcal{D}[e]}\}, \text{for } S \neq T} \\
\mathcal{D}[\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2] & := & \mathbf{let}\ x = \mathcal{D}[e_1]\ \mathbf{in}\ \mathcal{D}[e_2]
\end{array}
$$

For defunctionalization, the interesting cases are some of those expressions that are related to the type T to be defunctionalized, specifically:

- Comatches generating T, which become local constructor calls.
- Generator function calls for codefinitions generating T, which become global constructor calls.
- Global destructor calls to destructors of T, which become consumer function calls.

But the most important case is that for *local* destructor calls $e.d(\overline{e})$ (for a destructor of T). Such a local destructor call is translated to a local match. The cases for that match are collected from the comatches and generator functions generating T, fetching all the cocases cocases(d) for the destructor d under consideration. This is the same algorithm as for the collection of the cases of the new consumer functions described above.

The cases for destructors $e.d(\overline{e})$ and generator function calls $C(\overline{e})$ of T look like congruence cases; we list them above the horizontal line because the meaning of those expressions changes: The destructor call $e.d(\ldots)$ is turned into a consumer function call $\mathcal{D}[e].d(\ldots)$, which happens to have the same syntax (because it allows a more economical presentation of the language). Similarly, the generator function call $C(\ldots)$ is turned into a constructor call $C(\ldots)$ that happens to have the same syntax.

The refunctionalization $\mathcal{R}$ of expressions, which is described in definition 3.6.10, works analogously.

Definition 3.6.10 (Refunctionalization of expressions)**.** Refunctionalization of expressions in a given program w.r.t. a data type $\mathcal{T}$. Interesting cases are above the horizontal line, congruence cases are below.

$$
\begin{array}{lll}
\mathcal{R}[C(\overline{e})] & := & C(\overline{\mathcal{R}[e]}),\ \text{if } C \in \textsc{Ctor}(T) \text{ and } \mathsf{isGlobal}(C) \\
\mathcal{R}[C(\overline{e})] & := & \textbf{cocase}\ C\ \textbf{on}\ T\ \textbf{using}\ \overline{\mathcal{R}[e]}\ \{\overline{d \Rightarrow X}\}, \\
 & & \quad \text{if } C \in \textsc{Ctor}(T) \text{ and } \mathsf{isLocal}(C) \\
\mathcal{R}[e.d(\overline{e})] & := & \mathcal{R}[e].d(\overline{\mathcal{R}[e]}),\ \text{if } d \in \textsc{Cfun}(T) \\
\multicolumn{3}{l}{\mathcal{R}[e.\textbf{case}\ d\ \textbf{using}\ \overline{e}\ \{\overline{C \Rightarrow e}\}] := \mathcal{R}[e].d(\overline{\mathcal{R}[e]}),\ \text{if } e : T} \\
\hline
\mathcal{R}[x] & := & x \\
\mathcal{R}[C(\overline{e})] & := & C(\overline{\mathcal{R}[e]}),\ \text{if } C \notin \textsc{Ctor}(T) \\
\mathcal{R}[e.d(\overline{e})] & := & \mathcal{R}[e].d(\overline{\mathcal{R}[e]}) \\
\mathcal{R}[C(\overline{e})] & := & C(\overline{\mathcal{R}[e]}) \\
\mathcal{R}[e.d(\overline{e})] & := & \mathcal{R}[e].d(\overline{\mathcal{R}[e]}),\ \text{if } d \notin \textsc{Cfun}(T) \\
\multicolumn{3}{l}{\mathcal{R}\big[e.\textbf{case}\ d\ \textbf{using}\ \overline{e}\ \{\overline{C \Rightarrow e}\}\big] :=} \\
\multicolumn{3}{l}{\qquad \mathcal{R}[e].\textbf{case}\ d\ \textbf{using}\ \overline{\mathcal{R}[e]}\ \{\overline{C \Rightarrow \mathcal{R}[e]}\}\ \text{if } e : S \text{ and } S \neq T} \\
\multicolumn{3}{l}{\mathcal{R}\big[\textbf{cocase}\ C\ \textbf{on}\ S\ \textbf{using}\ \overline{e}\ \{\overline{d \Rightarrow e}\}\big] :=} \\
\multicolumn{3}{l}{\qquad \textbf{cocase}\ C\ \textbf{on}\ S\ \textbf{using}\ \overline{\mathcal{R}[e]}\ \{\overline{d \Rightarrow \mathcal{R}[e]}\}} \\
\multicolumn{3}{l}{\mathcal{R}[\textbf{let}\ x = e_1\ \textbf{in}\ e_2] := \textbf{let}\ x = \mathcal{R}[e_1]\ \textbf{in}\ \mathcal{R}[e_2]}
\end{array}
$$

Matches are turned into local destructor calls, consumer function calls are turned into global destructor calls, global constructor calls are turned into generator function calls, and local constructor calls are turned into comatches with the cocases collected among the matches and consumer functions. There is one slight technical complication in the refunctionalization part: To check which case of the function definition applies to a `case` expression, we need to know the type of the expression on which we match. We use the notation $e : T$ to refer to that type, which would in an actual implementation be stored and remembered during typechecking. The alternative would have been to make $\mathcal{R}$ type-directed instead of syntax-directed, but we considered this alternative to be more readable.

Due to local destructor or local constructor calls where the algorithm needs to collect the (co)cases from all over the original program, defunctionalization and refunctionalization of expressions is a whole-program transformation itself. Especially, this means that it must take as inputs not only the expression and the type to be transposed but also the program with respect to which the transformation happens, i.e. the program that contains the term. To simplify the development of our Coq implementation, we have therefore split it into three parts: 1.) lifting of (co)matches to top-level functions, 2.) actual program transposition, where we produce top-level generator/consumer functions potentially marked as local (i.e., to be inlined), and 3.) inlining of these marked-as-local functions as (co)matches. This way we have an improved separation of concerns and can implement core defunctionalization and refunctionalization of expression functions that do not require the full program as input and are simple folds over the given expression, replacing local xtor calls by local function calls to be inlined later as (co)matches.

3.6.4. Correctness of Defunctionalization and Refunctionalization

For the following, we fix a type T which is used for the transposition. We furthermore assume that the inputs are suitable for the transformations, i.e. that T is a data type if we perform refunctionalization and a codata type if we perform defunctionalization.

Theorem 3.6.11 (Transposition preserves typing)**.** *If e is any expression in the original program Θ (with Θ OK) such that $\Gamma \vdash e : T$ holds (in Θ), then $\Gamma \vdash \mathcal{D}[e] : T$ holds in $\mathcal{D}[\Theta]$. Similarly, $\Gamma \vdash \mathcal{R}[e] : T$ holds in $\mathcal{R}[\Theta]$.*

Theorem 3.6.12 (Transposition is total)**.** *If a program Θ is well-formed (Θ OK), then transposition will result in a well-formed program Θ' (Θ' OK).*

Theorem 3.6.13 (Transposition preserves reduction relation)**.** *If expression e which is a part of a program Θ (with Θ OK) reduces to e' (in Θ), then $\mathcal{D}[e]$ reduces to $\mathcal{D}[e']$ in $\mathcal{D}[\Theta]$. Likewise $\mathcal{R}[e]$ reduces to $\mathcal{R}[e']$ in $\mathcal{R}[\Theta]$.*

Theorem 3.6.14 (Transpositions are mutual inverses)**.** *If Θ OK, then defunctionalization and refunctionalization are mutual inverses on all (co)data types defined in Θ (up to the ordering of signatures, function definitions, or (co)cases).*

Programs in our language can be naturally thought of as consisting of sets of these signatures, definitions, and co(cases), thus their order does not matter for typechecking or the reduction relation.[6] We have to point out that the mutual inverses property depends on our de Bruijn representation of variables; for a language with ordinary variable names, the property holds only modulo α-equivalence.

3.7. Related Work

We mainly focus on related work not already discussed in the previous sections.

Defunctionalization Defunctionalization as a technique to eliminate higher-order functions to make control flow (combined with CPS transformation) explicit goes back to John Charles Reynolds's classic essay (John Charles Reynolds (1972)). Danvy and colleagues have shown how it can be more widely applied Danvy and Nielsen, 2001, and in particular how it can be usefully combined with CPS transformations to derive semantic artifacts. They also introduced the *partial* inverse to defunctionalization, refunctionalization (Danvy and Millikin (2009)), and showed a similar relation to direct-style transformation. Our case study in section 3.4 is inspired by their showcase of all these transformations (Danvy, Johannsen, and Zerny (2011)).

[6] All (co)pattern matches are exhaustive and non-overlapping, therefore such reordering cannot affect any property relating to them.

Coinduction, Codata, and Copatterns Coinduction and coinductive types are directly supported in some languages such as Coq (Giménez (1996)). Coinductive types still define a data type in terms of its constructors; the main difference to inductive data types is that the semantics changes (greatest fixed point instead of least fixed points, guarded corecursion instead of structural recursion). The modularity and extensibility of the program is not affected by the use of coinductive types. Codata types were first introduced by Hagino (1989) as an extension of ML. Since then, objects and classes have been described coalgebraically (Jacobs (1995)) and codata has been described as the essence of object-oriented programming (Cook, 2009). Abel, Pientka, Thibodeau, and Setzer (2013) proposed a language with both data types/pattern matching and codata/-copattern matching, which has inspired an implementation in Agda. Abel, Pientka, Thibodeau, and Setzer's language is not symmetric in the sense we analyzed in this work because it mixes two forms of codata: codata defined in codata types, and first-class functions. These two forms of codata crucially depend on each other (a destructor with arguments is modeled as a no-argument destructor that resolves to a function that expects the argument). Due to the interplay between functions and codata, it is not obvious what refunctionalization and defunctionalization should mean in this language.

Expression Problem There is a long string of works on programming techniques and language designs that allow simultaneous extensibility in the constructor and destructor dimension (see related work section in Oliveira and Cook, 2012, for instance), usually by enabling a kind of micro-modularity, where the implementation for each constructor/destructor combination can be a separate module that can be freely composed with other such modules. The aim of these works is different from this one. Their goal is to enable extensibility and composability of both constructors and destructors. Our goal is to provide a symmetric language where the extensibility dimension can be switched with our transposition algorithms. There have also been some works that discuss the relation between the data/codata duality and the expression problem. Lämmel and Rypacek (2008) discusses a category-theoretic formulation of the duality between data and codata in terms of the expression problem; however, that work is about semantic methods and not programming language design. The most closely related work to ours is the one by Rendel, Trieflinger, and Ostermann (2015), whose relation to this work has been discussed in detail at the end of section 3.2.

Data/Codata Transformations Downen, Sullivan, Zena M. Ariola, and Jones (2019) compile a language with data types into one with codata types and vice versa, but their transformations are very different from ours. They map data to codata via the visitor pattern, i.e., the result has a codata type with one destructor per constructor in the original data type and an additional codata type for the visitor. To translate codata to data, they use what they refer to as *tabulation*: the resulting data type represents a table of potential answers to the destructor observations. Laforgue and Régis-Gianas (2017) propose a macro to support codata in OCaml, in which codata operations are reified as a data type and codata types are encoded as dispatch functions on reified codata operations. The essential difference between those works and ours is that the transformations of Downen, Sullivan, Zena M. Ariola, and Jones and Laforgue and Régis-Gianas are compositional and hence do not change the extensibility of the program. Their aim is a compositional encoding, not a change in the decomposition of the whole program.

Defunctionalization in Compilers Defunctionalization is used as a compiler technique to achieve different goals. In many cases, they remain opaque to the programmer since they do not add new functionality, but are rather used for low-level optimizations. Examples of these kinds of usage include Boquist and Johnsson (1996)'s work on optimizations for lazy functional languages. Sometimes, they might also be employed to provide additional functionality, mainly first-class functions, to a language. One such instance is explored by Grust, Schweinsberg, and Ulrich (2013). Contrary to this, our approach intends to make different decompositions of a program *accessible* to the programmer and thus provide such transformations as a means to manipulate programs. Moreover, in this work, the transpositions are mainly a means to an end, i.e. providing a multi-faceted view of a program to programmers.

3.8. Outlook

Program Matrices and PL Design The matrix formalism approach to de/refunctionalization was first explored by Rendel, Trieflinger, and Ostermann (2015) (based on the relation to the expression problem drawn there, and thus also in the tradition of earlier matrix representations (Cook, 1990)). Our work did not explicitly employ this kind of formalism, but our mental image of the transformations is strongly influenced by the matrix idea.[7] We think it is possible to formalize our work in terms of matrices, but a key difference to prior work is that not everything has global scope: The matrices would need to be enhanced with additional constraints that represent the possible locality restrictions of constructors or destructors. The matrix formalism is close to how we envision the symmetric programming environment. Further, as was hinted at in the introduction, we hope to exploit dualities for PL design, and such an explicit two-dimensional representation of programs can potentially help us better understand the design space of programming languages. We hope that this way one can avoid design mistakes, such as the above-mentioned non-orthogonal design found in previous attempts at combining the functional and object-oriented paradigms. To bring this idea to fruition it might also be instructive to try to combine it with the more general language approach and/or more powerful type systems as outlined in the next paragraphs.

A Deeper Symmetry Throughout this chapter, we emphasized the symmetry inherent in our language and the transformations defined on it. However, if "two-for-the-price-of-one economy" Wadler, 2003 is what one hopes to gain from such symmetries, one

[7]For instance, our Coq proof of the program transpositions being mutual inverses amounts to reducing this problem to matrix transpose being involutive.

could conceivably be far more economical. Observe how constructors take in a certain sense the place of destructors, and vice versa, when switching between the data and the codata sides. Now, compare the structure of constructor signatures $con(\overline{T_c^a}) : T_c$ and of destructor signatures $T_d.des(\overline{T_d^a}) : T_o$: On one side, we have the constructed type T_c, and on the opposite side, the destructed type T_d, which form a dual pair, and on both sides we have lists of arguments ($\overline{T_c^a}$ and $\overline{T_d^a}$, respectively). But the *output* type of destructors T_o lacks a counterpart on its opposite (the constructor) side. It is this lack of a better symmetry that at all forced us to give two accounts, one for defunctionalization and one for refunctionalization, or one for the data and one for the codata fragment, at least when explicitly formalizing them in Coq. Even if the sides only differ slightly, the missing symmetry and consequent structural difference is still glaringly obvious. We will solve this problem in chapter 4 by going from a language based on natural deduction to the symmetric framework of the sequent calculus.

To the Type Level and Beyond The data and codata languages, as well as the defunctionalization and refunctionalization algorithms defined on them, have already been extended by Ostermann and Jabs (2018) to cover parametric polymorphism. The extension to dependent types was an open problem (cp. also Huang and Yallop, 2023) that we have recently solved. The concrete challenges that had to be overcome, and how we solved them, are described in our article Binder, Skupin, Süberkrüb, and Ostermann (2024a).

Part II.

Sequent Calculi for Data and Codata

4. Defunctionalization, Refunctionalization and Evaluation Order

The content of this chapter is based on the following, thorougly revised and updated, article:

> David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann (2022). *Data-Codata Symmetry and its Interaction with Evaluation Order.* DOI: `10.48550/ARXIV.2211.13004`. URL: `https://arxiv.org/abs/2211.13004`

In that article and this chapter I solve a problem with the defunctionalization and refunctionalization algorithm that I introduced in chapter 3. The language and algorithm presented in chapter 3 presume a global call-by-value evaluation order. There is one major problem with this approach, and we already hinted at that problem in the introduction in section 1.1: in a language which uses the call-by-value evaluation strategy for all types, the eta-laws are only valid for data types but not for codata types. In this chapter we solve this problem in the following way: Every type that can be defined by the user is specified with both a polarity and an evaluation order. The two possible polarities are data types and codata types, and the two possible evaluation orders are call-by-value and call-by-name. We then present two different whole-program transformations: Defunctionalization and refunctionalization change the polarity of a type by using the matrix transposition algorithm presented in chapter 3. A separate algorithm allows to change the evaluation order of a type by inserting so-called *shift connectives* in the program. By composing these two transformations we obtain a defunctionalization algorithm which allows to change the polarity of a type, without losing the validity of any η-laws in the process. The content of this chapter can therefore be summarized by the following diagram:

$$\begin{tikzcd}
\textbf{cbv data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] & \textbf{cbv codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] \\
\textbf{cbn data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"] & \textbf{cbn codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"]
\end{tikzcd}$$

The defunctionalization and refunctionalization algorithms presented in this chapter also remedy the asymmetry of the corresponding algorithms presented in chapter 3. We argue that the failure of complete symmetry is due to the inherent asymmetry of natural deduction as the logical foundation of the language design. Natural deduction is asymmetric in that its focus is on *producers* (proofs) of types, whereas consumers (contexts, continuations, refutations) have a second-class status. The language presented in this chapter is fully symmetric in that polarity (data type vs codata types) and evaluation order (call-by-value vs call-by-name) are untangled and become independent attributes of a single form of type declaration.

4.1. Introduction

Let us discuss the two problems, the lack of symmetry and the interaction of defunctionalization and refunctionalization with evaluation order, that this chapter solves in more detail.

First problem: Lack of Symmetry

The first problem with existing language designs for data and codata types is a lack of symmetry. The aforementioned de/refunctionalization transformations can serve as a tool to evaluate symmetry: A failure to define these transformations as total, semantics-preserving and mutually inverse functions indicate an asymmetry. For example, Pottier and Gauthier (2006) figured out that generalized algebraic data types (GADTs) are necessary in order to correctly defunctionalize polymorphic functions. But Ostermann and Jabs (2018) showed that the requirement for total and inverse de- and refunctionalization makes it necessary to add the dual feature of generalized algebraic codata types (GAcoDTs).

However, even when the design is symmetric in that sense, there is another glaring asymmetry, already mentioned at the end of chapter 3, that has been the main motivation for this chapter: The definitions of data and codata types as well as the transformations between them look very similar, but not similar enough to have just one generic definition of a type (with data and codata being just two modes of using them) and just one transformation.

The underlying cause which we identified is the asymmetry between producers and consumers. Programming languages whose design is based on natural deduction rules only represent producers (expressions) of a type as a first class citizen. That is, there is only one form of typing judgement $\Gamma \vdash t : T$ for typing (producer) terms. This bias in the design of programming languages, which is reflected in their formalizations, manifests itself in various ways:

- A constructor such as “Zero” can be typed on its own, whereas a destructor such as “isZero” must be typed together with its application, e.g., “3.isZero”.
- Pattern matches have a separate “return type” in addition to the type of the argument they pattern match on, while copattern matches lack such a separate return type.
- Constructors of a data type and observations of a codata type have a different shape, observations have an additional “return type” of the observation.

Sequent calculus is a more symmetric alternative to natural deduction in which proofs/producers and refutations/consumers are defined in a completely symmetric way. Inspired by previous work on sequent-calculus-based languages (Zeilberger (2008b), Curien and Herbelin (2000), Downen and Zena M. Ariola (2014), Downen and Zena M. Ariola (2020), and Wadler (2003)) we have been able to achieve our:

> **First contribution:** We present a language in which data and codata types are *fully* symmetric and in fact just modes of one type definition construct. Defunctionalization and refunctionalization are *one* algorithm.

Second problem: Evaluation Order

The second problem concerns the interaction between data and codata types, de/refunctionalization, and evaluation order. In languages that support both data and codata types, it is desirable to have fine-grained control over evaluation order instead of prescribing a global fixed evaluation order:

- As we already described in section 1.1, programmers want to use laziness for easier problem decomposition and composability of algorithms, and strictness for easier reasoning about time and space complexity.
- Ad hoc solutions to a globally fixed evaluation order like the "seq" expression in Haskell weaken the valid reasoning principles available to programmers (Johann and Voigtländer (2004)).
- The validity of η-rules depends both on the evaluation order of the language and the polarity of the type.

We have already mentioned this last point in the introduction, but since it is central to our argument, we illustrate it again with two examples from the lambda calculus with pairs (adapted from Downen and Zena M. Ariola (2020)).

The η rule for the function type (and for codata types more generally) is only valid under call-by-name evaluation order. Consider, for example, the lambda term $(\lambda x.5)(\lambda x.\Omega x)$, which can be reduced to 5 under both the call-by-value and call-by-name evaluation order. If we replace $\lambda x.\Omega x$ by its η-reduct Ω, on the other hand, the resulting term diverges under call-by-value semantics, while it still reduces to 5 under call-by-name:

$$\begin{tikzcd}
5 & & & 5 \\
& (\lambda x.5)(\lambda x.\Omega x) \arrow[ul, "\textbf{cbn}"'] \arrow[dl, "\textbf{cbv}"] \arrow[r, equal, "\eta"] & (\lambda x.5)\Omega \arrow[ur, "\textbf{cbn}"] \arrow[dr, "\textbf{cbv}"'] & \\
5 & & & \textit{diverges}
\end{tikzcd}$$

The inverse situation holds for pairs (and data types more generally); their η-rules are only valid under call-by-value evaluation order. The following two terms are η-equal. But while the left term diverges under both call-by-value and call-by-name, the η-reduct on the right reduces to 5 under call-by-name.

$$\begin{tikzcd}
\textit{diverges} & & & 5 \\
& \textbf{case } \Omega \textbf{ of } \{ \langle x_1, x_2 \rangle \Rightarrow (\lambda x.5)\langle x_1, x_2 \rangle \} \arrow[lu, "\textbf{cbn}"'] \arrow[ld, "\textbf{cbv}"] \arrow[r, equal, "\eta"] & (\lambda x.5)\Omega \arrow[ru, "\textbf{cbn}"] \arrow[rd, "\textbf{cbv}"'] & \\
\textit{diverges} & & & \textit{diverges}
\end{tikzcd}$$

These problems suggest that evaluation order should be based on types (cp. Downen and Zena M. Ariola (2020)), together with so-called shifts (analogous to the shifts introduced by Zeilberger (2008b)) to switch between different evaluation orders. When relating data and codata types via de/refunctionalization, however, these different evaluation orders and the shifts need to be taken into account to preserve the semantics of the program. Previous work, such as the algorithm presented in chapter 3, has sidestepped this problem by fixing one global evaluation order (either call-by-value or call-by-name). This leads to our

> **Second contribution:** We give the first presentation on the interaction between data/codata types, type-based evaluation order, and de/refunctionalization. We discuss, and solve, the problems which arise when we combine them.

Overview

The rest of the chapter is structured as follows:

- In section 4.2 we present our central contributions in an example-driven, informal style.
- In section 4.3 we present the formalization of the CPS-fragment of the language. Since all programs have to be written with explicit control flow, different evaluation strategies cannot be observed.
- In section 4.4 we present the single algorithm which subsumes both defunctionalization and refunctionalization.
- In section 4.5 we remove the restriction to programs with explicit control flow by introducing the μ and $\tilde{\mu}$ constructs from $\bar{\lambda}\mu\tilde{\mu}$ calculus. We discuss different global and type-based evaluation orders and their influence on the validity of η-equalities.
- In section 4.6 we discuss how to complement the transformations from section 4.4 with transformations which change the evaluation order of a given type.
- In section 4.7 we discuss related work, and we conclude in section 4.8.

The system presented in this chapter has been formalized and theorems 4.3.4, 4.3.5 and 4.4.2 to 4.4.4 have been proven in Coq. We restrict ourselves to a simply-typed language, since this is sufficient to illustrate our central ideas. We think that the generalization to a polymorphic variant of this calculus which supports generalized algebraic data types (GADTs) and their dual, GAcoDTs, does not pose substantial difficulties and expect this to be a straight-forward adaption of the approach of Ostermann and Jabs (2018).

4.2. Main Ideas

In this section we will use simple examples to present the types and terms of the language, as well as the transformation algorithms.

4.2.1. Programming with Symmetric Data and Codata

Natural deduction, and the programming languages based on it, are biased towards proof. A natural deduction derivation ending with a single formula at its root proves that formula. Term assignment systems for natural deduction, such as the simply typed lambda calculus, therefore only have one typing judgement. This typing judgement, usually written $\Gamma \vdash t : T$, types a term t as a *proof* of the type T.

Sequent calculus, on the other hand, is not biased towards proofs. For example, there are not only sequent calculus derivations ending in $\vdash \phi$ which *prove* a formula ϕ, but also derivations ending in $\phi \vdash$ which *refute* a formula ϕ. In the context of programming languages, proofs and refutations correspond to *producers* and *consumers* of a type. For example, the producers of the type $\mathbb{N}$ are numbers $1, 2, 3 \ldots$, whereas the consumers are *continuations* expecting a natural number. The single typing judgement $\Gamma \vdash t : T$ is replaced by two separate judgements $\Gamma \vdash p : T$ and $\Gamma \vdash c \overset{\text{con}}{:} T$, one which types producers and one which types consumers.

There are exactly two kinds of types in our system, data types and codata types (Downen, Sullivan, Zena M. Ariola, and Jones, 2019; Hagino, 1989), whose difference can be expressed in terms of *canonical* producers and consumers. Data types have canonical producers, which are called *constructors*. Canonical means that given an arbitrary producer, we know that it must have been built by one among a finite list of constructors. Having only a finite list of constructors justifies the use of *pattern matching* to build consumers of a data type. Codata types, on the other hand, have canonical consumers, called *destructors*; their producers are formed by *copattern matching* (Abel, Pientka, Thibodeau, and Setzer, 2013) on all destructors.

First-class support for both producers and consumers is necessary to make data and codata types completely symmetric. For data types, constructors can be typed as producers and pattern matches as consumers. Dually, destructors of a codata type can be typed as consumers, and copattern matches as producers. Without a first-class representation of consumers we would have to treat data and codata types asymmetrically: both pattern matches and destructors need an additional return type. We will now illustrate symmetric data and codata types with some simple examples.

The data type $\mathbb{N}$ is defined by two constructors `Zero` and `Suc`, the canonical producers of $\mathbb{N}$. Since we distinguish producers from consumers, we have to explicitly mark the argument of the constructor `Suc` as a producer.

$$\textbf{data type } \mathbb{N} \ \{ \ \texttt{Zero} \ ; \ \texttt{Suc}(x : \mathbb{N}) \ \}$$

Using this definition, we can now form the following producer and consumer of $\mathbb{N}$:

$$\texttt{Suc(Zero)} : \mathbb{N}$$
$$\textbf{match}_{\textbf{data}} \ \mathbb{N} \ \{ \ \texttt{Zero} \ \Rightarrow \ \ldots \ ; \ \texttt{Suc}(x : \mathbb{N}) \Rightarrow \ldots \ \} \overset{\text{con}}{:} \mathbb{N}$$

The definition of the consumer is still incomplete; we have not yet specified what should be inserted in the place of the two holes. We do not have a "result type" which would determine the type of terms to put in the holes; rather, the category of terms to insert in these places are called *commands*, which we discuss next.

Commands, sometimes also called "statements", are the syntactical category of reducible expressions; a closed command corresponds to the state of an abstract machine. We consider only two types of command. A logical command $\langle\, p \mid c \,\rangle$ combines a producer p and consumer c of the same type. Logical commands are evaluated using standard (co)pattern matching evaluation rules. The second type of command is **Done** which just terminates the program. The addition of **Done** is necessary since the Curry-Howard interpretation of a command is a contradiction (Zeilberger, 2008b). The only way to write a closed command is therefore to write a looping program using unrestricted recursion, or to postulate the existence of a closed command, which we have done here. We now present the first simple example of a complete program which reduces to **Done** in a single step before terminating.

$$\langle\, \texttt{Suc}(\texttt{Zero}) \mid \mathbf{match_{data}}\ \mathbb{N}\ \{\ \texttt{Zero}\ \Rightarrow\ \mathbf{Done}\ ;\ \texttt{Suc}(x) \Rightarrow \mathbf{Done}\}\,\rangle$$

We now consider some examples of codata types. The first example of a codata type, and the only codata type in many functional programming languages, is the function type. Codata types allow the function type to be user-defined instead of being hardwired into the language. The type of functions from $\mathbb{N}$ to $\mathbb{N}$ is represented by a codata type with one destructor `Ap`. This destructor corresponds to the only way a function can be used, namely to apply it to an argument. In the symmetric setting `Ap` takes two arguments, the producer argument x for the value passed to the function, and the consumer argument k for the consumer to be used on the result of the evaluation of the function[1].

$$\mathbf{codata\ type}\ \mathbb{N}{\to}\mathbb{N}\ \{\ \texttt{Ap}(x : \mathbb{N}, k \overset{\mathbf{con}}{:} \mathbb{N})\ \}$$

The identity function $\lambda x.x$ can be written as a comatch.

$$\mathrm{id} \coloneqq \mathbf{match_{codata}}\ \mathbb{N}{\to}\mathbb{N}\ \{\ \texttt{Ap}(x,k) \Rightarrow \langle\, x \mid k \,\rangle\} : \mathbb{N}{\to}\mathbb{N}$$

Codata types also allow for easy and intuitive programming with infinite structures. For example, the type of streams of natural numbers is defined by the two destructors which give the head and the tail of a stream. Note that the `Head` destructor does not directly return the first element; instead, a continuation for $\mathbb{N}$ has to be passed as an argument.

$$\mathbf{codata\ type}\ \mathbb{N}{-}\mathrm{Stream}\ \{\ \texttt{Head}(k \overset{\mathbf{con}}{:} \mathbb{N})\ ;\ \texttt{Tail}(k \overset{\mathbf{con}}{:} \mathbb{N}{-}\mathrm{Stream})\ \}$$

Codata also formalizes one essential aspect of object-oriented programming: programming against an interface (Cook, 1990; Cook, 2009). We give a simple example of a

[1]Readers familiar with linear logic might recognize this as the following decomposition of the function type: $\phi \multimap \psi = (\phi \otimes \psi^{\perp})^{\perp}$.

customer "interface" with name and address fields; an "object" that implements the interface could again be constructed with a copattern match.

$$\textbf{codata type}\ \text{Customer}\ \{\ \texttt{Name}(k \overset{\text{con}}{:} \text{String})\ ;\ \texttt{Address}(k \overset{\text{con}}{:} \text{Address})\ \}$$

In the next section we will see how symmetric data and codata types can be transformed back and forth other using the de/refunctionalization algorithms, and why we only need a single algorithm in the symmetric setting.

4.2.2. Defunctionalization and Refunctionalization

Having symmetric data and codata types turns de/refunctionalization into *one* algorithm, like we promised in the introduction. We illustrate this with a simple example program manipulating natural numbers. At first, the type of natural numbers is represented as a data type:

Example 4.2.1 (Natural numbers as a data type).

$\textbf{data}\ \mathbb{N}\ \{\ \texttt{Zero},\ \texttt{Suc}(x : \mathbb{N})\ \}$

$\texttt{pred}(k \overset{\text{con}}{:} \mathbb{N}) \coloneqq \textbf{match}_{\textbf{data}}\ \mathbb{N}\ \{\texttt{Zero} \Rightarrow \langle\, \texttt{Zero} \mid k\,\rangle,\ \texttt{Suc}(x) \Rightarrow \langle\, x \mid k\,\rangle\}$

$\texttt{add}(y : \mathbb{N}, k \overset{\text{con}}{:} \mathbb{N}) \coloneqq \textbf{match}_{\textbf{data}}\ \mathbb{N}\ \{\texttt{Zero} \Rightarrow \langle\, y \mid k\,\rangle,\ \texttt{Suc}(x) \Rightarrow \langle\, x \mid \texttt{add}(\texttt{Suc}(y), k)\,\rangle\}$

The natural numbers are represented by the two canonical constructors `Zero` and `Suc`, and the functions `pred` and `add` are defined by pattern matching. The same program, but with natural numbers represented as a codata type, looks as follows:

Example 4.2.2 (Natural numbers as a codata type).

$\textbf{codata}\ \mathbb{N}\ \{\ \texttt{pred}(k \overset{\text{con}}{:} \mathbb{N}), \texttt{add}(y : \mathbb{N}, k \overset{\text{con}}{:} \mathbb{N})\}$

$\texttt{Zero} \coloneqq \textbf{match}_{\textbf{codata}}\ \mathbb{N}\ \{\texttt{pred}(k) \Rightarrow \langle\, \texttt{Zero} \mid k\,\rangle,\ \texttt{add}(y, k) \Rightarrow \langle\, y \mid k\,\rangle\}$

$\texttt{Suc}(x : \mathbb{N}) \coloneqq \textbf{match}_{\textbf{codata}}\ \mathbb{N}\ \{\texttt{pred}(k) \Rightarrow \langle\, x \mid k\,\rangle,\ \texttt{add}(y, k) \Rightarrow \langle\, x \mid \texttt{add}(\texttt{Suc}(y), k)\,\rangle\}$

De/Refunctionalization is now just matrix transposition; the terms themselves remain unchanged. The same program which was represented above both in data and codata centric form can be presented as the following matrix.

Example 4.2.3 (Natural numbers in matrix form).

$\mathbb{N}$	`Zero`	$\texttt{Suc}(x : \mathbb{N})$
$\texttt{pred}(k \overset{\text{con}}{:} \mathbb{N})$	$\langle\, \texttt{Zero} \mid k\,\rangle$	$\langle\, x \mid k\,\rangle$
$\texttt{add}(y : \mathbb{N}, k \overset{\text{con}}{:} \mathbb{N})$	$\langle\, y \mid k\,\rangle$	$\langle\, x \mid \texttt{add}(\texttt{Suc}(y), k)\,\rangle$

In the asymmetric setting of previous work (chapter 3, as well as Rendel, Trieflinger, and Ostermann (2015) and Ostermann and Jabs (2018)), complications due to the asymmetry prevented this easy formulation. For instance, asymmetric destructors are defunctionalized to functions with a "special" argument (the `this` object, in OO terminology), so different kinds of function declarations and function calls had to be distinguished in the formalization. For the system presented so far, and for the constructs used in examples 4.2.1 and 4.2.2, it is clear after some reflection that de/refunctionalization is semantics-preserving. This is essentially due to the fact that the program still contains the same "rewrites" (cases of (co)pattern matches) - only the place where the rewrites are defined changes -, and there is no notion of evaluation order; all control flow is completely explicit. In the next subsection we introduce additional control flow constructs which introduce the need for evaluation strategies.

4.2.3. Evaluation Order

In order to speak about evaluation order, we need to add additional constructs which let us form new commands for which the evaluator has to make a *choice* on what to evaluate next. We do this by introducing the μ and $\tilde{\mu}$ abstractions from the $\bar{\lambda}\mu\tilde{\mu}$ calculus of Curien and Herbelin (2000). These two constructs introduce new ways to form producers and consumers.

A μ-abstraction $\mu(x \overset{\mathbf{con}}{:} T).c$ is a producer of T which abstracts over a consumer x of T in the command c. Dually, a $\tilde{\mu}$-abstraction $\mu(x : T).c$ is a consumer of T which abstracts over a producer x of T. Note that in distinction to Curien and Herbelin (2000) we use different type annotations on the variable instead of the tilde to distinguish the two kind of abstractions. This allows us to generalize over them syntactically in the formalization.

We illustrate their intuitive meaning using the data type of natural numbers and the functions given in example 4.2.1. The only terms that can be typed as producers of $\mathbb{N}$ in the system without μ abstractions are generated by the grammar $v \coloneqq \text{Zero} \,|\, \text{Suc}(v) \,|\, x$. In particular, there was no way to type an analogue of $2 + 2$ as a producer of $\mathbb{N}$ with is not yet fully evaluated. Using a μ-abstraction, we can now form the producer $\mu(k \overset{\mathbf{con}}{:} \mathbb{N}).(\langle\, 2 \mid \text{add}(2, k)\,\rangle)$. The $\tilde{\mu}$ abstraction for $\mathbb{N}$, on the other hand, behaves more like a let-binding for producers in a command. For example, the command $\langle\, 5 \mid \mu(x : \mathbb{N}).c\,\rangle$ behaves like $\textbf{let } x : \mathbb{N} \coloneqq 5 \textbf{ in } c$.

Once we have both μ and $\tilde{\mu}$ abstractions in the language confluence is lost, as witnessed by the following critical pair. In this example, Ω refers to some unspecified nonterminating command. Depending on which abstraction we evaluate first, we obtain either Ω or **Done**. This is clearly unsatisfactory.

$$\Omega \xleftarrow[\triangleleft]{\qquad} \langle\, \mu(x \overset{\mathbf{con}}{:} T).\Omega \mid \mu(y : T).\mathbf{Done}\,\rangle \xrightarrow[\triangleright]{\qquad} \mathbf{Done} \tag{4.1}$$

We regain confluence by prescribing an evaluation strategy for the redex above; either call-by-value or call-by-name. In call-by-value, only producers which belong to the more restrictive grammar, not containing μ-abstractions, can be substituted for producer variables. In particular, we cannot substitute the producer corresponding to $2+2$ for a producer variable of type $\mathbb{N}$. More generally: Under call-by-value we evaluate μ-abstractions before $\tilde{\mu}$ abstractions, and conversely for call-by-name.

This leads to the problem of which evaluation order to choose for the system. This choice should strive to maximize the number of valid η-equalities. For example, the η equality for the function type, assuming that x and k do not occur free in e, is:

$$\mathbf{match}_{\mathbf{codata}}\ \mathbb{N}{\to}\mathbb{N}\ \{\ \mathtt{Ap}(x,k) \Rightarrow \langle\, e \mid \mathtt{Ap}(x,k)\,\rangle\} =_\eta e$$

This rule makes the following two commands η-equivalent:

$$\langle\, \mathbf{match}_{\mathbf{codata}}\ \mathbb{N}{\to}\mathbb{N}\ \{\mathtt{Ap}(x,k) \Rightarrow \langle\, \mu(y \overset{\mathbf{con}}{:} \mathbb{N}{\to}\mathbb{N}).\Omega \mid \mathtt{Ap}(x,k)\,\rangle\} \mid \mu(x : \mathbb{N}{\to}\mathbb{N}).\mathbf{Done}\,\rangle$$
$$\langle\, \mu(y \overset{\mathbf{con}}{:} \mathbb{N}{\to}\mathbb{N}).\Omega \mid \mu(x : \mathbb{N}{\to}\mathbb{N}).\mathbf{Done}\,\rangle$$

But under the call-by-value strategy they evaluate to different results. In summary, we can thus observe that generally, η-rules are only valid for data types when evaluated under call-by-value, while they are only valid for codata types when evaluated under call-by-name. We can turn this observation into an evaluation strategy: Under the *polar* evaluation order, all data types are evaluated using call-by-value and all codata types using call-by-name.

4.2.4. De/Refunctionalization and evaluation order

What consequences does the introduction of evaluation order have for de/refunctionalization? The central observation is that we can combine the simple de/refunctionalization algorithm with global **cbv** or **cbn**, *but we cannot combine this simple algorithm with the polar evaluation order.* In order to see this, consider the critical pair in eq. (4.1). Whatever evaluation order we choose for this redex, we must use the same evaluation order for it after defunctionalizing or refunctionalizing the type T. This is guaranteed if we fix a global evaluation order, but it fails if the evaluation order is dependent on whether T is a data or codata type.

The following table summarizes this situation. If we choose call-by-value we loose the η-equalities for codata types, and if we choose call-by-name we loose the η-equalities for data types. But in both cases we can use the simple de/refunctionalization algorithm described above. If we choose the polar evaluation order, on the other hand, all η-equalities are valid, but the de/refunctionalization algorithm is *no longer semantics-preserving.*

Eval Order	η for data	η for codata	De/Refunc.
Global **cbv**	✓	✗	✓
Global **cbn**	✗	✓	✓
Polar	✓	✓	✗

How do we combine de/refunctionalization with the polar evaluation order, which validates all η-equalities? To solve this problem, we introduce a fourth evaluation order: the “nominal” evaluation order, where each type explicitly declares whether it should be evaluated by-value or by-name. This scheme allows the declaration of call-by-name data types and call-by-value codata types. However, we consider these two new types to be mere intermediate steps in the translation from call-by-value data types to call-by-name codata types, and vice versa, since they are not well-behaved when we consider their η-laws.

4.2.5. Summary

The following diagram gives a concise summary of how the two algorithms presented in this chapter interact.

$$\begin{tikzcd}
\textbf{cbv data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] & \textbf{cbv codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] \\
\textbf{cbn data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"] & \textbf{cbn codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"]
\end{tikzcd}$$

We factor the *full de/refunctionalization* transformation $\mathcal{F}_p$ into two simpler transformations: *core defunctionalization*[2] $\mathcal{C}_{\textbf{data}}$ and *core refunctionalization* $\mathcal{C}_{\textbf{codata}}$ exchange data and codata, as well as *evaluation order switch*, $\mathcal{T}_{\textbf{cbn}}$ and $\mathcal{T}_{\textbf{cbv}}$. The former will perform the essential part of de/refunctionalization, i.e. exchanging data with codata types and vice versa without any changes to the evaluation strategy. Correspondingly, the latter will only change the evaluation strategy, while keeping the semantics of all existing expressions intact. For this transformation we have to add appropriate *shift types* $\uparrow^{\textbf{cbn}} T$ and $\uparrow^{\textbf{cbv}} T$ to the program for every type T. Shift types were originally introduced by Girard (2001) in the context of ludics and polarized logic, where they mediate between positive and negative types. Their use to control the evaluation order has been described more accessibly by Zeilberger (2008b) and Downen and Zena M. Ariola (2018b). These transformations make the diagram commute up to the insertion of some double-shifts, which can be removed in the special case of a mere round-trip. This results in the full defunctionalization $\mathcal{F}_{\textbf{data}} = \mathcal{T}_{\textbf{cbv}} \circ \mathcal{C}_{\textbf{data}}$ and full refunctionalization $\mathcal{F}_{\textbf{codata}} = \mathcal{T}_{\textbf{cbn}} \circ \mathcal{C}_{\textbf{codata}}$.

Let us look at an example of a program which is transformed along the path *top left* $\rightarrow$ *top right* $\rightarrow$ *bottom right* in the above diagram. Note that all `main` commands evaluate to **Done**, which is a good indication that the transformations are semantics preserving. We start with the following program in the top left corner:

Example 4.2.4 (Original program)**.**

$$\begin{aligned}
&\textbf{cbv data type}\ \mathbb{N}\ \{\ \texttt{Zero}\ ;\ \texttt{Suc}(x : \mathbb{N})\ \}\\
&\texttt{pred}(k \overset{\textbf{con}}{:} \mathbb{N}) \coloneqq \textbf{match}_{\textbf{data}}\ \mathbb{N}\ \{\texttt{Zero} \Rightarrow \langle\, \texttt{Zero} \mid k \,\rangle,\ \texttt{Suc}(n) \Rightarrow \langle\, n \mid k \,\rangle\}\\
&\texttt{main} \coloneqq \langle\, \mu(k \overset{\textbf{con}}{:} \mathbb{N}).\textbf{Done} \mid \mu(n : \mathbb{N}).\Omega \,\rangle
\end{aligned}$$

After refunctionalizing this program by applying $\mathcal{C}_{\textbf{codata}}$ we obtain the following program which lives in the top right corner:

Example 4.2.5 (After refunctionalization)**.**

$$\begin{aligned}
&\textbf{cbv codata type}\ \mathbb{N}\ \{\ \texttt{pred}(k \overset{\textbf{con}}{:} \mathbb{N})\ \}\\
&\texttt{Zero} \coloneqq \textbf{match}_{\textbf{codata}}\ \mathbb{N}\ \{\texttt{pred}(k) \Rightarrow \langle\, \texttt{Zero} \mid k \,\rangle\}\\
&\texttt{Suc}(n : \mathbb{N}) \coloneqq \textbf{match}_{\textbf{codata}}\ \mathbb{N}\ \{\texttt{pred}(k) \Rightarrow \langle\, n \mid k \,\rangle\}\\
&\texttt{main} \coloneqq \langle\, \mu(k \overset{\textbf{con}}{:} \mathbb{N}).\textbf{Done} \mid \mu(n : \mathbb{N}).\Omega \,\rangle
\end{aligned}$$

[2]In the following, we will often refer to core de/refunctionalization simply as just de/refunctionalization.

In the last step we apply the cbv-to-cbn translation $\mathcal{T}_{\textbf{cbn}}$ to obtain the following program in the lower right corner:

Example 4.2.6 (Final result)**.**

$$\begin{array}{l}
\textbf{cbn codata type } \mathbb{N} \ \{ \ \texttt{pred}(k \overset{\textbf{con}}{:} \uparrow^{\textbf{cbv}} \mathbb{N}) \ \} \\
\textbf{cbv data type } \uparrow^{\textbf{cbv}} \mathbb{N} \ \{ \ \texttt{CBV}(x : \mathbb{N}) \ \} \\
\texttt{Zero} \coloneqq \textbf{match}_{\textbf{codata}} \ \mathbb{N} \ \{\texttt{pred}(k) \Rightarrow \langle \, \texttt{CBV(Zero)} \mid k \, \rangle\} \\
\texttt{Suc}(n : \uparrow^{\textbf{cbv}} \mathbb{N}) \coloneqq \textbf{match}_{\textbf{codata}} \ \mathbb{N} \ \{\texttt{pred}(k) \Rightarrow \langle \, n \mid k \, \rangle\} \\
\texttt{main} \coloneqq \langle \, \mu(k \overset{\textbf{con}}{:} \uparrow^{\textbf{cbv}} \mathbb{N}).\textbf{Done} \mid \mu(n : \uparrow^{\textbf{cbv}} \mathbb{N}).\Omega \, \rangle
\end{array}$$

4.3. Formalization

In this section we will formally describe the syntax, type system and operational semantics of our language. We will extend this language in section 4.5 with constructs which allow to write programs in direct style.

The inherent symmetry of the system allows us to minimize the number of rules, since we don't have to write down separate rules for data and codata types, matches and comatches, constructors and destructors. In order to do this, we introduce polarities p which can either be **data** or **codata**. For example, a $\textbf{match}_{\textbf{data}}$ is a pattern match and a $\textbf{match}_{\textbf{codata}}$ is a copattern match and so on. The syntax of our language is defined in definition 4.3.1, and the typing and well-formedness rules are given in figs. 4.1 and 4.2, which we will now explain in turn.

Definition 4.3.1 (Program declarations and terms)**.** The syntax of terms and programs is given by the following grammar.

$$\begin{array}{lll r}
x, y \in \textsc{Var} & T \in \textsc{Tname} & \mathcal{X}, \mathcal{Y} \in \textsc{Name} & \textit{Variables and Names} \\
p & ::= & \textbf{data} \mid \textbf{codata} & \textit{Polarity} \\
s & ::= & \textbf{cbv} \mid \textbf{cbn} & \textit{Evaluation Strategy} \\
o & ::= & \textbf{prd} \mid \textbf{con} & \textit{Orientation} \\
\Gamma, \Delta, \Pi & ::= & \diamond \mid \Gamma, x \overset{o}{:} T & \textit{Contexts} \\
\sigma, \tau & ::= & () \mid (\sigma, e) & \textit{Substitutions} \\
e & ::= & x \mid \mathcal{X}\sigma \mid \textbf{match}_p T \{ \overline{\mathcal{X}\Delta \Rightarrow c} \} & \textit{Expressions} \\
c & ::= & \langle \, e \mid e \, \rangle \mid \textbf{Done} & \textit{Commands} \\
d & ::= & s \ p \ \textbf{type} \ T \ \{ \ \overline{\mathcal{X}\Delta} \ \} \ \textbf{with} \ \overline{f} & \textit{Type declarations} \\
f & ::= & \mathcal{X}\Pi \coloneqq \textbf{match}_p \ T \ \{\overline{\mathcal{Y}\Delta \Rightarrow c}\} & \textit{Functions} \\
P & ::= & (\overline{d}, \ c) & \textit{Program}
\end{array}$$

When specifying various of the rules and transformations which govern the system we also use the following helper functions:

Definition 4.3.2 (Helper functions).

$$\begin{aligned} \mathbf{val}(\mathbf{data}) &\coloneqq \mathbf{prd} & \qquad \mathbf{val}(\mathbf{codata}) &\coloneqq \mathbf{con} \\ \mathbf{cnt}(\mathbf{data}) &\coloneqq \mathbf{con} & \mathbf{cnt}(\mathbf{codata}) &\coloneqq \mathbf{prd} \\ \widehat{\mathbf{data}} &\coloneqq \mathbf{codata} & \widehat{\mathbf{codata}} &\coloneqq \mathbf{data} \\ \widehat{\mathbf{cbv}} &\coloneqq \mathbf{cbn} & \widehat{\mathbf{cbn}} &\coloneqq \mathbf{cbv} \\ \widehat{\mathbf{prd}} &\coloneqq \mathbf{con} & \widehat{\mathbf{con}} &\coloneqq \mathbf{prd} \end{aligned}$$

We use the set TNAME for names of types, and the set NAME for for names of constructors, destructors and functions. Whenever something can stand for either a constructor or destructor, we refer to t as an *xtor*. All the typing rules implicitly have a program in their context, and we use lookup functions to obtain various information about the declarations in that program.

4.3.1. Type declarations and the program

A *program* P consists of a list of data and codata type declarations d and an entry point in the form of a top-level command. In order to check that a program is well-formed, we have to use the rule WF-PROG to check that the entry point typechecks as a command, which in turn uses the rule WF-TYPE to check that each of the type declarations is correct. Note that we do not consider declarations to be ordered and all data and codata types, constructors, destructors and functions may reference one another in mutual recursion. We also require all names that are used to be unambiguous, but we don't write down the obvious rules.

A *type* declaration $s\;p\;\mathbf{type}\;T\;\{\;\overline{\mathcal{X}\Delta}\;\}\;\mathbf{with}\;\overline{f}$ introduces a data or codata type T (with evaluation strategy s) by specifying both its xtors $\mathcal{X}_i\Delta_i$ and a list of functions f_i which pattern match on its xtors. The only reason why the xtors of a type have to be declared together with the functions matching on them is to allow for a simpler presentation of the algorithm in section 4.4; a real programming language implementing these ideas would not use this restriction. Each function declaration $\mathcal{Y}\Pi \coloneqq \mathbf{match}_p\,T\,\{\overline{\mathcal{X}\Delta \Rightarrow c}\}$ declares a function $\mathcal{X}$ with arguments Π by (co)pattern matching on all xtors of the type T to which they belong. The rule WF-FUN uses the expression typing judgement introduced above to typecheck these global (co)pattern matches.

4.3.2. Contexts and Substitutions

The definition of data and codata types, constructors and destructors, pattern matching and copattern matching can be formulated more concisely and uniformly by using *contexts* and *substitutions*. Consider the example program from example 4.2.1, where natural numbers are defined by two constructors: `Zero` and `Suc`. `Zero` does not bind anything, but `Suc` is defined using the context $\Delta = x : \mathbb{N}$. This context Δ determines simultaneously that pattern matching on a `Suc` constructor extends the context by Δ, and that in order to construct a natural number with `Suc`, a substitution σ for Δ has to be provided.

$\boxed{\text{Context Formation: } \vdash \Gamma}$

$$\frac{}{\vdash \diamond}\ \textsc{T-Ctx}_1 \qquad \frac{\vdash \Gamma \quad T \in \text{Prog} \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma, x \overset{o}{:} T}\ \textsc{T-Ctx}_2$$

$\boxed{\text{Substitution typing: } \Gamma \vdash \sigma : \Delta}$

$$\frac{}{\Gamma \vdash () : \diamond}\ \textsc{T-Subst}_1 \qquad \frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash e \overset{o}{:} T \qquad \textsc{Subst}(e)}{\Gamma \vdash (\sigma, e) : \Delta, x \overset{o}{:} T}\ \textsc{T-Subst}_2$$

$\boxed{\text{Expression typing: } \Gamma \vdash e \overset{o}{:} T}$

$$\frac{\Gamma(x) = (o, T)}{\Gamma \vdash x \overset{o}{:} T}\ \textsc{T-Var}$$

$$\frac{\mathcal{X}\Delta \in \text{Xtors}(T) \qquad \text{Polarity}(T) = p \qquad \Gamma \vdash \tau : \Delta}{\Gamma \vdash \mathcal{X}\tau \overset{\mathbf{val}(p)}{:} T}\ \textsc{T-Xtor}$$

$$\frac{\mathcal{X}\Pi \coloneqq \ldots \in \text{Funs}(T) \qquad \text{Polarity}(T) = p \qquad \Gamma \vdash \tau : \Pi}{\Gamma \vdash \mathcal{X}\tau \overset{\mathbf{cnt}(p)}{:} T}\ \textsc{T-Fun}$$

$$\frac{\Gamma, \overline{\Delta} \vdash \overline{c} \qquad \text{(Side condition: see text)}}{\Gamma \vdash \mathbf{match}_p\, T\ \{\, \overline{\mathcal{X}\Delta \Rightarrow c}\,\} \overset{\mathbf{cnt}(p)}{:} T}\ \textsc{T-Match}$$

$\boxed{\text{Command typing: } \Gamma \vdash c}$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 \overset{\mathbf{con}}{:} T}{\Gamma \vdash \langle\, e_1 \mid e_2 \,\rangle}\ \textsc{T-Cut} \qquad \frac{}{\Gamma \vdash \mathbf{Done}}\ \textsc{T-Done}$$

Figure 4.1.: Typing rules.

$$\frac{\Pi \vdash \mathbf{match}_p\ T\ \{\overline{\mathcal{X}\Delta \Rightarrow c}\} \overset{\mathbf{cnt}(p)}{:} T}{\vdash \mathcal{X}\Pi \coloneqq \mathbf{match}_p\ T\ \{\overline{\mathcal{X}\Delta \Rightarrow c}\} \overset{\mathbf{cnt}(p)}{:} T\ \textsc{Ok}}\ \textsc{Wf-Fun}$$

$$\frac{\vdash \overline{\Gamma} \qquad \vdash \overline{f}\ \textsc{Ok}}{\vdash s\ p\ \mathbf{type}\ T\ \{\ \overline{\mathcal{X}\Delta}\ \}\ \mathbf{with}\ \overline{f}\ \textsc{Ok}}\ \textsc{Wf-Type} \qquad \frac{\vdash \overline{d}\ \textsc{Ok} \qquad \vdash c}{\vdash (\overline{d}, c)\ \textsc{Ok}}\ \textsc{Wf-Prog}$$

Figure 4.2.: Well-formedness rules.

Contexts map variables from the set VAR to their type and orientation (producer or consumer) and are constructed according to the rules T-CTX$_1$ and T-CTX$_2$ from fig. 4.1. A substitution σ for a context Δ consists of one expression for each variable in Δ. We use the typing judgement $\Gamma \vdash \sigma : \Delta$ to express that each expression in σ can be typed in the context Γ with the type and orientation specified in the context Δ. The meaning of the SUBST(e) premise in the rule T-SUBST$_2$ will be explained in section 4.5 and is irrelevant for now, as it holds for all expressions in the current setting. Note that there is an obvious identity substitution id_Γ for every context which satisfies $\Pi, \Gamma \vdash \mathrm{id}_\Gamma : \Gamma$.

4.3.3. Expressions and Commands

While we have only one syntactic category e of expressions, there are two different typing judgements for expressions; expressions can either be typed as a producer with $\Gamma \vdash e : T$ or as a consumer with $\Gamma \vdash e \overset{\mathbf{con}}{:} T$. In all, there are four different rules which govern the typing of expressions.

The rule T-VAR is quite self-explanatory, we just have to look up both the type and the orientation of the variable in the context.

The rule T-XTOR covers the typing of both constructors and destructors, which we collectively call *xtors*. Constructors have to be typed as producers, and destructors as consumers; the rule accomplishes this by looking up the type T to which the xtor belongs in the program, and the polarity p of that type. Looking up the xtor in the program also tells us what arguments we have to provide in the substitution τ, and the helper function **val**(p) guarantees that the result is typechecked with the correct orientation.

The rule T-FUN is very similar to the rule T-XTOR. Instead of typechecking constructors and destructors, it governs the call of functions declared in the program. The difference between T-XTOR and T-FUN is that we don't look up the signature of a constructor or destructor, but the declaration of a function, i.e. its signature together with its body. If the polarity of the type to which the function belongs is **data**, then the function is defined by pattern matching and the function call has to be typed as a consumer; similarly, if the polarity is **codata**, the function is defined by copattern matching and the call must be typed as a producer. The helper function **cnt**(p) ensures that this is the case.

The rule T-MATCH covers local pattern and copattern matches. In that rule we have to check that the right-hand side of each case typechecks as a command. In each case we extend the outer context with the context bound by the constructor or destructor. Pattern matching has to be exhaustive, and the arguments bound in the case have to be identical to the ones declared in the program. We have omitted these requirements in the formulation of the rule T-MATCH in order to keep it more legible.

There are two ways to form commands. A logical command $\langle\, e_1 \mid e_2 \,\rangle$ consists of two expressions; the expression e_1 has to typecheck as producer and the expression e_2 as a consumer of the same type. The rule is named after the "Cut" rule from sequent calculus, to which it corresponds. The command **Done** typechecks in any context and corresponds to a successfully terminated computation.

4.3.4. Operational Semantics

Reduction only applies to closed commands, and the rules are given in definition 4.3.3.

Definition 4.3.3 (Operational semantics)**.** The reduction rules for local matches and comatches are:

$$\langle\, \mathcal{X}\sigma \mid \mathbf{match_{data}}\, T\,\{\, \mathcal{X}\Delta \Rightarrow c;\, \ldots \,\}\,\rangle \;\triangleright\, c\,\sigma \qquad \text{(Match)}$$
$$\langle\, \mathbf{match_{codata}}\, T\,\{\, \mathcal{X}\Delta \Rightarrow c;\, \ldots \,\} \mid \mathcal{X}\sigma \,\rangle \triangleright c\,\sigma \qquad \text{(Comatch)}$$

The reduction rules for calls of producers and consumers are:

$$\langle\, \mathcal{Y}\tau \mid \mathcal{X}\sigma \,\rangle \triangleright\; (c\;\tau)\,\sigma \quad \text{if } (\mathcal{X}\Pi \coloneqq \mathcal{Y}\Delta \Rightarrow c; \ldots) \in \textsc{Prog} \qquad \text{(ConCall)}$$
$$\langle\, \mathcal{X}\sigma \mid \mathcal{Y}\tau \,\rangle \triangleright\; (c\;\tau)\,\sigma \quad \text{if } (\mathcal{X}\Pi \coloneqq \mathcal{Y}\Delta \Rightarrow c; \ldots) \in \textsc{Prog} \qquad \text{(PrdCall)}$$

Suppose that c is well-typed in the context Γ, Δ, and that σ is a substitution from the empty context for Δ, i.e. $\diamond \vdash \sigma : \Delta$. Then the result of substituting σ in c for the variables from Δ, which we write $c\ \sigma$, is well-defined and typechecks in the context Γ. We don't give the full rules for substitution, since these are obvious but involve the usual technical complications of variable capture. Since all the xtors of the corresponding type must occur exactly once inside a match, their order does not matter and we will thus adopt the notational convention that the matching case in a pattern match is written as its first case. When a value (constructor or destructor) meets a continuation (pattern match or copattern match), evaluation proceeds by straightforward substitution. The case is slightly more difficult if a constructor meets the call of a globally defined function. In that case we have the two contexts, Γ and Δ, where the function $\mathcal{X}$ is declared in the program. In that case c is typed in the context Γ, Δ, and σ and τ are substitutions for Γ and Δ, respectively. The **Done** command is in normal form and cannot be evaluated further.

4.3.5. Type Soundness

The soundness of the system has been mechanically verified in Coq by formalizing the following two theorems.

Theorem 4.3.4 (Preservation)**.** *If* $\diamond \vdash c_1$ *and* $c_1 \triangleright c_2$*, then* $\diamond \vdash c_2$

Theorem 4.3.5 (Progress)**.** *If* $\diamond \vdash c_1$ *then either* $c_1 = \textbf{Done}$ *or there exists a command* c_2 *such that* $c_1 \triangleright c_2$*.*

4.4. Symmetric De/Refunctionalization

In this section we explain how to change the polarity of a type T via core defunctionalization $\mathcal{C}_{\mathbf{data}}$ and core refunctionalization $\mathcal{C}_{\mathbf{codata}}$, which are subsumed by one algorithm $\mathcal{C}_p$. That is, we show how to implement the horizontal transformations of the diagram introduced in section 4.2.

$$\begin{tikzcd}
\textbf{cbv data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] & \textbf{cbv codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] \\
\textbf{cbn data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"] & \textbf{cbn codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"]
\end{tikzcd}$$

We make one simplifying assumption; we assume that the program does not contain any local pattern matches on T. This does not reduce the expressiveness of the system, but simplifies the presentation of the algorithm.[3]

In definition 4.3.1 we used the same set $\textsc{Name}$ for the names of both constructors/destructors and function calls. This choice was motivated by the fact that core de/refunctionalization should be the identity function on terms. Typing derivations, on the other hand, are affected by de/refunctionalization. For example, if a constructor $\mathcal{X}$ with arguments Δ is declared for the data type T, then a corresponding function declaration $\mathcal{X}$ will be declared for the codata type T in the refunctionalized program. That is, the the following type derivation for a term $\mathcal{X}\sigma$ on the top will be replaced by the derivation below, where $\mathcal{D}$ is some derivation for $\Gamma \vdash \sigma : \Delta$ and $\mathcal{D}'$ is the corresponding derivation for the same judgement within the refunctionalized program.

$$\frac{\mathcal{X}\Delta \in \textsc{Xtors}(T) \qquad \text{Polarity}(T) = \textbf{data} \qquad \overset{\mathcal{D}}{\Gamma \vdash \sigma : \Delta}}{\Gamma \vdash \mathcal{X}\sigma : T}\ \textsc{T-Xtor}$$

$$\frac{\mathcal{X}\Delta \in \textsc{Funs}(T) \qquad \text{Polarity}(T) = \textbf{codata} \qquad \overset{\mathcal{D}'}{\Gamma \vdash \sigma : \Delta}}{\Gamma \vdash \mathcal{X}\sigma : T}\ \textsc{T-Fun}$$

De/Refunctionalization $\mathcal{C}_p^T$ does not change any type declaration apart from the type declaration for T itself, which we now define. We do this by matrix transposition, as described by Ostermann and Jabs (2018). A type declaration

$$s\ p\ \textbf{type}\ T\ \{\ \overline{\mathcal{X}\Delta}\}\ \textbf{with}\ \bar{f}$$

is read into the first matrix of definition 4.4.1 and then transposed to obtain the second matrix. The second matrix is then used to generate the new type declaration.

Definition 4.4.1 (Defunctionalization and refunctionalization as transposition)**.**

[3] In order to lift this restriction, local pattern matches have to be first lambda-lifted and replaced by functions declarions. Only in the next step can the the algorithm presented in this section be used. Functions written by the programmer have to be distinguished from functions which are the result of lambda lifting by use of annotations. After de/refunctionalization of the program, all functions which resulted from lambda-lifting have to be inlined. All details about annotations, lifting and inlining have been described in chapter 3.

$$\begin{tikzcd}
\begin{array}{c|ccc}
\textsc{Dtor} \backslash \textsc{Fun} & \mathcal{X}_1\Gamma_1 & \cdots & \mathcal{X}_m\Gamma_m \\
\hline
\mathcal{Y}_1\Delta_1 & c_{1,1} & \cdots & c_{1,m} \\
\vdots & \vdots & & \vdots \\
\mathcal{Y}_n\Delta_n & c_{n,1} & \cdots & c_{n,m}
\end{array}
\arrow[r, bend left, "\text{defunctionalize}"] &
\begin{array}{c|ccc}
\textsc{Ctor} \backslash \textsc{Fun} & \mathcal{Y}_1\Delta_1 & \cdots & \mathcal{Y}_n\Delta_n \\
\hline
\mathcal{X}_1\Gamma_1 & c_{1,1} & \cdots & c_{n,1} \\
\vdots & \vdots & & \vdots \\
\mathcal{X}_m\Gamma_m & c_{1,m} & \cdots & c_{n,m}
\end{array}
\arrow[l, bend left, "\text{refunctionalize}"]
\end{tikzcd}$$

For the system presented in section 4.3, the following properties are easily established.

4.4.1. Properties of De/Refunctionalization

Since matrix transposition is its own inverse, it is obvious that defunctionalization and refunctionalization are mutually inverse:

Theorem 4.4.2 (Mutual Inverse)**.** *For every well-formed program P and data type T in P:*

$$\mathcal{C}^T_{\hat{p}}(\mathcal{C}^T_p(P)) = P$$

De/Refunctionalization also preserve well-formedness of programs.

Theorem 4.4.3 (Typeability preservation)**.** *If P is a well-formed program, then $\mathcal{C}^T_p(P)$ is also well-typed.*

De/Refunctionalization is semantics-preserving.

Theorem 4.4.4 (Semantic preservation)**.** *Let P be a program, with $\vdash P$ Ok, T a type of polarity p in P and c_1 a closed command in P, i.e. $\vdash c_1$. Furthermore, c_1 may not contain any local pattern matches on T. Then, if $c_1 \triangleright c_2$ in P, $c_1 \triangleright c_2$ in $\mathcal{C}^T_p(P)$.*

Proof. Since this transformation only transposes the matrix, the cell which is addressed in this matrix by a pair of Fun and Xtor does not change, thus the body which will be substituted for a command consisting of such a pair will be the same as before. □

4.5. Evaluation Order

In this section we will formally introduce a new syntactic construct to the language: μ-abstractions. The introduction of μ abstractions will create *critical pairs*. A critical pair is a redex that can be evaluated to two different commands which don't reduce to the same normal form. In order to "defuse" these critical pairs we have to introduce an evaluation order, which prescribes precisely how to evaluate these redexes. After presenting several different alternative evaluation strategies we will end with a *nominal* strategy in which every type declares how it's redexes are evaluated.

4.5.1. Extending the calculus with μ abstractions

We will now extend the syntax and typing rules with μ abstractions.

Definition 4.5.1 (Syntax of μ-abstractions).

$$e \quad ::= \quad \ldots \quad | \quad \mu(x \overset{o}{:} T).c \quad \textit{Expressions}$$

These μ-abstractions are governed by one additional typing rule

$$\frac{\Gamma, x \overset{o}{:} T \vdash c}{\Gamma \vdash \mu(x \overset{o}{:} T).c \overset{\hat{o}}{:} T}\ \text{T-}\mu$$

and the following two reduction rules

Definition 4.5.2 (Reduction of μ-abstractions).

$$\begin{aligned} \langle\, \mu(x \overset{\texttt{con}}{:} T).c \mid e \,\rangle &\ \triangleright\ c[e/x] \qquad \text{if } \textsc{Subst}(e) \qquad (\text{R-}\mu_1) \\ \langle\, e \mid \mu(x : T).c \,\rangle &\ \triangleright\ c[e/x] \qquad \text{if } \textsc{Subst}(e) \qquad (\text{R-}\mu_2) \end{aligned}$$

In the system of section 4.3 the syntax directly determined the only possible operational semantics. But with the addition of μ the calculus is no longer confluent, since there is a critical pair which is well-known in the $\bar{\lambda}\mu\tilde{\mu}$ literature: A command where μ and $\tilde{\mu}$ meet. In our setting, this corresponds to a producer $\mu(x \overset{\texttt{con}}{:} T).c_1$ and a consumer $\mu(x : T).c_2$, as presented in section 4.2.3.

$$c_1 \quad \triangleleft \quad \langle\, \mu(x \overset{\texttt{con}}{:} T).c_1 \mid \mu(y : T).c_2 \,\rangle \quad \triangleright \quad c_2$$

In order to guarantee confluence we therefore need the $\textsc{Subst}(e)$ predicate which we introduced in section 4.3. If we know that $\textsc{Subst}\big(\mu(x \overset{\texttt{con}}{:} T).c_1\big)$ and $\textsc{Subst}\big(\mu(y : T).c_2\big)$ never hold simultaneously for the same type T, then the two evaluation rules from definition 4.5.2 don't overlap.

A sensible definition of $\textsc{Subst}(e)$ will therefore always determine which μ abstraction to evaluate first. This corresponds precisely to specifying an *evaluation order*. In the following subsection we will therefore discuss some possible evaluation strategies.

4.5.2. Specifying evaluation strategies

First, let us specify precisely what we mean by "call-by-value" and "call-by-name" in this data-codata system. An evaluation strategy is characterized by what kind of expressions we are allowed to substitute for a variable. For example, call-by-value lambda calculus prohibits the substitution of non-canonical terms such as 1+1 for x in the redex $(\lambda x.t)(1+1)$. We can directly translate this example to the symmetric calculus using the functions from example 4.2.1. The term $1 + 1$ corresponds to $\mu(k \overset{\texttt{con}}{:} \mathbb{N}).\langle\, 1 \mid \texttt{add}(1, k) \,\rangle$. We don't allow this term to be substituted for a producer variable in a reduction step if we follow a call-by-value reduction strategy. Generalizing from this concrete example, call-by-value disallows the substitution of any μ-abstractions for producer variables in a reduction step.

An evaluation strategy is fully specified by the rules that govern the $\textsc{Subst}(e)$ predicate. All evaluation strategies share the rules of definition 4.5.3.

Definition 4.5.3 (Common rules for evaluation orders)**.**

$$\textsc{Subst}(\mathcal{X}\sigma) \quad \textsc{Subst}\big(\mathbf{match}_p\, T\,\{\ldots\}\big) \quad \textsc{Subst}(x)$$

These rules state that all function calls, xtors and local matches are substitutable. Moreover, variables are also substitutable. The reasoning behind the latter is as follows: Since expressions are only substituted into other expressions when they appear at the top-level of a command that is being reduced, they cannot be variables, as the command would not be closed otherwise. Thus, they must have been replaced with some other term in an earlier substitution step and this term must have been itself substitutable for this substitution to occur.

One way to specify an evaluating strategy is to prescribe a *global* evaluation order. For example, OCaml chose call-by-value, whereas Haskell chose call-by-name.[4] In our formalism, this corresponds to the definitions given in definition 4.5.4 and definition 4.5.5. Specifying a global evaluation order works, but has the disadvantage already indicated in the introduction: the validity of η is restricted to data types (under call-by-value) or codata types (under call-by-name).

Definition 4.5.4 (Global call-by-value evaluation order)**.** $\textsc{Subst}\big(\mu(x : T).c\big)$

Definition 4.5.5 (Global call-by-name evaluation order)**.** $\textsc{Subst}\big(\mu(x \overset{\mathbf{con}}{:} T).c\big)$

A better alternative is to specify evaluation order per-type; that is, whether a μ abstraction is substitutable depends on the polarity of the type T. This corresponds to the rules given in definition 4.5.6, which we call the *polar* evaluation order. The "polar" evaluation strategy is the most natural, since it satisfies the most η-laws.

Definition 4.5.6 (Polar evaluation order)**.**

$$\begin{aligned} \textsc{Subst}\big(\mu(x : T).c\big) &\qquad \text{if } T \text{ has polarity } \mathbf{data} \\ \textsc{Subst}\big(\mu(x \overset{\mathbf{con}}{:} T).c\big) &\qquad \text{if } T \text{ has polarity } \mathbf{codata} \end{aligned}$$

Ideally we would want to prescribe the polar evaluation order in our system, but this doesn't interact nicely with de- and refunctionalization, which we discuss in the next subsection. We therefore need an additional evaluation strategy, the *nominal* strategy given in definition 4.5.7. In the nominal strategy every type declares whether to evaluate its redexes by-value or by-name.

[4]We ignore the difference between call-by-name and call-by-need in this chapter.

Definition 4.5.7 (Nominal evaluation order)**.**

$$\begin{aligned} \textsc{Subst}\big(\mu(x : T).c\big) &\qquad \text{if } T \text{ is a } \textbf{cbv} \text{ type} \\ \textsc{Subst}\big(\mu(x \overset{\textbf{con}}{:} T).c\big) &\qquad \text{if } T \text{ is a } \textbf{cbn} \text{ type} \end{aligned}$$

This nominal evaluation strategy is the one we are ultimately going to adopt.

4.5.3. Extensionality

We now come back to the problem of η equalities mentioned in section 4.2. Assume that the variables bound in the Γ_i do not occur free in the expression e. We define $=_\eta$ as the congruence on terms generated by the following two equations.

$$\textbf{match}_{\textbf{data}}\, T\ \{\overline{\mathcal{X}\Delta} \Rightarrow \langle\, \overline{\mathcal{X}\mathrm{id}_\Gamma} \mid e \,\rangle\} =_\eta e \qquad (\eta_{\textbf{data}})$$

$$\textbf{match}_{\textbf{codata}}\, T\ \{\overline{\mathcal{X}\Delta} \Rightarrow \langle\, e \mid \overline{\mathcal{X}\mathrm{id}_\Gamma} \,\rangle\} =_\eta e \qquad (\eta_{\textbf{codata}})$$

In section 4.2.3 we presented an example which showed that the validity of η equality depends on the evaluation order of the language. For the polar evaluation order as defined above, all η equalities are valid.

Lemma 4.5.8. *Assuming the polar evaluation order from definition 4.5.6, we have for any expression e and e':*

$$(\textsc{Subst}(e) \wedge e =_\eta e') \Rightarrow \textsc{Subst}(e')$$

Since $=_\eta$ is defined as a congruence generated by eq. ($\eta_{\textbf{data}}$) and eq. ($\eta_{\textbf{codata}}$), the following theorem needs the reflexive-transitive closure $\triangleright^*$ of the reduction relation $\triangleright$ since multiple expansions might have occurred within one use of $=_\eta$.

Theorem 4.5.9. *Assuming the polar evaluation order from definition 4.5.6, let c_1, c_1', c_2 be commands s.t. $c_1 \triangleright c_2$ and $c_1 =_\eta c_1'$. Then there exists a command c_2', s.t. $c_2 =_\eta c_2'$ and $c_1' \triangleright^* c_2'$.*

4.6. Changing Evaluation Order

In this section, we will define the missing two transformations which change the evaluation strategy of a type, without changing its polarity (data or codata). That is, we show how to implement the vertical transformations of the diagram introduced in section 4.2.

$$\begin{tikzcd}
\textbf{cbv data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] & \textbf{cbv codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[d, shift left, "\mathcal{T}_{\textbf{cbn}}"] \\
\textbf{cbn data} \arrow[r, shift left, "\mathcal{C}_{\textbf{codata}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"] & \textbf{cbn codata} \arrow[l, shift left, "\mathcal{C}_{\textbf{data}}"] \arrow[u, shift left, "\mathcal{T}_{\textbf{cbv}}"]
\end{tikzcd}$$

The full refunctionalization from call-by-value data to call-by-name codata is then just $\mathcal{C}_{\mathbf{codata}}$, extended to account for μ, followed by $\mathcal{T}_{\mathbf{cbn}}$ (or the other way around, which makes no difference), and similarly for defunctionalization. The $\mathcal{T}_{\mathbf{cbn}}$ and $\mathcal{T}_{\mathbf{cbv}}$ transformations use so-called *shifts*, which we introduce in section 4.6.2, to embed cbn into cbv types or vice versa. When we apply defunctionalization and then refunctionalization (or vice versa), we use $\mathcal{T}_{\mathbf{cbn}}$ after $\mathcal{T}_{\mathbf{cbv}}$ and thus introduce *double-shift* artifacts (thanks to $\mathcal{C}_{\mathbf{data}}$ and $\mathcal{C}_{\mathbf{codata}}$ being mutually inverse, apart from that aspect, we do get back to the original program). However, we show how these artifacts can be removed so that the two transformations are indeed mutually inverse.

4.6.1. De/Refunctionalization and evaluation order

Extending the transformations $\mathcal{C}_p$ of section 4.4 to cover μ abstractions is straightforward: Since evaluation order is not changed in these transformations, we can just keep all μ abstractions within the program unchanged, i.e. the transformation on terms still remains the identity function. Furthermore, the rest of the transformation remains a simple matrix transposition, as before. However, as explained in section 4.2.4, the resulting transformation is unsatisfying on its own, since it does not allow the input and output to both have the transformed type in its polar, i.e. natural evaluation order. Therefore, we introduce *nominal* evaluation order and the $\mathcal{T}_{\mathbf{cbn}}$ and $\mathcal{T}_{\mathbf{cbv}}$ transformations.

Specifically, there are two observations that we can make: Firstly, this extended transformation only preserves typechecking if we keep the evaluation order of the type T constant. Otherwise we might violate the additional premise in the $\textsc{Subst}_2$ rule. Secondly, even if the transformation were type-preserving, it would not be semantics-preserving, since we would now disambiguate the critical pair differently. Thus, the central problem that we tackle in this section is: **We cannot combine the simple de/refunctionalization approach described above with polar evaluation order.**

Previously, user-defined types could be either data or codata types. We now extend this scheme and parameterize user-defined types by their evaluation order, written with prefix **cbv** or **cbn** e.g.

$$\mathbf{cbv\ data\ type}\ \mathbb{N}\ \{\ \texttt{Zero}()\ ;\ \texttt{Suc}(x:\mathbb{N})\ \}.$$

The evaluation order is now induced by this parameter of the type, as formally defined by the two rules for the $\textsc{Subst}(-)$ judgement for *nominal* evaluation order (definition 4.5.7).

As sketched above, we can now factorize the transformations between a **cbv** data type and a **cbn** codata type into two steps, the first of which changes the polarity of the type without changing its evaluation order ($\mathcal{C}_{\mathbf{codata}}$ and $\mathcal{C}_{\mathbf{data}}$ presented in section 4.4, trivially extended to μ). In order to define the second part of this algorithm, we have to introduce shift types, which we do in the next subsection.

4.6.2. Shift Types

The shift types, briefly introduced in section 4.2.5, are ordinary user-defined types in this system; the type system does not need to be extended for them, and the normal evaluation and typing rules presented above apply. We allow ourselves to write the definitions of the shifts parameterized by a type, even though our formalism doesn't allow this. This is not essential; we could alternatively add one shift type to the program for every type that we want to shift. The definitions of the shift types are:

$$\begin{array}{l}\textbf{cbv data type } \uparrow^{\textbf{cbv}} T\ \{\ \texttt{CBV}(x : T)\ \}\\ \textbf{cbn codata type } \uparrow^{\textbf{cbn}} T\ \{\ \texttt{CBN}(x \overset{\textbf{con}}{:} T)\ \}\end{array}$$

As is apparent from their definition, these types do not change the logical meaning of the type they shift. Their effect is restricted to the evaluation order of the program, in every other respect they behave like an identity wrapper. This is why the following helper function $\mathcal{S}^o_s$ allows to embed any producer or consumer expression in the corresponding shifted type:

$$\begin{aligned}\mathcal{S}^{\textbf{prd}}_{\textbf{cbv}}(e) &\coloneqq \texttt{CBV}(e)\\ \mathcal{S}^{\textbf{prd}}_{\textbf{cbn}}(e) &\coloneqq \textbf{match}_{\textbf{codata}} \uparrow^{\textbf{cbn}} T\ \{\texttt{CBN}(x) \Rightarrow \langle\, e \mid x \,\rangle\}\\ \mathcal{S}^{\textbf{con}}_{\textbf{cbv}}(e) &\coloneqq \textbf{match}_{\textbf{data}} \uparrow^{\textbf{cbv}} T\ \{\texttt{CBV}(x) \Rightarrow \langle\, x \mid e \,\rangle\}\\ \mathcal{S}^{\textbf{con}}_{\textbf{cbn}}(e) &\coloneqq \texttt{CBN}(e)\end{aligned}$$

That is, the following rule is derivable for all o, s, e and T:

$$\frac{\Gamma \vdash e \overset{o}{:} T}{\Gamma \vdash \mathcal{S}^o_s(e) \overset{o}{:} \uparrow^s T}$$

We will now give some examples for how shift types can be used. The **cbv** data type $\mathbb{N}$ of natural numbers can be wrapped as $\uparrow^{\textbf{cbn}} \mathbb{N}$ to obtain a codata type of *delayed* natural numbers. A list of ordinary natural numbers cannot contain the unevaluated expression $\mu(k \overset{\textbf{con}}{:} \mathbb{N}).(\langle\, 1 \mid \texttt{add}(1, k) \,\rangle)$, since this expression is not substitutable and can therefore (according to rule T-SUBST$_2$) not occur in a substitution. However, a list of delayed natural numbers can contain the equivalent expression $\textbf{match}_{\textbf{codata}} \uparrow^{\textbf{cbn}} T\ \{\texttt{CBN}(k) \Rightarrow \langle\, 1 \mid \texttt{add}(1, k) \,\rangle\}$. This allows a very fine-grained control over evaluation order in the types. In a program which contains **cbv** data type definitions of natural numbers and lists, as well as a function type, three different kinds of functions which expect a list of natural numbers can be distinguished. A function with argument of type `List` $\mathbb{N}$ evaluates its arguments to a list of fully evaluated natural numbers. If the argument type is `List` $(\uparrow^{\textbf{cbn}} \mathbb{N})$, then the spine of the list has to be fully evaluated before it is substituted in the body of the function, but the elements of the list might still be unevaluated. Thirdly, if an argument of type $\uparrow^{\textbf{cbn}}$ (`List` $\mathbb{N}$) is expected, the argument is passed by-name to the body of the function.

4.6.3. Changing the evaluation order

Let us now define the transformation $\mathcal{T}^T_s$, which changes the evaluation strategy specified for type T to the evaluation strategy s, for the different syntactic entities. For type declarations we specify the transformation in definition 4.6.1.

Definition 4.6.1 (Changing the evaluation order in type declarations)**.**

$$\begin{aligned}\mathcal{T}^T_s(s'\ p\ \textbf{type}\ T'\ \{\overline{\mathcal{X}\,\Delta}\}\ \textbf{with}\ \overline{\mathcal{Y}\Pi := e}) :=&\\ s''\ p\ \textbf{type}\ T'\ \{\overline{\mathcal{X}\ \mathcal{T}^T_s(\Delta)}\}\ \textbf{with}\ &\overline{\mathcal{Y}\mathcal{T}^T_s(\Pi) := \mathcal{T}^T_s(e)}\end{aligned}$$

where s'' is specified in the following way

$$s'' := \begin{cases} s' & (T \neq T') \\ s & (T = T') \end{cases}$$

$\mathcal{T}^T_s$ of course changes the specified evaluation order of T to s, and in all declarations applies $\mathcal{T}^T_s$ to the contexts in the signatures. Note that we assume that the transformation is only applied to applied to types where it makes sense, i.e. $\mathcal{T}^T_{\mathbf{cbv}}$ is only applied if T is a **cbn** type and $\mathcal{T}^T_{\mathbf{cbn}}$ only if T is a **cbv** type. If used this way, note that in the case of $T = T'$, we also have $s = \hat{s'}$.

In contexts $\mathcal{T}^T_s$ replaces T with the relevant shifted type to retain the original evaluation order s. This is specified in definition 4.6.2.

Definition 4.6.2 (Changing the evaluation order in contexts and substitutions)**.**

$$\mathcal{T}^T_s(\diamond) := \diamond \qquad \mathcal{T}^T_s(\Gamma, x \overset{o}{:} T') := \begin{cases} \mathcal{T}^T_s(\Gamma), x \overset{o}{:} T' & (T \neq T') \\ \mathcal{T}^T_s(\Gamma), x \overset{o}{:} \uparrow^{\hat{s}} T & (T = T') \end{cases}$$

$$\mathcal{T}^T_s(()) := () \qquad \mathcal{T}^T_s((\sigma, e) := (\mathcal{T}^T_s(\sigma), \mathcal{T}^T_s(e))$$

Accordingly, in the appropriate places in expressions, whose transformation is specified in definition 4.6.3, the type of a μ has to be changed to the corresponding shifted type, and a **match** or a call to `CBV` or `CBN` has to be inserted to wrap expressions of type T, resulting in the corresponding shifted type. Thus, overall, the evaluation order specified for T is changed while retaining the operational semantics of the program.[5]

Definition 4.6.3 (Changing the evaluation order of expressions and commands)**.** We change the evaluation order of commands as follows:

$$\mathcal{T}^T_s(\textbf{Done}) := \textbf{Done}$$
$$\mathcal{T}^T_s(\langle\, e_1 \mid e_2 \,\rangle) := \langle\, \mathcal{T}^T_s(e_1) \mid \mathcal{T}^T_s(e_2) \,\rangle$$

[5]Up to reduction of administrative redexes caused by the wrapping.

and of expressions as follows:

$$\mathcal{T}_s^T(x) \coloneqq x$$

$$\mathcal{T}_s^T(\mathcal{X}\sigma) \coloneqq \begin{cases} \mathcal{X}\,\mathcal{T}_s^T(\sigma) & \mathcal{X} \notin \text{Funs}(T) \text{ and } \mathcal{X} \notin \text{Xtors}(T) \\ \mathcal{S}_{\hat{s}}^{\mathbf{val}(p)}(\mathcal{X}\ \mathcal{T}_s^T(\sigma)) & \mathcal{X} \in \text{Xtors}(T) \text{ and Polarity}(T) = p \\ \mathcal{S}_{\hat{s}}^{\mathbf{cnt}(p)}(\mathcal{X}\ \mathcal{T}_s^T(\sigma)) & \mathcal{X} \in \text{Funs}(T) \text{ and Polarity}(T) = p \end{cases}$$

$$\mathcal{T}_s^T(\mathbf{match}_p\, T'\, \{\overline{\mathcal{X}\Delta \Rightarrow c}\}) \coloneqq \begin{cases} \mathbf{match}_p\, T'\, \{\overline{\mathcal{X}\Delta \Rightarrow \mathcal{T}_s^T(c)}\} & (T \neq T') \\ \mathcal{S}_{\hat{s}}^{\mathbf{cnt}(p)}(\mathbf{match}_p\, T\, \{\overline{\mathcal{X}\Delta \Rightarrow \mathcal{T}_s^T(c)}\}) & (T = T') \end{cases}$$

$$\mathcal{T}_s^T(\mu(x \overset{o}{:} T').c) \coloneqq \begin{cases} \mu(x \overset{o}{:} T').\mathcal{T}_s^T(c) & (T \neq T') \\ \mu(x \overset{o}{:} \uparrow^{\hat{s}} T).\mathcal{T}_s^T(c) & (T = T') \end{cases}$$

4.6.4. Converting from call-by-value to call-by-name and back: removing double-shifts

When using the transformation twice on a program P w.r.t. a type T, the resulting program, i.e. $\mathcal{T}_{\hat{s}}^T(\mathcal{T}_s^T(P))$, contains double-shift artifacts, which can be removed in order to obtain the original program (with type T replacing type $\uparrow^{\hat{s}}\uparrow^{s} T$). Expressions of type $\uparrow^{\mathbf{cbv}}\uparrow^{\mathbf{cbn}} T$ which contain such artifacts are either

$$\texttt{CBV}(\mathbf{match}_p \uparrow^{\mathbf{cbn}} T\ \{\texttt{CBN}(k) \Rightarrow \langle\, e \mid k \,\rangle\})$$

or

$$\mu(x \overset{\mathbf{con}}{:} \uparrow^{\mathbf{cbv}}\uparrow^{\mathbf{cbn}} T).c.$$

These can respectively be replaced by the result of recursively replacing subexpressions in the expressions e and $\mu(x \overset{\mathbf{con}}{:} T).c$, obtaining expressions of type T. For a type $\uparrow^{\mathbf{cbn}}\uparrow^{\mathbf{cbv}} T$, we proceed analogously. By a simple structural induction, one can show that the just described transformation, which we call $\mathcal{A}^T$, indeed goes from $\mathcal{T}_{\hat{s}}^T(\mathcal{T}_s^T(P))$ to P.

Combining this result with theorem 4.4.2, we have shown that walking from one corner of our diagram to another and back (and eliminating double-shifts) leads back to the original program.

Theorem 4.6.4 (Mutual Inverse (overall))**.** *For every well-formed program P and data type T in P:*

$$\mathcal{A}^T(\mathcal{T}_{\hat{s}}^T(\mathcal{C}_{\hat{p}}^T(\mathcal{T}_s^T(\mathcal{C}_p^T(P))))) = P$$

4.7. Related Work

There are two separate strands of related work: the development of the theory of defunctionalization and refunctionalization on the one hand, and the development of symmetric calculi on the other hand.

Defunctionalization and refunctionalization De/Refunctionalization of the function type have a long history, of which we only cite the seminal papers (John Charles Reynolds, 1972; Danvy and Nielsen, 2001; Danvy and Millikin, 2009). The generalization of defunctionalization from functions to arbitrary codata types was described by Rendel, Trieflinger, and Ostermann (2015) for a simply typed system without local lambda abstractions or local pattern matches. That the defunctionalization of polymorphic functions requires GADTs was first observed by Pottier and Gauthier (2006); that the generalization to data and codata types then also requires GAcoDTs has been described by Ostermann and Jabs (2018). How to treat local pattern and copattern matches in such a way as to preserve invertibility of defunctionalization and refunctionalization has been described by Binder, Jabs, Skupin, and Ostermann (2019), which is the basis of chapter 3. The novel aspect of the present work is that it presents completely symmetric data and codata types, and that it also considers the interaction with evaluation order.

Symmetric calculi Our core calculus was strongly influenced by the ideas of Zeilberger's Calculus of Unity (**CU**) (Zeilberger, 2008b). In contrast to our system, **CU** does not provide direct means for user-defined data types (though a related system does have a similar mechanism (Zeilberger, 2008a)). Furthermore, its treatment of pattern-matches on recursive types is not syntactical: a pattern match on $\mathbb{N}$ contains an infinite amount of cases, similar to the ω-rule (Hilbert, 1931) in formal systems of arithmetic. By contrast, our system (which is hence easily implementable) only allows finite matches, which also have to be shallow anyway, in order to facilitate de- and refunctionalization.

The other important heritage is the ongoing quest for proof-theoretically well-behaved term assignment systems for sequent calculus. The $\bar{\lambda}\mu\tilde{\mu}$ calculus of Curien and Herbelin (2000) is such a system with a fixed set of types; its authors discuss the problems of confluence and evaluation order, but do not consider data and codata types in their general form. The "codata–data" calculus (**CD**) of Downen and Zena M. Ariola (2020) is an extension of $\bar{\lambda}\mu\tilde{\mu}$ with user-defined data and codata types, polymorphism and higher kinds. They discuss extensively the relation between evaluation order and the polarity of types (data or codata), but do not consider algorithms for switching these properties. Their data and codata types are symmetric, but they do not exploit this fact in their formalization; instead they present separate rules for data and codata types.

Another well-known symmetric calculus is the dual calculus of Wadler (2003). This calculus uses the same judgements as our system and $\bar{\lambda}\mu\tilde{\mu}$, and contains both μ and $\tilde{\mu}$ constructs. The types of that system are of ambiguous polarity; for example, the product type is defined by a positive introduction rule (pairing) and negative elimination rules (first and second projection). Wadler also discusses De Morgan duality. He defines a dualizing operation $-^\circ$ which behaves as an involution on both types and terms, and relates proofs of a type T to refutations of T° (in our notation: $\Gamma \vdash t : T \Leftrightarrow \Gamma^\circ \vdash t^\circ \overset{\text{con}}{:} T^\circ$). It is possible to define the same operation in our system. Taking polarity into account, this relates proofs (refutations) of a product data type $\otimes$, i.e. a data type with one "pair" constructor, to refutations (proofs) of sum codata type $⅋$, a codata type with one “case” destructor. The proofs and refutations for the sum data type $\oplus$, a data type with two constructors “inl” and “inr”, and products codata type &, a codata type with two destructors “outl” and “outr”, are similarly related.

4.8. Conclusion

This chapter presented a system with completely symmetric data and codata types. There are two transformations which transform cbv data into cbn codata, and vice versa; this transformation has been factored into one transformation which changes the polarity and one transformation which changes the evaluation order. This has revealed that evaluation order and polarity of a type, even though related, are best treated separately when considering the conversion between data and codata. In particular, codata types have more to offer than just the representation of infinite data, they are an essential ingredient in getting the evaluation strategy of a language right.

5. Introduction and Elimination, Left and Right

The contents of this chapter are based on an article that has been accepted at the international conference on functional programming (ICFP), and the paper has been published as:

> Klaus Ostermann, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen (2022a). "Introduction and Elimination, Left and Right". In: *Proc. ACM Program. Lang.* 6.ICFP. DOI: `10.1145/3547637`. URL: `https://doi.org/10.1145/3547637`

The Coq formalization of the main results of that paper are available online as Ostermann, Binder, Skupin, Süberkrüb, and Downen (2022b).

In this chapter I elaborate the idea that I have already introduced in section 1.2: Different logic calculi correspond to different programming styles. In the introduction we looked at three such correspondences for axiomatic calculi, natural deduction and the sequent calculus. In this chapter we approach the problem more systematically. We start with the observation that there are four different kinds of rules in derivation systems based on sequents: left-introduction rules, right-introduction rules, left-elimination rules and right-elimination rules. For a given logical connective we present *all* of its rules, even though a subset of just the introduction, just the elimination, just the left or just the right rules would be sufficient to prove all logically valid formulas. The completeness of these subcalculi is proved by a principle called *bi-expressibility*: Left-introduction and right-eliminiation rules, as well as left-elimination and right-introduction rules, can be expressed through each other. This encompassing view of different calculi is the first, theoretical contribution of this chapter.

The second contribution of this chapter is our analysis of how the chosen rules determine the resulting programming style. If we choose just the right rules (i.e. right introduction and right elimination rules), then we obtain ordinary functional programs which introduce and eliminate *producers* of a given type. The left rules, on the other hand, allow us to directly manipulate continuations. It might not be obvious that this increased expressiveness can be useful in practice: In order to show that they are useful we present two examples which use the increased expressiveness relative to ordinary functional languages. The first example is about the *parsimonious filter function* that yields the same result as an ordinary filter function which removes all the elements from a list which don't satisfy a given predicate. But the parsimonious filter function can share the tail of the list in which each element satisfies the predicate; the ordinary filter function will always create a copy of this tail in memory. And the parsimonious filter function does all that while traversing the list only once. The second example concerns error handling. The study of linear logic has shown that there are two ways to polarize disjunctions $\vee$: either as a positive data type $\oplus$ or as a negative codata type $\parr$. The data type $\oplus$ is used in programming languages such as Haskell and Rust to handle errors. A function which returns an element of type $\tau_1 \oplus \tau_2$ can return either an element of τ_1 or of τ_2, and the caller of that function has to be able to handle these two possibilities. The codata type $\parr$ offers another way which is closer to how languages like C++ or Java handle errors with exceptions handlers. A function which returns an element of type $\tau_1 \parr \tau_2$ has to be called with two continuations, one for the error case and one for the success case, and the function itself decides which continuation to call.

5.1. Introduction

Undoubtedly, the λ-calculus has had a profound impact on functional programming: from language design, to implementation, to practical programming techniques. Through the Curry-Howard correspondence, we know that the λ-calculus — and likewise, λ-based functional languages — are oriented around the interplay between the *introduction* and *elimination* rules of types as first formulated in natural deduction (ND). This natural deduction style of programming nicely allows for a quite "natural" way of combining sub-problems in programs, like basic operations of function composition $f\ (g\ x)$ and swapping the pair x as $(\mathbf{Snd}\ x, \mathbf{Fst}\ x)$. The natural compositional style is afforded by the fact that all expressions in the λ-calculus *produce* exactly one result that is implicitly taken by *exactly one* consumer: namely, the enclosing context of that expression. While the single implicit consumer is natural for composition, it can be rather *unnatural* if we ever want to work with more than one consumer. In these cases, we must reify the implicit consumer to make it explicit — such as resorting to continuation-passing style John Charles Reynolds, 1972 or control operators like `call/cc` — which leads to the rather asymmetric situation of having a mix of one special implicit output with additional explicit outputs.

When might functional programmers want to juggle multiple consumers at the same time? Consider the familiar filter function, naturally expressible in any typed functional language:

$$\begin{aligned}
&\mathit{filter} && : (a \to \mathit{Bool}) \to [a] \to [a]\\
&\mathit{filter}\ p\ [] && = []\\
&\mathit{filter}\ p\ (x :: xs) && = \mathbf{If}\ p\ x\ \mathbf{Then}\ x :: \mathit{filter}\ p\ xs\\
& && \qquad\qquad \mathbf{Else}\ \mathit{filter}\ p\ xs
\end{aligned}$$

This function successfully removes any elements from a given list xs which fail the test p, leaving only those elements x for which $(p\ x)$ is true. From a quick inspection of the definition, we can see that the output of this function always fits its specification. So what's wrong? The problem is with *efficiency.* It's quite likely that a large chunk of the filtered output will be the same as the input, and yet *filter* will reallocate a new cons cell for every (::) in the output, even if that same list already exists in memory. As an extreme case, if we filter with the constant function $\mathit{const}\ x\ y = x$, the call *filter* (*const True*) *xs* will allocate and return an entirely new list that is equal to *xs*.

Instead, we would rather that a call like *filter* (*const True*) *xs* notices its output will be equal to *xs*, so that it can return the *exact same* object x that was already allocated in the heap. In a more general case, it may be that only some suffix of *xs* all passes the test p; if so, *filter p xs* should return a list that only allocates new cons cells for the prefix of the list that is changed, connected to an identical pointer to the same suffix of the original *xs* list in the heap. And in any case, *filter p xs* should only traverse the list *xs* once, and when it has reached the end of the list, it should return immediately and not have to unwind a call-stack first, so it would need to be written in a partially tail-recursive way. This is quite a tall order to satisfy in a conventional functional language. Shivers and Fisher (2004) showed how to express efficient filtering using *multi-return functions*. However, this solution complicates ordinary first-class functions with another concern (multiple different return paths); going against orthogonal language design philosophy of keeping separate features separate.

The classical sequent calculus (SC) can also be interpreted Curry-Howard-style as a prototypical programming language. Sequent-style languages turn several implicit aspects of the λ-calculus into explicit, symmetric entities. There is a context containing many named outputs, dual to the many named inputs in the context of free variables. Besides terms which primarily produce results, there are terms which primarily consume inputs. Elimination rules are replaced by left rules operating on consumers. Computation happens inside of a *command* $\langle e \mid f \rangle$, which connects the output of a producer e with the input of a consumer f. For instance, the consumer $\mathbf{Fst} \fatsemi f$ should be read as "project the implicit pair to its first component and then continue with f" and is reduced as $\langle (e_1, e_2) \mid \mathbf{Fst} \fatsemi f \rangle \triangleright \langle e_1 \mid f \rangle$. Compared to ND, programs in SC have an "inside-out" structure, which Wadler (2003) has compared to the external plumbing of the Pompidou center in Paris. Yet, whereas gazing upon the Pompidou center may be a beautiful sight, sifting through the bureaucratic plumbing of a large-scale sequent-style program is not.

Still, the use of left rules lets the classical sequent calculus express *new kinds of types* which are not expressible in conventional functional languages. These new, exotic types would let us decompose a complex feature like multi-return functions into more basic parts. But must we always suffer through painful bureaucracy to access these types for programming? No! Our insight is that we can combine the best of both natural deduction (introduction versus elimination) and sequent calculus (left versus right) styles in the same program, making use of exotic new types of control flow that juggle multiple consumers while still enjoying pleasantly natural, functional composition.

We propose having four styles of rules: introductions *and* eliminations on both the left *and* the right. Doing so lets us introduce new ways of programming that keeps all the familiar features we already know as they are, and *also* lets us talk about new types that can't be expressed in conventional functional languages. For example, we can decompose the idea of multi-return functions (Shivers and Fisher, 2004) into several

$$
\begin{aligned}
&(\circ) : b \prec \neg(a \to b) \prec a \\
&\langle\, x \mid \alpha \circ \mathbf{Not}\ f \,\rangle = \langle\, f\ x \mid \alpha \,\rangle \\
\\
&\mathit{filter} : (a \to \mathit{Bool}) \to [a] \to [a] \\
&\langle\, \mathit{filter}\ p\ xs \mid \mathit{start} \,\rangle = \\
&\quad \langle\, \mathbf{Handle}\ (\mathit{filter/pass}\ p\ xs)\ \mathbf{with}\ (\mathit{start} \circ \mathbf{Not}\ (\mathit{const}\ xs)) \mid \mathit{start} \,\rangle \\
\\
&\mathit{filter/pass} : (a \to \mathit{Bool}) \to [a] \to [a] \parr () \\
&\langle\, \mathit{filter/pass}\ p\ [] \mid [\mathit{diff}, \mathit{same}] \,\rangle = \langle\, () \mid \mathit{same} \,\rangle \\
&\langle\, \mathit{filter/pass}\ p\ (x :: xs) \mid [\mathit{diff}, \mathit{same}] \,\rangle = \\
&\quad \mathbf{If}\ p\ x \\
&\quad \mathbf{Then}\ \langle\, \mathit{filter/pass}\ p\ xs \mid [\mathit{diff} \circ \mathbf{Not}\ (x ::), \mathit{same}] \,\rangle \\
&\quad \mathbf{Else}\ \ \langle\, \mathit{filter/pass}\ p\ xs \mid [\mathit{diff}, \mathit{diff} \circ \mathbf{Not}\ (\mathit{const}\ xs)] \,\rangle
\end{aligned}
$$

Figure 5.1.: Parsimonious filter function

orthogonal features: regular functions (of type $T_1 \to T_2$), computations juggling two continuations (of type $T_1 \parr T_2$), functions transforming a consumer of T_1s to a consumer of T_2s (of type $T_1 \prec T_2$), and the reversal between producers and consumers (of type $\neg T$).

Together, these features let us write the efficient[1] filtering function using familiar programming concepts (first-class functions and exception handling) and reusable combinators (like *const* above and the following composition $\circ$ of a function with a continuation) based on a function *filter/pass* which either filters a list by removing at least one failing element, or finds element passes (fig. 5.1). The rest of this chapter will go into the detail needed to read, understand, and specify how this example works.

So, should we program in ND style or in SC style? We say: both! The purpose of this work is to analyze and complete the logic calculus design space opened up by ND and SC and investigate the interdependency between logical calculus style and program structure. More specifically, we make the following contributions:

- We investigate the usage of all four kinds of rules, introduction and elimination, both left and right. We identify four different sub-calculi: The *intro calculus* (which corresponds to classical SC), the *right calculus* (which corresponds to ND), the *elimination calculus* (which has only left and right elimination rules), and the *left calculus* (which features left introduction and elimination rules).

[1] We are defining a calculus via a reduction semantics, and that semantics does not feature pointers, so we cannot directly talk about sharing and do not make any practical efficiency claims here. We will clarify that point in section 5.7.

- We analyze the influence of these calculi on program structure (and hence modularity, extensibility etc.). Specifically, we argue that the common programming design guideline “the structure of the program follows the structure of the data” can be interpreted in four ways, and that those four ways correspond to the four calculi described above.

- We clarify the relation between the different rules of a connective by the concept of *bi-expressibility*. Informally, bi-expressibility means that the left introduction rule is as powerful as the right elimination rule and the right introduction rule is as powerful as the left elimination rule.

- We deepen the known dualities between connectives by using the *proof/refutation duality* Tranchini, 2012. Specifically, we show that the typing, term-level representation, and reduction of “dual” connectives can be derived mechanically, which gives rise to the possibility of a new form of *consumer/producer polymorphism*, in which a term can be interpreted as both a producer of type T or as a consumer of the dual of T.

All theorems of this chapter have been mechanized and proven in Coq (submitted as supplementary material).

The remainder of this chapter is structured as follows: In section 5.2, we give an informal introduction to the language framework we present and describe the influence of rule choice on program structure. In section 5.3, we present a small core language featuring functions as well as positive sums and products. In section 5.4, we introduce bi-expressibility and present an operational semantics of the language. In section 5.5, we demonstrate how to mechanically derive the dual connectives (cofunctions, negative sums and products) and elaborate on the idea of duality polymorphism. In section 5.6 we describe extensions of the calculus framework with logical constants and universal and existential types. In section 5.7.2 we present examples to illustrate the new possibilities of programming in a language with all four kinds of rules. Section 5.8 presents related and future work and section 5.9 concludes.

5.2. Motivation

To get started, let’s analyze the interaction between logical structure and program structure. As an example, consider the connective $\oplus$ as an algebraic data type in a typical functional language. Values of the type $X \oplus Y$ are built using the constructors $\mathbf{In}_1$ and $\mathbf{In}_2$ — these correspond nicely to the two introduction rules of $X \oplus Y$ in natural deduction. To use values of type $X \oplus Y$, we can employ a **Case**-expression that pattern-matches on the two possible alternatives — likewise corresponding exactly to natural deduction’s elimination rule for $X \oplus Y$. Using these tools, we can swap these two options — transforming an unknown value $z : X \oplus Y$ into a result of $Y \oplus X$ — by matching on the originally-chosen option and replacing it with the opposite constructor like so:

$$\begin{aligned}\mathbf{Case}\ z\ \{\ &\mathbf{In}_1\ x \mapsto \mathbf{In}_2\ x;\\ &\mathbf{In}_2\ y \mapsto \mathbf{In}_1\ y\ \}\end{aligned}$$

Rather than always thinking of producing some (implicit) output using introductions and eliminations, a sequent-based language does away with eliminations altogether. Instead, it uses the dichotomy between *producers* which implicitly return some result as an output — just like we are used to in conventional functional programming — versus *consumers* which implicitly take an input to use analogous to continuations. The same swapping operation — converting an unknown $z : X \oplus Y$ into a $Y \oplus X$ — can be written in sequent style as:

$$\begin{aligned}\langle z \mid \mathbf{Match}\ \{\ &\mathbf{In}_1\ x \mapsto \langle\, \mathbf{In}_2\ x \mid \alpha\,\rangle\\ &\mathbf{In}_2\ y \mapsto \langle\, \mathbf{In}_1\ y \mid \alpha\,\rangle\}\rangle\end{aligned}$$

Notice the several differences in this version of the same program. The top-most operation is a *command* of the form $\langle\, e \mid f\,\rangle$ which connects the implicit output returned from a producer e into the implicit input expected by a consumer f. Unlike producers and consumers, commands have no implicit input or output. Rather than the **Case** expression — which has an explicit input used to implicitly produce some result — we use a **Match**, which forms a new consumer implicitly expecting an input of type $X \oplus Y$. The **Match** consumer has two branches pattern-matching on its implicit input — one for each possibility between $\mathbf{In}_1$ and $\mathbf{In}_2$ — and because consumers have no implicit output, both possible branches lead to commands. In either case, the swapped sum value is explicitly "returned" (that is, passed via a command) to α, which gives a name to the previously-implicit output of the whole operation.

Under our analysis of comparing different structures of logic — and their impact on their corresponding programs — natural deduction has only *right* rules. In other words, every rule of natural deduction is concerned with concluding the *truth* of propositions (traditionally written on the right-hand side of the hypothetical turnstyle $\vdash$, hence the name "right"). By the proofs-as-programs paradigm, this corresponds to the fact that every primitive tool that typical functional languages have for working with its various types of information — both introductions and eliminations — are inherently concerned with *producing* something. The constructors $\mathbf{In}_1$ and $\mathbf{In}_2$ *produce* unique values of the sum type $X \oplus Y$. The **Case** expression uses an $X \oplus Y$ value, yes, but only in the service of *producing* some other result (which may not necessarily be another sum type). In contrast, the application of the sequent calculus as the basis for a programming language has revealed a different way of organizing programs. Instead, it pairs rules working on the *right* (which look just like natural deduction's introduction rules) with rules working on the *left*. These left rules can be seen as ways to *refute* propositions, which are concerned with the *falsehood* of propositions (or equivalently, with *assumed truth* of propositions, traditionally written on the left-hand side of $\vdash$, hence the name "left"). But importantly, no matter which side of the divide the rules are focused, the sequent calculus has only *introduction* rules

Table 5.1.: Four different ways to swap the components of $z : X \oplus Y$ and send to consumer $\alpha : Y \oplus X$.

Calculus	Program
Right	$\mathbf{Case}\ z\ \{\mathbf{In}_1\ x \mapsto \langle\, \mathbf{In}_2\ x \mid \alpha\,\rangle; \mathbf{In}_2\ y \mapsto \langle\, \mathbf{In}_1\ y \mid \alpha\,\rangle\}$
Intro	$\langle z \mid \mathbf{Match}\ \{\mathbf{In}_1\ x \mapsto \langle\, \mathbf{In}_2\ x \mid \alpha\,\rangle; \mathbf{In}_2\ y \mapsto \langle\, \mathbf{In}_1\ y \mid \alpha\,\rangle\}\rangle$
Left	$\langle z \mid \mathbf{Match}\ \{\mathbf{In}_1\ x \mapsto \langle\, x \mid \mathbf{Out}_2\ \alpha\,\rangle; \mathbf{In}_2\ y \mapsto \langle\, y \mid \mathbf{Out}_1\ \alpha\,\rangle\}\rangle$
Elim	$\mathbf{Case}\ z\ \{\mathbf{In}_1\ x \mapsto \langle\, x \mid \mathbf{Out}_2\ \alpha\,\rangle; \mathbf{In}_2\ y \mapsto \langle\, y \mid \mathbf{Out}_1\ \alpha\,\rangle\}$

We can compare these two logical styles of programming — one based on natural deduction and the other on the sequent calculus — by modifying the elimination rule slightly. When in the context of a command with an explicit consumer named α, a **Case** does not need to implicitly produce some result, but can instead indicate what command to execute next by sending either result to α. In effect, this "absorbs" the consumer α into the **Case** like so:

$$\begin{aligned}&\langle\, \mathbf{Case}\ e\ \{\mathbf{In}_1\ x \mapsto e_1;\ \mathbf{In}_2\ y \mapsto e_2\} \mid \alpha\,\rangle = \\ &\quad \mathbf{Case}\ e\ \{\mathbf{In}_1\ x \mapsto \langle\, e_1 \mid \alpha\,\rangle; \mathbf{In}_2\ y \mapsto \langle\, e_2 \mid \alpha\,\rangle\}\end{aligned}$$

With this in mind, we present the two styles of sum-swapping in table 5.1, in each case connecting an input $z : X \oplus Y$ to an output $\alpha : Y \oplus X$. The first line uses only right rules (both introduction and eliminations on the right) in a typical functional style. The next line illustrates how the same program can be written using only introduction rules (both on the left and the right) in sequent style. The gray parts indicate how we can connect the program to an explicitly given producer z.

But what about other combinations of rules? If we can make due with only right rules or only introductions, can we also write the same program using only left rules and only eliminations? Yes! The third line shows again the same program but this time using only left rules. $\mathbf{Out}_1\ \alpha$ and $\mathbf{Out}_2\ \alpha$ are the left elimination rules for a consumer α of type $Y \oplus X$. $\mathbf{Out}_1\ \alpha$ should be read as: Inject the implicit value into the left component of a sum and then continue with α. Finally, the last line shows how the program can be formulated using only elimination rules. As we will later see, every program can be written in each of these styles, and there is a simple and systematic way to transition between them.

But what difference does it make which style we choose?

One of the main principles of programming that is taught in most introductory programming courses (such as Felleisen, Findler, Flatt, and Krishnamurthi (2001)) is that the structure of a program follows the structure of its input. A less common but equally valid principle is that the structure of a program follows the structure of its output Gibbons, 2021. Both styles can also be reversed in that a program may also be "inside-out", that is, it follows the structure of the input or output from the inside to the outside and not vice versa. Such a structure is common, for instance, in continuation-passing style. The choice of style has a major influence on the modularity properties of the program: How easily it can be read and understood, how extensible it is, and so forth.

Table 5.2.: Computation from $x : (\top \,\&\, X) \,\&\, \top$ to $\alpha : \bot \oplus ((X \oplus \bot) \oplus \bot)$.

Calculus	Program	Program Structure
Right	$\langle \mathbf{In}_2\ (\mathbf{In}_1\ (\mathbf{In}_1\ (\mathbf{Out}_2\ (\mathbf{Out}_1\ x)))\ \mid \alpha \rangle$	α outside-in, x inside-out
Intro	$\langle x \mid \mathbf{In}_1\ (\mathbf{In}_2\ \tilde{\mu}x.\langle \mathbf{In}_2\ (\mathbf{In}_1\ (\mathbf{In}_1\ x)) \mid \alpha \rangle) \rangle$	x outside-in, α outside-in
Left	$\langle x \mid \mathbf{In}_1\ (\mathbf{In}_2\ (\mathbf{Out}_1\ (\mathbf{Out}_1\ (\mathbf{Out}_2\ \alpha)))) \rangle$	x outside-in, α inside-out
Elim	$\langle \mathbf{Out}_2\ (\mathbf{Out}_1\ x) \mid \mathbf{Out}_1\ (\mathbf{Out}_1\ (\mathbf{Out}_2\ \alpha)) \rangle$	α inside-out, x inside-out

The observation that motivated this work is that the choice of rules determines which of these styles our programs will naturally have. We illustrate this in table 5.2, which shows four ways of projecting out of a nested product type $(\top \,\&\, X) \,\&\, \top$ and injecting into a nested sum type $\bot \oplus ((X \oplus \bot) \oplus \bot)$. Our interest here is in four different calculi corresponding exactly to the four different program styles illustrated in table 5.2.

The Intro variant uses Curien and Herbelin (2000)'s $\tilde{\mu}$ operator to reify the currently consumed value as a variable, which brings us to the last difference between the calculi that we want to point out. Producer expressions have explicit inputs (given by variables) and an implicit output (the current continuation). Consumer expressions have explicit outputs (given by covariables) and an implicit input. Whenever we need to construct programs where the flow of the respective implicit input/output does not match the nesting structure of the program, we need to resort to μ or $\tilde{\mu}$ operators to reify that implicit input/output.

Here is another example: With right elimination rules, the composition of two functions g and h is easy to express: $\langle h\ (g\ x) \mid \alpha \rangle$. However, when we have to express the same program using the left introduction rule for functions instead (which constructs a pair $e \cdot f$ consisting of a function argument e and a consumer f for the function result), we have to express the program as $\langle g \mid x \cdot (\tilde{\mu}y.\langle h \mid y \cdot \alpha \rangle) \rangle$. The "intermediate result" $g\ x$ must be given a name y because the flow of data does not correspond to the structure of the left introduction rule.

Last but not least, let's consider how modularity and extensibility depend on the program style we choose. We analyze the nesting structure of the term in relation to the nesting structure of the type. Informally, outside-in means that the term is nested similar to the type: subterms of the type correspond to subterms of the term. Inside-out means that inner nodes of the term structure represent outer nodes of the type structure. In general, introduction rules yield terms nesting outside-in because the introduced type appears in the conclusion of the rule, while elimination rules induce an inside-out nesting as the eliminated type occurs in the premise of the rule.

When we use elimination forms on the left to introduce a type on the right or vice-versa we therefore reverse the nesting structure of the program and thereby also alter its modularity properties.

Consider the programs in table 5.2 again. In the right calculus row, we destruct x inside-out with right elimination rules and then construct a producer with the type required by α outside-in using right introduction rules. The situation is reversed in the left calculus row. Here we destruct the continuation α inside-out with left elimination rules and construct a continuation with the type required by x outside-in using left introduction rules. Similarly, we can get any other combination of nesting orders by choosing one of the other calculi.

To summarize, having a choice of all four kinds of rules makes it easier for the programmer to choose a program structure that has the desired modularity and extensibility properties and maximizes the usage of implicit producers/consumers to avoid the naming of intermediate results and the associated CPS-like program structure.

5.3. Introduction versus Elimination, Left versus Right

In this and the next section, we present our language framework, divided into logical steps and parts. As the first step, we present a core language of inputs, outputs, and interactions, without any logical connectives[2] (Figure 5.2).

As usual in presentations of computational sequent calculi, we have three kinds of sequents: $\Gamma \vdash e : T$ describes a producer term e that produces an output of type T in variable context Γ. Symmetrically, $\Gamma \vdash f \overset{\text{con}}{:} T$ describes a consumer term f that consumes an input of type T. Finally, a command $\Gamma \vdash c$ describes a complete and executable program. The core language only contains cut commands $\langle e \mid f \rangle$.

With regard to reduction, we opt to use the standard call-by-value evaluation strategy, which involves prioritizing producers before consumers in cuts Downen and Zena M. Ariola, 2018a. Alternatively, we could have chosen call-by-name evaluation in the standard way by reversing this priority Wadler, 2003. The purpose of the focusing contexts $\mathcal{E}$ and $\mathcal{F}$ (which are empty in the core language) and the $\triangleright_{\varsigma}$ reduction rules are to push pending computations embedded in subterms to the top-level. This is also standard Wadler, 2003; Downen and Zena M. Ariola, 2018a.

In fig. 5.3 we extend the language with three connectives: $\rightarrow$ (functions), $\oplus$ (positive sums), and $\otimes$ (positive products). The adjective "positive" comes from polarized type theory Andreoli, 1992; Zeilberger, 2009 and denotes data types that are defined via constructors, as opposed to negative types, which are codata types that are defined via destructors. Positive types are evaluated eagerly, when constructed, whereas negative types are evaluated on demand, when they are destructed. In this chapter, we use blue for positive and red for negative connectives.

All connectives come with all four kinds of rules: introduction and elimination, both left and right. Like in linear logic, we make a clear distinction between positive and negative connectives (the latter will be shown later); it turns out that for our bi-expressibility property (introduced in section 5.4) it is essential that rules do not duplicate or destroy information. That property also necessitates a few generalizations of standard rules.

For instance, the syntax of λ abstraction is $\lambda(x{\cdot}\alpha).c$ rather than $\lambda x.e$; the latter can be encoded as $\lambda x.e := \lambda(x{\cdot}\alpha).\langle e \mid \alpha \rangle$. The left introduction rule for $\rightarrow$ consists of a function argument and a consumer for the function's returned result. The left elimination rule allows us to inspect both components of that pair, yielding a command.

[2]The core is similar to the presentation in Sec. 4 of Downen and Zena M. Ariola (2018a)

Syntax

$$\begin{array}{lcll}
T & ::= & X & \textit{Types} \\[4pt]
e & ::= & x \mid \mu\alpha.c & \textit{Producers} \\
f & ::= & \alpha \mid \tilde{\mu}x.c & \textit{Consumers} \\
c & ::= & \langle\, e \mid f \,\rangle & \textit{Commands} \\
v & ::= & x & \textit{Values} \\[4pt]
\mathcal{E}[] & ::= & \epsilon & \textit{Focusing Context} \\
\mathcal{F}[] & ::= & \epsilon & \textit{Cofocusing Context} \\[4pt]
\Gamma & ::= & x : T, \Gamma \mid \alpha \overset{\textbf{con}}{:} : T, \Gamma \mid \epsilon & \textit{Context}
\end{array}$$

Typing rules

$$\dfrac{x : T \ \in \Gamma}{\Gamma \vdash x : T}\ (\text{R-}\textsc{Var}) \qquad \dfrac{\alpha \overset{\textbf{con}}{:} T \ \in \Gamma}{\Gamma \vdash \alpha \overset{\textbf{con}}{:} T}\ (\text{L-}\textsc{Var})$$

$$\dfrac{\Gamma, \alpha \overset{\textbf{con}}{:} T \vdash c}{\Gamma \vdash \mu\alpha.c : T}\ (\text{R-}\mu) \qquad \dfrac{\Gamma, x : T \vdash c}{\Gamma \vdash \tilde{\mu}x.c \overset{\textbf{con}}{:} T}\ (\text{L-}\mu) \qquad \dfrac{\Gamma \vdash e : T \quad \Gamma \vdash f \overset{\textbf{con}}{:} T}{\Gamma \vdash \langle\, e \mid f \,\rangle}\ (\textsc{Cut})$$

Reduction

$$\begin{array}{lll}
\langle\, v \mid \tilde{\mu}x.c \,\rangle & \triangleright_{\tilde{\mu}} & c\{x := v\} \\
\langle\, \mu\alpha.c \mid f \,\rangle & \triangleright_{\mu} & c\{\alpha := f\}
\end{array}$$

In ς rules, x is fresh, $e \notin v$.

$$\begin{array}{lll}
\langle\, \mathcal{E}[e] \mid f \,\rangle & \triangleright_{\varsigma} & \langle\, e \mid \tilde{\mu}x.\langle\, \mathcal{E}[x] \mid f \,\rangle \,\rangle \\
\langle\, v \mid \mathcal{F}[e] \,\rangle & \triangleright_{\varsigma} & \langle\, e \mid \tilde{\mu}x.\langle\, v \mid \mathcal{F}[x] \,\rangle \,\rangle
\end{array}$$

Figure 5.2.: Core language

Syntax

$$\begin{array}{llll}
T & ::= & \ldots & \mid T \rightarrow T \mid T \oplus T \mid T \otimes T \\
e & ::= & \ldots & \mid \lambda(x \cdot \alpha).c \mid e\ e \mid \mathbf{In}_i\ e \mid [e, e] \\
f & ::= & \ldots & \mid e \cdot f \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \mid \mathbf{Out}_i\ f \\
 & & & \mid \mathbf{Match}\ \{[x, x] \mapsto c\} \mid \mathbf{Handle}_i\ f\ \mathbf{with}\ e \\
c & ::= & \ldots & \mid \mathbf{Case}\ f\ \{x \cdot \alpha \mapsto c\} \mid \mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \mid \mathbf{Case}\ e\ \{[x, x] \mapsto c\} \\
v & ::= & \ldots & \mid \lambda(x \cdot \alpha).c \mid \mathbf{In}_i\ v \mid [v, v] \\
\mathcal{E}[] & ::= & \ldots & \mid [\square, e] \mid [v, \square] \mid \mathbf{In}_i\ \square \\
\mathcal{F}[] & ::= & \ldots & \mid \square \cdot f
\end{array}$$

Typing

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash f \overset{\text{con}}{:} T_2}{\Gamma \vdash e \cdot f \overset{\text{con}}{:} T_1 \rightarrow T_2}\ (\text{L-}\rightarrow\text{-Intro}) \qquad \frac{\Gamma, x : T_1, \alpha \overset{\text{con}}{:} T_2 \vdash c}{\Gamma \vdash \lambda(x \cdot \alpha).c : T_1 \rightarrow T_2}\ (\text{R-}\rightarrow\text{-Intro})$$

$$\frac{\Gamma \vdash f \overset{\text{con}}{:} T_1 \rightarrow T_2 \qquad \Gamma, x : T_1, \alpha \overset{\text{con}}{:} T_2 \vdash c}{\Gamma \vdash \mathbf{Case}\ f\ \{x \cdot \alpha \mapsto c\}}\ (\text{L-}\rightarrow\text{-Elim}) \qquad \frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ (\text{R-}\rightarrow\text{-Elim})$$

$$\frac{\forall i, \Gamma, x_i : T_i \vdash c_i}{\Gamma \vdash \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \overset{\text{con}}{:} T_1 \oplus T_2}\ (\text{L-}\oplus\text{-Intro}) \qquad \frac{\Gamma \vdash e : T_i}{\Gamma \vdash \mathbf{In}_i\ e : T_1 \oplus T_2}\ (\text{R-}\oplus\text{-Intro}_i)$$

$$\frac{\Gamma \vdash f \overset{\text{con}}{:} T_1 \oplus T_2}{\Gamma \vdash \mathbf{Out}_i\ f \overset{\text{con}}{:} T_i}\ (\text{L-}\oplus\text{-Elim}_i) \qquad \frac{\Gamma \vdash e : T_1 \oplus T_2 \qquad \forall i, \Gamma, x_i : T_i \vdash c_i}{\Gamma \vdash \mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\}}\ (\text{R-}\oplus\text{-Elim})$$

$$\frac{\Gamma, x : T_1, y : T_2 \vdash c}{\Gamma \vdash \mathbf{Match}\ \{[x, y] \mapsto c\} \overset{\text{con}}{:} T_1 \otimes T_2}\ (\text{L-}\otimes\text{-Intro}) \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash [e_1, e_2] : T_1 \otimes T_2}\ (\text{R-}\otimes\text{-Intro})$$

$$\frac{\Gamma \vdash f \overset{\text{con}}{:} T_1 \otimes T_2 \qquad \Gamma \vdash e : T_i}{\Gamma \vdash \mathbf{Handle}_i\ f\ \mathbf{with}\ e \overset{\text{con}}{:} T_{2-i+1}}\ (\text{L-}\otimes\text{-Elim}_i) \qquad \frac{\Gamma \vdash e : T_1 \otimes T_2 \qquad \Gamma, x : T_1, y : T_2 \vdash c}{\Gamma \vdash \mathbf{Case}\ e\ \{[x, y] \mapsto c\}}\ (\text{R-}\otimes\text{-Elim})$$

Reduction

$$\begin{array}{lll}
\langle\, \lambda(x \cdot \alpha).c \mid v \cdot f \,\rangle & \triangleright_\beta & c\{x := v, \alpha := f\} \\
\langle\, \mathbf{In}_j\ v \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \,\rangle & \triangleright_\beta & c_j\{x_j := v\} \\
\langle\, [v_1, v_2] \mid \mathbf{Match}\ \{[x, y] \mapsto c\} \,\rangle & \triangleright_\beta & c\{x := v_1, y := v_2\}
\end{array}$$

Figure 5.3.: Syntax, typing and reduction for functions, positive sums, and positive products

The rule R-$\oplus$-Intro$_i$ is standard. R-$\oplus$-Elim uses again commands in its branches and is in fact itself a command. An encoding for the more standard **Case** e $\{\overline{\mathbf{In}_i\ x_i \mapsto e_i}\}$ can be given as $\mu\alpha.($**Case** e $\{\overline{\mathbf{In}_i\ x_i \mapsto \langle\, e_i \mid \alpha\,\rangle}\})$. The reason for the deviation from the standard is symmetry with the L-$\oplus$-Intro rule, which has the same structure except that the value to be pattern-matched on is implicit. L-$\oplus$-Elim$_i$ has also been designed to be symmetric to R-$\oplus$-Intro$_i$.

The rules for $\otimes$ follow the same design principles. Of particular note are the L-$\otimes$-Elim$_i$ rules. Their names **Handle**$_i$ are motivated by $\parr$ (the dual of $\otimes$), whose right elimination rules are reminiscent of error handlers as shown in fig. 5.1.

The reduction rules cover only the introduction forms of the constructs. We will see in the next section why that is sufficient.

This language and all its extensions we are about to present, has four subcalculi identified by considering the languages having only introduction rules (the Intro calculus), only elimination rules (the Elim calculus), only right rules (the Right calculus), or only left rules (the Left calculus).

We have proven (mechanized in Coq) standard type safety theorems for this language, but before we can talk about the details we need to introduce bi-expressibility.

5.4. Bi-Expressibility and Soundness

Our language has been designed in such a way that all four sub-calculi are in a sense equally powerful. This is made precise in fig. 5.4, which shows that each typing rule is derivable using only core constructs and the diagonally opposing rule (left intro is diagonally opposing to right elim and vice versa left elim to right intro):

Theorem 5.4.1 (Bi-Expressibility)**.** *Every typing rule of a connective can be encoded using the diagonally opposing rule and core constructs only.*

Proof. Simple inspection and type-checking of the encoding rules in fig. 5.4. □

Corollary 5.4.2 (Subcalculus Restriction)**.** Given any well-typed command $\Gamma \vdash c$, producer $\Gamma \vdash e : T$, or consumer $\Gamma \vdash f \overset{\text{con}}{:} T$ and any subcalculus $\mathfrak{C} \in \{\text{Intro}, \text{Elim}, \text{Left}, \text{Right}\}$, there exists a translation result $\Gamma \vdash c'$, $\Gamma \vdash e' : T$, or $\Gamma \vdash f' \overset{\text{con}}{:} T$ which uses only the syntax available in subcalculus $\mathfrak{C}$.

Proof. We give a translation function (formalized in Coq) that restricts the command, producer or consumer to the subcalculus $\mathfrak{C}$ based on theorem 5.4.1 and show that it preserves type soundness. □

One thing to note about the encodings is that, when applied to a full program, they will introduce administrative redexes. For instance, when applying the encodings to the examples in section 5.2, then the encoding of the right elimination construct by left introduction gives us directly the second line, whereas applying the encoding of right introduction by left elimination gives as an administrative redex of the form

$$\frac{\begin{array}{c}\Gamma \vdash e : T_1 \\ \Gamma \vdash f \overset{\textbf{con}}{:} T_2\end{array}}{\Gamma \vdash \tilde{\mu} x.\langle\, x\ e \mid f \,\rangle \overset{\textbf{con}}{:} T_1 \rightarrow T_2}\ (\text{L-}{\rightarrow}\text{-}\textsc{Intro})$$

$$\frac{\Gamma, x : T_1, \alpha \overset{\textbf{con}}{:} T_2 \vdash c}{\Gamma \vdash \mu\beta.\textbf{Case}\ \beta\ \{x \cdot \alpha \mapsto c\} : T_1 \rightarrow T_2}\ (\text{R-}{\rightarrow}\text{-}\textsc{Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\textbf{con}}{:} T_1 \rightarrow T_2 \\ \Gamma, x : T_1, \alpha \overset{\textbf{con}}{:} T_2 \vdash c\end{array}}{\Gamma \vdash \langle\, \lambda(x \cdot \alpha).c \mid f \,\rangle}\ (\text{L-}{\rightarrow}\text{-}\textsc{Elim})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \rightarrow T_2 \\ \Gamma \vdash e_2 : T_1\end{array}}{\Gamma \vdash \mu\alpha.\langle\, e_1 \mid e_2 \cdot \alpha \,\rangle : T_2}\ (\text{R-}{\rightarrow}\text{-}\textsc{Elim})$$

$$\frac{\forall i, \Gamma, x_i : T_i \vdash c_i}{\Gamma \vdash \tilde{\mu} x.\ \textbf{Case}\ x\ \{\overline{\textbf{In}_i\ x_i \mapsto c_i}\} \overset{\textbf{con}}{:} T_1 \oplus T_2}\ (\text{L-}{\oplus}\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash e : T_i}{\Gamma \vdash \mu\alpha.\ \langle\, e \mid \textbf{Out}_i\ \alpha \,\rangle : T_1 \oplus T_2}\ (\text{R-}{\oplus}\text{-}\textsc{Intro}_i)$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} T_1 \oplus T_2}{\Gamma \vdash \tilde{\mu} x.\ \langle\, \textbf{In}_i\ x \mid f \,\rangle \overset{\textbf{con}}{:} T_i}\ (\text{L-}{\oplus}\text{-}\textsc{Elim}_i)$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_1 \oplus T_2 \\ \forall i, \Gamma, x_i : T_i \vdash c_i\end{array}}{\Gamma \vdash \langle\, e \mid \textbf{Match}\ \{\overline{\textbf{In}_i\ x_i \mapsto c_i}\} \,\rangle}\ (\text{R-}{\oplus}\text{-}\textsc{Elim})$$

$$\frac{\Gamma, x : T_1, y : T_2 \vdash c}{\Gamma \vdash \tilde{\mu} z.\textbf{Case}\ z\ \{[x, y] \mapsto c\} \overset{\textbf{con}}{:} T_1 \otimes T_2}\ (\text{L-}{\otimes}\text{-}\textsc{Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \\ \Gamma \vdash e_2 : T_2\end{array}}{\Gamma \vdash \mu\alpha.\langle\, e_1 \mid \textbf{Handle}_2\ \alpha\ \textbf{with}\ e_2 \,\rangle : T_1 \otimes T_2}\ (\text{R-}{\otimes}\text{-}\textsc{Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\textbf{con}}{:} T_1 \otimes T_2 \\ \Gamma \vdash e : T_1\end{array}}{\Gamma \vdash \tilde{\mu} x.\langle\, [e, x] \mid f \,\rangle \overset{\textbf{con}}{:} T_2}\ (\text{L-}{\otimes}\text{-}\textsc{Elim}_1)$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\textbf{con}}{:} T_1 \otimes T_2 \\ \Gamma \vdash e : T_2\end{array}}{\Gamma \vdash \tilde{\mu} x.\langle\, [x, e] \mid f \,\rangle \overset{\textbf{con}}{:} T_1}\ (\text{L-}{\otimes}\text{-}\textsc{Elim}_2)$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_1 \otimes T_2 \\ \Gamma, x : T_1, y : T_2 \vdash c\end{array}}{\Gamma \vdash \langle\, e \mid \textbf{Match}\{[x, y] \mapsto c\} \,\rangle}\ (\text{R-}{\otimes}\text{-}\textsc{Elim})$$

Figure 5.4.: Bi-Expressibility: Diagonal encodings. New (co)variable names are always assumed to be fresh.

$\langle \mu\beta.\langle x \mid \mathbf{Out}_2\ \beta \rangle \mid \alpha \rangle$, which requires a reduction step to $\langle x \mid \mathbf{Out}_2\ \alpha \rangle$ to yield the third line.

Another thing to note is that the notion of value depends on the subcalculus. Since we only define reduction for the Intro calculus, our value definition is tailored for it. A term like a λ-abstraction is a value in the Intro calculus; when it gets encoded into the Left or Elim calculus by the encoding rule, it turns into a μ-abstraction - which looks superficially as if we would turn a value into a non-value. But this is misleading, since each calculus would have its own definition of value, if we would define them completely separately. Instead, we use bi-expressibility to give reduction rules for the introduction constructs only: We assume that the elimination rules are "desugared" using the encodings in fig. 5.4.

For the language where the elimination forms are encoded as in fig. 5.4, we have also proven (in Coq) standard progress and preservation theorems.

Theorem 5.4.3 (Preservation)**.** *For all commands c and all typing contexts Γ, if $\Gamma \vdash c$ and $c \triangleright c'$, then $\Gamma \vdash c'$.*

For the statement of the progress theorem, it is convenient to have a **Done** command with typing axiom $\mathbf{Done} : \Gamma \vdash \Delta$, such that there are non-trivial closed commands:

Theorem 5.4.4 (Progress)**.** *For all closed commands c and all typing contexts Γ, if $\Gamma \vdash c$, then either $c = \mathbf{Done}$ or there exists c' such that $c \triangleright c'$.*

Furthermore, reduction is deterministic:

Theorem 5.4.5 (Deterministic reduction)**.** *For all commands c, c', c'' and all typing contexts Γ, if $\Gamma \vdash c$, $c \triangleright c'$, and $c \triangleright c''$, then $c' = c''$.*

Let us briefly illustrate why bi-expressibility requires some generalizations of introduction or elimination forms of standard constructs; in particular, why we have replaced expressions by commands in some places. As an example, the standard right introduction rule for $\rightarrow$ is $\lambda x.e$, whereas we use $\lambda(x \cdot \alpha).c$. If we look at R-$\rightarrow$-Intro in fig. 5.4, then $\lambda x.e$ could be encoded as $\mu\beta.\mathbf{Case}\ \beta\ \{x \cdot \alpha \mapsto \langle e \mid \alpha \rangle\}$. However, L-$\rightarrow$-Elim would then be stronger than R-$\rightarrow$-Intro and the L-$\rightarrow$-Elim encoding in fig. 5.4 would no longer work; if we wanted to reverse the transformation, we cannot guarantee that e does not have α as free variable. Weakening L-$\rightarrow$-Elim to $\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto f\}$ instead of $\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto c\}$ would not help; both constructs would be weaker, but they would not be bi-expressible. $\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto e\}$, on the other hand, would be rather useless in the left calculus, since the only valid (non-μ) producer expression is a variable.

5.5. Proof/Refutation Duality

Our setting with all four kinds of rules lets us use the proof/refutation duality Tranchini, 2012. The proof/refutation duality is a duality between proofs of a statement and refutations of the dual statement. Tranchini (2012) defined a natural deduction calculus of refutation which is completely isomorphic to the standard natural deduction calculus of proofs. For instance, the introduction and elimination rules for refutations of disjunctions (written $\vee_R$), are identical to the ones for proofs of conjunctions — another facet of the duality between disjunction and conjunction:

$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 \vee_R T_2}\ (\vee_R\text{-}\textsc{Intro}) \qquad\qquad \frac{\Gamma \vdash T_1 \vee_R T_2}{\Gamma \vdash T_i}\ (\vee_R\text{-}\textsc{Elim}_i)$$

The introduction rule should be read as “if T_1 is false and T_2 is false, then $T_1 \vee T_2$ is false”, the elimination rules as “if $T_1 \vee T_2$ is false, then T_i is false”. Tranchini has shown that a full (intuitionistic) refutation calculus containing all standard propositional connectives can be defined in such a way that the rules are mirrors of the respective dual connectives in the proof calculus. The duality between disjunction and conjunction as well as between the logical constants $\top$ and $\bot$ are standard; dualizing implication requires the less common notion of co-implication Tranchini, 2012, also known as subtraction Crolard, 2004 or difference Curien and Herbelin, 2000 (sometimes with reversed order of arguments).

Tranchini’s work is purely in the logic domain and does not discuss programming, but from the perspective of programming, a refutation calculus can be seen as a language of consumers or continuations. The close symmetry between the corresponding proof and refutation calculi suggests that the term language of the proof calculus can also be used as a term language of the refutation calculus. In other words, we can have different interpretations of the same term: once as a producer and once as a consumer.

Figure 5.5 illustrates the idea by defining typing rules for $\prec$, $\&$, and $⅋$, the duals of $\rightarrow$, $\oplus$, and $\otimes$, respectively. What is noteworthy about these rules is that they are completely determined by the typing rules of their respective duals. In fact, the rules in fig. 5.5 could be replaced by fusing the syntactic categories e and f,

$$\begin{array}{lll}
e & ::= & x \mid \mu x.c \mid \tilde{\mu} x.c \mid e \cdot e \mid \textbf{Match}\ \{\overline{\textbf{In}_i\ x_i \mapsto c_i}\} \mid \textbf{Out}_i\ e \mid \textbf{Match}\ \{[x,x] \mapsto c\} \\
 & & \mid \textbf{Handle}_i\ e\ \textbf{with}\ e \mid \lambda(x \cdot z).c \mid e\ e \mid \textbf{In}_i\ e \mid [e,e] \\
x, \alpha & ::= & \textit{identifier}
\end{array}$$

and adding these two rules

$$\frac{\Gamma^\circ \vdash e^\circ \overset{\textbf{con}}{:} T^\circ}{\Gamma \vdash e : T} \qquad\qquad \frac{\Gamma^\circ \vdash e^\circ : T^\circ}{\Gamma \vdash e \overset{\textbf{con}}{:} T}$$

where T° is defined as

$$\begin{array}{lllllll}
X^\circ & = & X & & & \\
(T_1 \rightarrow T_2)^\circ & = & T_1^\circ \prec T_2^\circ & (T_1 \prec T_2)^\circ & = & T_1^\circ \rightarrow T_2^\circ \\
(T_1 \otimes T_2)^\circ & = & T_1^\circ ⅋ T_2^\circ & (T_1 ⅋ T_2)^\circ & = & T_1^\circ \otimes T_2^\circ \\
(T_1 \& T_2)^\circ & = & T_1^\circ \oplus T_2^\circ & (T_1 \oplus T_2)^\circ & = & T_1^\circ \& T_2^\circ\ ,
\end{array}$$

Γ° is the obvious extension to typing contexts, and e° creates a copy of the expression that is identical except that $\langle\, e_1 \mid e_2 \,\rangle^\circ = \langle\, e_2^\circ \mid e_1^\circ \,\rangle$ where e contains a command.

A fully shared and symmetric term syntax between producers and consumers would enable an exciting new feature we call *consumer/producer polymorphism*. It becomes possible to have libraries of code that can be used as both producers and consumers.

Syntax

$$
\begin{array}{llll}
T & ::= & \ldots & \mid T \prec T \mid T \mathbin{\&} T \mid T \parr T \\
e & ::= & \ldots & \mid f \cdot e \mid \textbf{Match } \{\overline{\textbf{In}_i\ x_i \mapsto c_i}\} \mid \textbf{Out}_i\ e \\
 & & & \mid \textbf{Match } \{[x, x] \mapsto c\} \mid \textbf{Handle}_i\ e \textbf{ with } f \\
f & ::= & \ldots & \mid \lambda(\alpha \cdot x).c \mid f\ f \mid \textbf{In}_i\ f \mid [f, f] \\
c & ::= & \ldots & \mid \textbf{Case } e\ \{\alpha \cdot x \mapsto c\} \mid \textbf{Case } f\ \{\overline{\textbf{In}_i\ \alpha_i \mapsto c_i}\} \mid \textbf{Case } f\ \{[\alpha, \alpha] \mapsto c\} \\
v & ::= & \ldots & \mid f \cdot v \mid \textbf{Match } \{\overline{\textbf{In}_i\ \alpha_i \mapsto c_i}\} \mid \textbf{Match } \{[\alpha, \alpha] \mapsto c\} \\
\mathcal{E}[] & ::= & \ldots & \mid f \cdot \square
\end{array}
$$

Typing

$$\frac{\Gamma, x : T_2, \alpha \overset{\text{con}}{:} T_1 \vdash c}{\Gamma \vdash \lambda(\alpha \cdot x).c \overset{\text{con}}{:} T_1 \prec T_2}\ (\text{L-}\prec\text{-Intro}) \qquad \frac{\begin{array}{c}\Gamma \vdash f \overset{\text{con}}{:} T_1 \\ \Gamma \vdash e : T_2\end{array}}{\Gamma \vdash f \cdot e : T_1 \prec T_2}\ (\text{R-}\prec\text{-Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash f_1 \overset{\text{con}}{:} T_1 \prec T_2 \\ \Gamma \vdash f_2 \overset{\text{con}}{:} T_1\end{array}}{\Gamma \vdash f_1\ f_2 \overset{\text{con}}{:} T_2}\ (\text{L-}\prec\text{-Elim}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : T_1 \prec T_2 \\ \Gamma, x : T_1, \alpha \overset{\text{con}}{:} T_2 \vdash c\end{array}}{\Gamma \vdash \textbf{Case } e\ \{\alpha \cdot x \mapsto c\}}\ (\text{R-}\prec\text{-Elim})$$

$$\frac{\Gamma \vdash f \overset{\text{con}}{:} T_i}{\Gamma \vdash \textbf{In}_i\ f \overset{\text{con}}{:} T_1 \mathbin{\&} T_2}\ (\text{L-}\&\text{-Intro}_i) \qquad \frac{\forall i, \Gamma, \alpha_i \overset{\text{con}}{:} T_i \vdash c_i}{\Gamma \vdash \textbf{Match } \{\overline{\textbf{In}_i\ \alpha_i \mapsto c_i}\} : T_1 \mathbin{\&} T_2}\ (\text{R-}\&\text{-Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\text{con}}{:} T_1 \mathbin{\&} T_2 \\ \forall i, \Gamma, \alpha_i : T_i \vdash c_i\end{array}}{\Gamma \vdash \textbf{Case } f\ \{\overline{\textbf{In}_i\ \alpha_i \mapsto c_i}\}}\ (\text{L-}\&\text{-Elim}) \qquad \frac{\Gamma \vdash e : T_1 \mathbin{\&} T_2}{\Gamma \vdash \textbf{Out}_i\ e : T_i}\ (\text{R-}\&\text{-Elim}_i)$$

$$\frac{\begin{array}{c}\Gamma \vdash f_1 \overset{\text{con}}{:} T_1 \\ \Gamma \vdash f_2 \overset{\text{con}}{:} T_2\end{array}}{\Gamma \vdash [f_1, f_2] \overset{\text{con}}{:} T_1 \parr T_2}\ (\text{L-}\parr\text{-Intro}) \qquad \frac{\Gamma, \alpha \overset{\text{con}}{:} T_1, \beta \overset{\text{con}}{:} T_2 \vdash c}{\Gamma \vdash \textbf{Match } \{[\alpha, \beta] \mapsto c\} : T_1 \parr T_2}\ (\text{R-}\parr\text{-Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\text{con}}{:} T_1 \parr T_2 \\ \Gamma, \alpha \overset{\text{con}}{:} T_1, \beta \overset{\text{con}}{:} T_2 \vdash c\end{array}}{\Gamma \vdash \textbf{Case } f\ \{[\alpha, \beta] \mapsto c\}}\ (\text{L-}\parr\text{-Elim}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : T_1 \parr T_2 \\ \Gamma \vdash f \overset{\text{con}}{:} T_i\end{array}}{\Gamma \vdash \textbf{Handle}_i\ e \textbf{ with } f : T_{2-i+1}}\ (\text{R-}\parr\text{-Elim}_i)$$

Reduction

$$
\begin{array}{lll}
\langle f \cdot v \mid \lambda(\alpha \cdot x).c \rangle & \triangleright_\beta & c\{x := v, \alpha := f\} \\
\langle \textbf{Match } \{\overline{\textbf{In}_i\ \alpha_i \mapsto c_i}\} \mid \textbf{In}_j\ f \rangle & \triangleright_\beta & c_j\{\alpha_j := f\} \\
\langle \textbf{Match } \{[\alpha, \beta] \mapsto c\} \mid [f_1, f_2] \rangle & \triangleright_\beta & c\{\alpha := f_1, \alpha := f_2\}
\end{array}
$$

Figure 5.5.: Dual rules for $\prec$, $\&$, and $\parr$

For instance, λ-abstraction is the common way to abstract over certain code patterns, and that is true regardless of whether the code consumes or produces values. A generic function like the one for function composition, $\lambda f.\lambda g.\lambda x.f\ (g\ x)$, which desugars to

$$comp = \lambda(f \cdot \alpha).\langle\, \lambda(g \cdot \beta).\langle\, \lambda(x \cdot \gamma).\langle\, f\ (g\ x) \mid \gamma \,\rangle \mid \beta \,\rangle \mid \alpha \,\rangle$$

can be used both as a producer of type $(Y \rightarrow Z) \rightarrow (X \rightarrow Y) \rightarrow X \rightarrow Z$ and, via

$$comp^{\circ} = \lambda(f \cdot \alpha).\langle\, \alpha \mid \lambda(g \cdot \beta).\langle\, \beta \mid \lambda(x \cdot \gamma).\langle\, \gamma \mid f\ (g\ x) \,\rangle\,\rangle\,\rangle$$

as a consumer of type $(Y \prec Z) \prec (X \prec Y) \prec X \prec Z$. Similarly, a swap function of type $(X \otimes Y) \rightarrow (Y \otimes X)$ can also be used as a cofunction of type $(X \parr Y) \prec (Y \parr X)$. Both are equally useful. Every program can be used in two ways - an exciting avenue that we intend to explore more in future work.

In the design above, e° is not yet completely identical to e. In an earlier design, we phrased the calculus in such a way that the dualization operation on expressions is indeed the identity function (by having statements where the producer/consumer side switch depending on whether one abstracts over a μ or a $\tilde{\mu}$), but this made the presentation more complicated in other ways. But even without this, code could be reused in the form of "macro" transformations or with a dedicated language construct (say, a $dual(e)$ construct) where the interpreter takes care of switching the sides when necessary. We leave the elaboration of these ideas to future work.

To summarize, the proof/refutation duality allows us to mechanically derive the syntax, typing and reduction rules for the respective dual connective, and it opens a path towards the reuse of a term as both a producer of a type and a consumer of its dual type.

5.6. Extensions

In this section, we consider some standard extensions of the calculi in the new light of having all four rules and bi-expressibility.

The first extension we consider are the logical constants, which serve as units for products and sums. Since we have two differently polarized products and two differently polarized sums, we consequently need 4 different units whose rules are given in fig. 5.6. Following the standard notation from linear logic, the unit for $\otimes$ is written 1, the unit for $\oplus$ is 0, the unit for $\&$ is $\top$, and the unit for $\parr$ is $\bot$. 1 is dual to $\bot$ and 0 is dual to $\top$, hence both the syntax and the typing rules of the respective dual connective follows mechanically like described in the previous section.

One of the exciting insights of linear logic is that the function type $\rightarrow$ can be decomposed into a combination of a negative sum $\parr$ and negation type. The decomposition of both negative functions $\rightarrow$ and positive cofunctions $\prec$ requires two different kinds of negations; negative negations $\neg$ and positive negations $\sim$. The rules for both kinds of negations, which are dual to each other, are given in fig. 5.7. With negation in place, we can very directly see that the type $T_1 \rightarrow T_2$ is isomorphic to $\neg T_1 \parr T_2$:

Syntax

$$\begin{array}{lcl}
T & ::= & \ldots \mid 1 \mid 0 \mid \top \mid \bot \\
e & ::= & \ldots \mid \textbf{triv} \mid \textbf{Case}\ e\ \{\} \mid \textbf{Match}\ \{\textbf{triv} \mapsto c\} \mid \textbf{Match}\ \{\} \\
f & ::= & \ldots \mid \textbf{triv} \mid \textbf{Case}\ f\ \{\} \mid \textbf{Match}\ \{\textbf{triv} \mapsto c\} \mid \textbf{Match}\ \{\} \\
c & ::= & \ldots \mid \textbf{UnTriv}\ f \mid \textbf{Case}\ e\ \{\textbf{triv} \mapsto c\} \mid \textbf{UnTriv}\ e \mid \textbf{Case}\ f\ \{\textbf{triv} \mapsto c\} \\
v & ::= & \ldots \mid \textbf{triv} \mid \textbf{Match}\ \{\textbf{triv} \mapsto c\}
\end{array}$$

Typing

$$\frac{\Gamma \vdash c}{\Gamma \vdash \textbf{Match}\ \{\textbf{triv} \mapsto c\} \overset{\textbf{con}}{:} 1}\ (\text{L-}1\text{-}\textsc{Intro}) \qquad \frac{}{\Gamma \vdash \textbf{triv} : 1}\ (\text{R-}1\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} 1}{\Gamma \vdash \textbf{UnTriv}\ f}\ (\text{L-}1\text{-}\textsc{Elim}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : 1 \\ \Gamma \vdash c\end{array}}{\Gamma \vdash \textbf{Case}\ e\ \{\textbf{triv} \mapsto c\}}\ (\text{R-}1\text{-}\textsc{Elim})$$

$$\frac{}{\Gamma \vdash \textbf{Match}\ \{\} \overset{\textbf{con}}{:} 0}\ (\text{L-}0\text{-}\textsc{Intro})$$

no right introduction rule for 0

no left elimination rule for 0

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \textbf{Case}\ e\ \{\}}\ (\text{R-}0\text{-}\textsc{Elim})$$

no left introduction rule for $\top$

$$\frac{}{\Gamma \vdash \textbf{Match}\ \{\} : \top}\ (\text{R-}\top\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} \top}{\Gamma \vdash \textbf{Case}\ f\ \{\}}\ (\text{L-}\top\text{-}\textsc{Elim})$$

no right elimination rule for $\top$

$$\frac{}{\Gamma \vdash \textbf{triv} \overset{\textbf{con}}{:} \bot}\ (\text{L-}\bot\text{-}\textsc{Intro}) \qquad \frac{\Gamma \vdash c}{\Gamma \vdash \textbf{Match}\ \{\textbf{triv} \mapsto c\} : \bot}\ (\text{R-}\bot\text{-}\textsc{Intro})$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\textbf{con}}{:} \bot \\ \Gamma \vdash c\end{array}}{\Gamma \vdash \textbf{Case}\ f\ \{\textbf{triv} \mapsto c\}}\ (\text{L-}\bot\text{-}\textsc{Elim}) \qquad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash \textbf{UnTriv}\ e}\ (\text{R-}\bot\text{-}\textsc{Elim})$$

Reduction (*no rules for* $\top$ *and* 0)

$$\langle \textbf{triv} \mid \textbf{Match}\ \{\textbf{triv} \mapsto c\} \rangle \rhd_\beta c \qquad \langle \textbf{Match}\ \{\textbf{triv} \mapsto c\} \mid \textbf{triv} \rangle \rhd_\beta c$$

Bi-Expressibility (*rules for* $\top$ *and* $\bot$ *are analogous to those for* 1 *and* 0)

$$\frac{\Gamma \vdash c}{\Gamma \vdash \tilde{\mu}x.\textbf{Case}\ x\ \{\textbf{triv} \mapsto c\} \overset{\textbf{con}}{:} 1}\ (\text{L-}1\text{-}\textsc{Intro}) \qquad \frac{}{\Gamma \vdash \mu\alpha.\textbf{UnTriv}\ \alpha : 1}\ (\text{R-}1\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} 1}{\Gamma \vdash \langle \textbf{triv} \mid f \rangle}\ (\text{L-}1\text{-}\textsc{Elim}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : 1 \\ \Gamma \vdash c\end{array}}{\Gamma \vdash \langle e \mid \textbf{Match}\ \{\textbf{triv} \mapsto c\} \rangle}\ (\text{R-}1\text{-}\textsc{Elim})$$

$$\frac{}{\Gamma \vdash \tilde{\mu}x.\textbf{Case}\ x\ \{\} \overset{\textbf{con}}{:} 0}\ (\text{L-}0\text{-}\textsc{Intro})$$

no right introduction rule for 0

no left elimination rule for 0

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \langle e \mid \textbf{Match}\ \{\} \rangle}\ (\text{R-}0\text{-}\textsc{Elim})$$

Figure 5.6.: Positive units 1, 0 and negative units $\top$, $\bot$

Syntax

$$
\begin{array}{lcl}
T & ::= & \ldots \mid \neg T \mid {\sim} T \\
e & ::= & \ldots \mid \textbf{Not}\ f \mid \textbf{Throw}\ f\ f \mid \textbf{Match}\ \{\textbf{Not}\ x \mapsto c\} \mid \textbf{Case}\ e\ \{\textbf{Not}\ \alpha \mapsto c\} \\
f & ::= & \ldots \mid \textbf{Not}\ e \mid \textbf{Throw}\ e\ e \mid \textbf{Match}\ \{\textbf{Not}\ \alpha \mapsto c\} \mid \textbf{Case}\ f\ \{\textbf{Not}\ x \mapsto c\} \\
v & ::= & \ldots \mid \textbf{Not}\ f \\
\mathcal{F}[] & ::= & \ldots \mid \textbf{Not}\ \square
\end{array}
$$

Typing

$$
\frac{\Gamma, \alpha \overset{\text{con}}{:} T \vdash c}{\Gamma \vdash \textbf{Match}\ \{\textbf{Not}\ \alpha \mapsto c\} \overset{\text{con}}{:} {\sim} T}\ (\text{L-}{\sim}\text{-}\textsc{Intro})
\qquad
\frac{\Gamma \vdash f \overset{\text{con}}{:} T}{\Gamma \vdash \textbf{Not}\ f : {\sim} T}\ (\text{R-}{\sim}\text{-}\textsc{Intro})
$$

$$
\frac{\begin{array}{c}\Gamma \vdash f_1 \overset{\text{con}}{:} {\sim} T \\ \Gamma \vdash f_2 \overset{\text{con}}{:} T\end{array}}{\Gamma \vdash \textbf{Throw}\ f_1\ f_2}\ (\text{L-}{\sim}\text{-}\textsc{Elim})
\qquad
\frac{\begin{array}{c}\Gamma \vdash e : {\sim} T \\ \Gamma, \alpha \overset{\text{con}}{:} T \vdash c\end{array}}{\Gamma \vdash \textbf{Case}\ e\ \{\textbf{Not}\ \alpha \mapsto c\}}\ (\text{R-}{\sim}\text{-}\textsc{Elim})
$$

$$
\frac{\Gamma \vdash e : T}{\Gamma \vdash \textbf{Not}\ e \overset{\text{con}}{:} \neg T}\ (\text{L-}\neg\text{-}\textsc{Intro})
\qquad
\frac{\Gamma, x : T \vdash c}{\Gamma \vdash \textbf{Match}\ \{\textbf{Not}\ x \mapsto c\} : \neg T}\ (\text{R-}\neg\text{-}\textsc{Intro})
$$

$$
\frac{\begin{array}{c}\Gamma \vdash f \overset{\text{con}}{:} \neg T \\ \Gamma, x : T \vdash c\end{array}}{\Gamma \vdash \textbf{Case}\ f\ \{\textbf{Not}\ x \mapsto c\}}\ (\text{L-}\neg\text{-}\textsc{Elim})
\qquad
\frac{\begin{array}{c}\Gamma \vdash e_1 : \neg T \\ \Gamma \vdash e_2 : T\end{array}}{\Gamma \vdash \textbf{Throw}\ e_1\ e_2}\ (\text{R-}\neg\text{-}\textsc{Elim})
$$

Reduction

$$
\begin{array}{lcl}
\langle \textbf{Not}\ f \mid \textbf{Match}\ \{\textbf{Not}\ \alpha \mapsto c\} \rangle & \rhd_\beta & c\{\alpha \coloneqq f\} \\
\langle \textbf{Match}\ \{\textbf{Not}\ x \mapsto c\} \mid \textbf{Not}\ v \rangle & \rhd_\beta & c\{x \coloneqq v\}
\end{array}
$$

Bi-Expressibility

$$
\frac{\Gamma, \alpha \overset{\text{con}}{:} T \vdash c}{\Gamma \vdash \mu x.\textbf{Case}\ x\ \{\textbf{Not}\ \alpha \mapsto c\} \overset{\text{con}}{:} {\sim} T}\ (\text{L-}{\sim}\text{-}\textsc{Intro})
\qquad
\frac{\Gamma \vdash f \overset{\text{con}}{:} T}{\Gamma \vdash \tilde{\mu}\alpha.\textbf{Throw}\ \alpha\ f : {\sim} T}\ (\text{R-}{\sim}\text{-}\textsc{Intro})
$$

$$
\frac{\begin{array}{c}\Gamma \vdash f_1 \overset{\text{con}}{:} {\sim} T \\ \Gamma \vdash f_2 \overset{\text{con}}{:} T\end{array}}{\Gamma \vdash \langle \textbf{Not}\ f_2 \mid f_1 \rangle}\ (\text{L-}{\sim}\text{-}\textsc{Elim})
\qquad
\frac{\begin{array}{c}\Gamma \vdash e : {\sim} T \\ \Gamma, \alpha \overset{\text{con}}{:} T \vdash c\end{array}}{\Gamma \vdash \langle e \mid \textbf{Match}\ \{\textbf{Not}\ \alpha \mapsto c\} \rangle}\ (\text{R-}{\sim}\text{-}\textsc{Elim})
$$

$$
\frac{\Gamma \vdash e : T}{\Gamma \vdash \mu x.\textbf{Throw}\ x\ e \overset{\text{con}}{:} \neg T}\ (\text{L-}\neg\text{-}\textsc{Intro})
\qquad
\frac{\Gamma, x : T \vdash c}{\Gamma \vdash \tilde{\mu}\alpha.\textbf{Case}\ \alpha\ \{\textbf{Not}\ x \mapsto c\} : \neg T}\ (\text{R-}\neg\text{-}\textsc{Intro})
$$

$$
\frac{\begin{array}{c}\Gamma \vdash f \overset{\text{con}}{:} \neg T \\ \Gamma, x : T \vdash c\end{array}}{\Gamma \vdash \langle \textbf{Match}\ \{\textbf{Not}\ x \mapsto c\} \mid f \rangle}\ (\text{L-}\neg\text{-}\textsc{Elim})
\qquad
\frac{\begin{array}{c}\Gamma \vdash e_1 : \neg T \\ \Gamma \vdash e_2 : T\end{array}}{\Gamma \vdash \langle e_1 \mid \textbf{Not}\ e_2 \rangle}\ (\text{R-}\neg\text{-}\textsc{Elim})
$$

Figure 5.7.: Extension with negation

Syntax

$$
\begin{array}{lll}
T & ::= & \ldots \mid \forall X.T \mid \exists X.T \\
e & ::= & \ldots \mid \Lambda\{X,\alpha\}.c \mid e\ [T] \mid \{T,e\} \\
f & ::= & \ldots \mid \Lambda\{X,x\}.c \mid f\ [T] \mid \{T,f\} \\
c & ::= & \ldots \mid \textbf{Case}\ f\ \{X,\alpha \mapsto c\} \mid \textbf{Case}\ e\ \{X,x \mapsto c\} \\
v & ::= & \ldots \mid \Lambda\{X,\alpha\}.c \mid \{T,v\} \\
\mathcal{E}[] & ::= & \ldots \mid \{T,\Box\}
\end{array}
$$

Typing

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} T_2\{X := T_1\}}{\Gamma \vdash \{T_1,f\} \overset{\textbf{con}}{:} \forall X.T_2}\ (\text{L-}\forall\text{-}\textsc{Intro}) \qquad \frac{\Gamma, X, \alpha \overset{\textbf{con}}{:} T \vdash c \quad X \notin FV(\Gamma)}{\Gamma \vdash \Lambda\{X,\alpha\}.c : \forall X.T}\ (\text{R-}\forall\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} \forall X.T \quad \Gamma, X, \alpha \overset{\textbf{con}}{:} T \vdash c}{\Gamma \vdash \textbf{Case}\ f\ \{X,\alpha \mapsto c\}}\ (\text{L-}\forall\text{-}\textsc{Elim}) \qquad \frac{\Gamma \vdash e : \forall X.T}{\Gamma \vdash e\ [T_1] : T\{X := T_1\}}\ (\text{R-}\forall\text{-}\textsc{Elim})$$

$$\frac{\Gamma, X, x : T \vdash c \quad X \notin FV(\Gamma)}{\Gamma \vdash \Lambda\{X,x\}.c \overset{\textbf{con}}{:} \exists X.T}\ (\text{L-}\exists\text{-}\textsc{Intro}) \qquad \frac{\Gamma \vdash e \overset{\textbf{con}}{:} T_2\{X := T_1\}}{\Gamma \vdash \{T_1,e\} : \exists X.T_2}\ (\text{R-}\exists\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} \exists X.T}{\Gamma \vdash f\ [T_1] \overset{\textbf{con}}{:} T\{X := T_1\}}\ (\text{L-}\exists\text{-}\textsc{Elim}) \qquad \frac{\Gamma \vdash e : \exists X.T \quad \Gamma, X, x : T \vdash c}{\Gamma \vdash \textbf{Case}\ e\ \{X,x \mapsto c\}}\ (\text{R-}\exists\text{-}\textsc{Elim})$$

Reduction

$$
\begin{array}{rl}
\langle\, \Lambda\{X,\alpha\}.c \mid \{T,f\}\,\rangle \rhd_\beta & c\{X := T, \alpha := f\} \\
\langle\, \{T,v\} \mid \Lambda\{X,x\}.c\,\rangle \rhd_\beta & c\{X := T, x := v\}
\end{array}
$$

Bi-Expressibility (*Rules for* $\exists$ *are analogous to those for* $\forall$)

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} T_2\{X := T_1\}}{\Gamma \vdash \tilde{\mu}x.\langle\, x\ [T_1] \mid f\,\rangle \overset{\textbf{con}}{:} \forall X.T_2}\ (\text{L-}\forall\text{-}\textsc{Intro}) \qquad \frac{\Gamma, X, \alpha \overset{\textbf{con}}{:} T \vdash c \quad X \notin FV(\Gamma)}{\Gamma \vdash \mu\beta.\textbf{Case}\ \beta\ \{X,\alpha \mapsto c\} : \forall X.T}\ (\text{R-}\forall\text{-}\textsc{Intro})$$

$$\frac{\Gamma \vdash f \overset{\textbf{con}}{:} \forall X.T_1 \quad \Gamma, X, \alpha \overset{\textbf{con}}{:} T_1 \vdash c}{\Gamma \vdash \langle\, \Lambda\{X,\alpha\}.c \mid f\,\rangle}\ (\text{L-}\forall\text{-}\textsc{Elim}) \qquad \frac{\Gamma \vdash e : \forall X.T}{\Gamma \vdash \mu\alpha.\langle\, e \mid \{T_1,\alpha\}\,\rangle : T\{X := T_1\}}\ (\text{R-}\forall\text{-}\textsc{Elim})$$

Figure 5.8.: Extension with universal and existential types.

$$\begin{array}{lcl}
\lambda(x \cdot \beta).c & = & \textbf{Match } \{[\alpha, \beta] \mapsto \textbf{Case } \alpha \ \{\textbf{Not } x \mapsto c\}\} \\
e_1 \ e_2 & = & \textbf{Handle}_1 \ e_1 \ \textbf{with } (\textbf{Not } e_2) \\
e \cdot f & = & [\textbf{Not } e, f] \\
\textbf{Case } f \ \{x \cdot \beta \mapsto c\} & = & \textbf{Case } f \ \{[\alpha, \beta] \mapsto \textbf{Case } \alpha \ \{\textbf{Not } x \mapsto c\}\}
\end{array}$$

And, by duality, the same holds for $T_1 \prec T_2$ and $\sim T_1 \otimes T_2$. It is also not hard to see that **Not** $e : \neg T$ is isomorphic to $e \cdot \textbf{triv} : T \to \bot$ and **Not** $f : \sim T$ is isomorphic to $f \cdot \textbf{triv} : T \prec 0$.

With regard to universal and existential types, whose rules are given in fig. 5.8, it is pleasing to see that the left elimination rule for universal types is basically the usual right elimination rule for "opening an existential package", and the left elimination rule of existential types corresponds to the usual type application right elimination rule for universal types. Of course, $\forall$ is dual to $\exists$, and the duality is again reflected directly in the typing rules. Bi-expressibility follows the same structure as bi-expressibility for functions.

5.7. Programming with all Rules

In this section, we present a few examples that illustrate the utility of polarized connectives and the flexibility of having all rules. We also revisit the filter example from section 5.1.

5.7.1. Error Handling

The first example serves to illustrate both the value of having first-class consumers, and the naturality of programs using rules from all calculi. Consider the familiar problem of error-handling. Suppose we have three functions $f : A \to B \vee E$, $g : B \to C \vee E$ and $h : C \to D \vee E$. These functions take an argument (of type A, B or C) and either return a result (of type B, C or D) or return an error E. The problem is how to compose these functions in such a way that $h \circ g \circ f$ is a function of type $A \to D \vee E$.

Solving this problem crucially depends on how the type $\vee$ is represented. In our language, we have two choices: Positive ("data type", evaluated when constructed) disjunction $\oplus$ or negative ("codata type", evaluated when destructed) disjunction $⅋$. We will now discuss both alternatives in turn.[3]

Error Handling Using a Positive Type

When we choose to model $\vee$ as a positive type $\oplus$, we have constructors $\textbf{In}_1$ and $\textbf{In}_2$, and can destruct a term of type $A \oplus B$ by pattern matching on it. The composition of

[3]This example is inspired by Spiwack (2014), who discusses this example in the context of the $\bar{\lambda}\mu\tilde{\mu}$-calculus, where the sequent calculus syntax leads to terms that are less natural from a programmer's point of view.

the three functions f, g and h can thus be written as follows:

$$\begin{aligned}
&h \circ g \circ f : A \to D \oplus E \\
&h \circ g \circ f = \lambda(x \cdot \alpha).\mathbf{Case}\ (f\,x)\ \{ \\
&\qquad \mathbf{In}_1\ y \mapsto \mathbf{Case}\ (g\,y)\ \{ \\
&\qquad\quad \mathbf{In}_1\ z \mapsto \langle h\,z \mid \alpha \rangle \\
&\qquad\quad \mathbf{In}_2\ e \mapsto \langle \mathbf{In}_2\ e \mid \alpha \rangle\} \\
&\qquad \mathbf{In}_2\ e \mapsto \langle \mathbf{In}_2\ e \mid \alpha \rangle\}
\end{aligned}$$

This style of error handling is familiar in conventional functional programming languages with algebraic types. The verbosity of this implementation can, of course, be greatly reduced by syntactic sugar or monadic do-notation. But what we really want to point out is how un-noteworthy the implementation is. This illustrates our point that we do not lose the naturality of natural deduction when writing programs using these calculi.

Error Handling Using a Negative Type

Implementing the same example using the negative type $⅋$ is less familiar:

$$\begin{aligned}
&h \circ g \circ f : A \to D ⅋ E \\
&h \circ g \circ f = \lambda(x \cdot \alpha). \\
&\qquad \langle \mathbf{Match}\ \{[\mathit{res}, \mathit{err}] \mapsto \\
&\qquad\quad \langle\ \mathbf{Handle}_2 \\
&\qquad\qquad h\ (\mathbf{Handle}_2 \\
&\qquad\qquad\quad g\ (\mathbf{Handle}_2\ f\ x\ \mathbf{with}\ \mathit{err}) \\
&\qquad\qquad\quad \mathbf{with}\ \mathit{err}) \\
&\qquad\qquad \mathbf{with}\ \mathit{err}) \mid \mathit{res}\ \rangle\ \} \\
&\qquad \mid \alpha \rangle
\end{aligned}$$

After introducing the variables x and α by a lambda abstraction, we encounter the $⅋$ introduction form $\mathbf{Match}\ \{[\mathit{res}, \mathit{err}] \mapsto \langle \ldots \mid \mathit{res} \rangle\}$, bringing two continuations into scope: the continuation res which we can use to return a result of type D and the continuation err which we can use to return an error of type E. Since we want to write our program following the happy path, we return directly to the result continuation, so we have to fill the hole with a term of type D. If t is a term of type $A ⅋ E$, then the elimination form $\mathbf{Handle}_2\ t\ \mathbf{with}\ e$ returns the first option A along the implicit happy path, and the possibility of an error case of type E is handled by the continuation e explicitly in the handler.

Comparison

Programming with both polarities feels natural, leaving the choice of which construct to use up to the programmer. In conventional (functional) programming languages, exceptions are usually modelled using the positive type $\oplus$. Since these languages are based on natural deduction, there is no natural way to model exceptions using the negative type $⅋$, apart from rewriting the program into continuation-passing or callback style. On the other hand, (checked) exceptions correspond more closely to the way we modelled exceptions with $⅋$. The difference is that instead of representing the additional exception path in the types, languages using checked exceptions usually model this continuation using the `throws` keyword, and usually scope the exception handler dynamically instead of lexically.

5.7.2. Parsimonious Filter

Let us now return to the parsimonious filter example from fig. 5.1 in section 5.1. How does that definition, based on equality between commands with nested (co)patterns, relate to the formal calculus that we have seen so far? To begin, consider just the left-hand sides of each definition in fig. 5.1 written by matching on the structures in a command (since the right-hand sides of each equality will be the same in each step, we will omit them for now):

$$\begin{aligned}
&\langle x && \mid \alpha \circ \mathbf{Not}\ f\ \rangle = \dots \\[1em]
&\langle \mathit{filter}\ p\ xs && \mid \mathit{start} \qquad\ \rangle = \dots \\[1em]
&\langle \mathit{filter/pass}\ p\ [] && \mid [\mathit{diff}, \mathit{same}]\rangle = \dots \\
&\langle \mathit{filter/pass}\ p\ (x :: xs) && \mid [\mathit{diff}, \mathit{same}]\rangle = \dots
\end{aligned}$$

The first step to desugaring this syntax is to replace each (left or right) elimination rule with the corresponding (left or right) introduction rule, according to the notion of bi-expressibility given here. For example, given a definition clause *filter* $p\ xs = \dots$ familiar to functional programmers, we replace the application forms (corresponding to right function elimination) with call stacks(corresponding to left function introduction) around the starting continuation $p \cdot xs \cdot \mathit{start}$. Dually, the composition operator $\circ$ combining a (negated) function with a continuation defines a consumer instead of a producer. The infix application form $\alpha \circ (\mathbf{Not}\ f)$ can be rewritten in prefix notation as $(\circ)\ \alpha\ (\mathbf{Not}\ f)$ (corresponding to left subtraction elimination). This is replaced with a stack (corresponding to right subtraction introduction) around the starting value: $\alpha \cdot \mathbf{Not}\ f \cdot x$. Replacing each (left and right) eliminations by its equivalent introduction leads to these definition clauses:

$$\begin{aligned}
&\langle \alpha \cdot \mathbf{Not}\ f \cdot x \mid (\circ) && \rangle = \ldots \\[1em]
&\langle \mathit{filter} \mid p \cdot xs \cdot \mathit{start} && \rangle = \ldots \\[1em]
&\langle \mathit{filter/pass} \mid p \cdot [\,] && \cdot [\mathit{diff}, \mathit{same}]\rangle = \ldots \\
&\langle \mathit{filter/pass} \mid p \cdot (x :: xs) && \cdot [\mathit{diff}, \mathit{same}]\rangle = \ldots
\end{aligned}$$

The next step is to combine the (multi-)clause definitions on commands into the definition of a single consumer or producer which matches on its input or output, respectively. This step can be performed uniformly with a **Match** introduction written with nested (co)patterns like so:

$$\begin{aligned}
(\circ) &= \mathbf{Match}\ \{\ \alpha \cdot \mathbf{Not}\ f \cdot x \mapsto \ldots \} \\[1em]
\mathit{filter} &= \mathbf{Match}\ \{\ p \cdot xs \cdot \mathit{start} \mapsto \ldots \} \\[1em]
\mathit{filter/pass} &= \mathbf{Match}\ \{\ p \cdot [\,] \qquad\quad \cdot [\mathit{diff}, \mathit{same}] \mapsto \ldots \\
&\qquad\qquad\quad\ p \cdot (x :: xs) \cdot [\mathit{diff}, \mathit{same}] \mapsto \ldots \}
\end{aligned}$$

The final step to desugaring is to flatten out the **Match** with nested (co)patterns into their single-step counterparts for each individual type. Fundamentally, flattening combined patterns and copatterns is not that different from the procedure of flattening ordinary nested patterns, involving tuples and sum types and other algebraic data types, typically done in conventional functional programming languages. The flattening of our parsimonious filter function looks like this:

$$\begin{aligned}
(\circ) &= \lambda(\alpha \cdot y).\ \mathbf{Case}\ y\ \{\ \beta \cdot x \mapsto \mathbf{Case}\ \beta\ \{\ \mathbf{Not}\ f \mapsto \ldots \}\ \} \\[1em]
\mathit{filter} &= \lambda(p \cdot \alpha).\ \mathbf{Case}\ \alpha\ \{\ xs \cdot \mathit{start} \mapsto \ldots \} \\[1em]
\mathit{filter/pass} &= \lambda(p \cdot \alpha).\ \mathbf{Case}\ \alpha\ \{\ ys \cdot \beta \mapsto \mathbf{Case}\ ys\ \{ \\
&\qquad [\,] \qquad\ \mapsto \mathbf{Case}\ \beta\ \{\ [\mathit{diff}, \mathit{same}] \mapsto \ldots \} \\
&\qquad x :: xs \mapsto \mathbf{Case}\ \beta\ \{\ [\mathit{diff}, \mathit{same}] \mapsto \ldots \}\ \}\ \}\ \}
\end{aligned}$$

The final, completely desugared and type-annotated version of fig. 5.1 into the core calculus syntax (with some trivial extensions, such as Booleans or recursion) is shown in fig. 5.9.

The desugared `filter/pass` function takes a predicate p on X as well as a list ys. Using $⅋$, it returns a List X if at least one of the elements of the input do not fulfil p and a unit value otherwise. This function uses direct style to analyze the list and split into one of three cases:

```
$\circ \colon Z \prec (\neg(Y \to Z) \prec Y)$
$\circ = \lambda(\alpha \colon Z \cdot y \colon \neg(Y \to Z) \prec Y).$
        $\textbf{Case}\ y\ \{\ \beta \colon \neg(Y \to Z) \cdot x \colon Y \mapsto$
          $\textbf{Case}\ \beta\ \{\ \textbf{Not}\ (f \colon Y \to Z) \mapsto \langle\, f\ x \mid \alpha \,\rangle\ \}\ \}$

$\texttt{filter} \colon (X \to \textsc{Bool}) \to \textsc{List}\ X \to \textsc{List}\ X$
$\texttt{filter} = \lambda(p \colon X \to \textsc{Bool} \cdot \alpha \colon \textsc{List}\ X \to \textsc{List}\ X).$
            $\textbf{Case}\ \alpha\ \{\ xs \colon \textsc{List}\ X \cdot start \colon \textsc{List}\ X \mapsto$
              $\langle \textbf{Handle}_2\ (\texttt{filter/pass}\ p)\ xs\ \textbf{with}\ (start \circ (\textbf{Not}\ (const\ xs)))$
              $\mid start \rangle\ \}$

$\texttt{filter/pass} \colon (X \to \textsc{Bool}) \to \textsc{List}\ X \to (\textsc{List}\ X \parr \top)$
$\texttt{filter/pass} =$
  $\lambda(p \colon X \to \textsc{Bool} \cdot \alpha \colon \textsc{List}\ X \to \textsc{List}\ X \parr \top).$
    $\textbf{Case}\ \alpha\ \{\ ys \colon \textsc{List}\ X \cdot \beta \colon \textsc{List}\ X \parr \top \mapsto$
      $\textbf{Case}\ ys\ \{$
        $[] \mapsto \textbf{Case}\ \beta\ \{\ [diff \colon \textsc{List} X, same \colon \top] \mapsto \langle\, \textbf{Match}\ \{\} \mid same \,\rangle \}$
        $(x \colon X :: xs \colon \textsc{List}\ X) \mapsto$
          $\textbf{Case}\ \beta\ \{\ [diff \colon \textsc{List} X, same \colon \top] \mapsto$
            $\textbf{Case}\ (p\ x)\ \{$
              $\textbf{True} \mapsto \langle\, \texttt{filter/pass}\ p\ xs \mid [diff \circ (\textbf{Not}\ (x\ ::)), same] \,\rangle$
              $\textbf{False} \mapsto \langle\, \texttt{filter/pass}\ p\ xs \mid [diff, diff \circ (\textbf{Not}\ (const\ xs))] \,\rangle\ \}\ \}\ \}\ \}$
```

Figure 5.9.: Parsimonious filter function reloaded

- The list is empty. In this case, we send the unit **Match** {} to the continuation *same*, which signals that the list contains no filtered elements.
- The list is non-empty and its head satisfies p. In this case, we call filter recursively on the tail. If that call signals that nothing is filtered in the tail, then nothing is filtered in the whole list (*same*). If something is filtered in the tail, we start constructing the new tail with the curried constructor $(x ::)$ and send the whole tail (constructed from this call to $(x ::)$ and the result of the recursive call) to *diff*.
- The list is non-empty and its head does not satisfy p. In this case we keep the default continuation as-is and reset the shared tail to the current tail, since we cannot use the previous one which would have contained x. We never invoke *same* and implicitly discard it before the recursive call.

We can then use this function with the `filter` wrapper, which captures the current continuation *start* with a λ and uses $\mathbf{Handle}_2$ to discharge the "same" case with a continuation that passes the list to *start*.

The infix $\circ$ utility function is interesting in that it uses both the $\rightarrow$ and the $\neg$ connective. It demonstrates the utility of having λ and application forms on the consumer side, and of using negation to pass producers into a consumer context.

This example demonstrates the benefits of having all rules available. Compared to sequent calculus, all program parts can be written in direct style, using appropriate elimination forms. The usage and choice of polarized connectives leads to a more structured and readable program than low-level usages of `call/cc` and similar control operators.

In what sense is this implementation of `filter` "efficient"? Consider the expression

$$\langle\, \texttt{filter}(> 100)[0 \ldots 10^6] \mid \mathit{start}\,\rangle.$$

It will reduce to

$$\begin{aligned}&\langle \texttt{filter/pass}(> 100)\ [] \mid \\ &\quad [\mathit{start} \circ \mathbf{Not}(101 ::) \circ \ldots \circ \mathbf{Not}\ (10^6 ::), \mathit{start} \circ \mathbf{Not}\ (\texttt{const}[101 \ldots 10^6])]\rangle\end{aligned}$$

which can immediately return to the second continuation with

$$\langle\, () \mid \mathit{start} \circ \mathbf{Not}\ (\mathit{const}[101 \ldots 10^6])\,\rangle$$

and thus avoid unwinding the stack built up in the left-hand *diff* continuation. In this sense, the filter/pass function is "partially" tail-recursive: the recursive stack frame for the pass case can be optimized away in this program, but not the return pointer for when an element is removed. In a real implementation of our calculus, this would allow the compiler to use sharing on the $[101 \ldots 10^6]$ tail of the input list.

To summarize, in a real implementation of the language, the code in fig. 5.9 would combine these three properties:

1. Tail-call optimization of recursive calls when the head is removed.

2. Sharing the common list suffix rather than reallocating it.

3. Immediately jumping to the starting caller when there is no more prefix to append to a modified tail.

5.8. Related and Future Work

Computational Sequent Calculus. Our work is obviously related to previous sequent calculus-based languages, in particular the $\overline{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin (2000) and the dual calculus of Wadler (2003). Being directly inspired by the sequent calculus, each of these calculi define *distinct* syntactic categories for producers and consumers — for syntactically representing the sequent calculus' left and right rules logical rules — and do not feature elimination forms. The syntactic categories of $\overline{\lambda}\mu\tilde{\mu}$ with only function types are similar but not fully isomorphic; this is why Curien and Herbelin (2000) extend $\overline{\lambda}\mu\tilde{\mu}$ with a subtraction type, corresponding to the $-\!\!\prec$ connective discussed here, which completes the duality with function types. In contrast, Wadler (2003)'s dual calculus eschews functions altogether, instead focusing on only conjunction, disjunction, and negation, presented in a non-polarized style (that is, the single conjunction type is produced by the pair (x, y) and consumed by the projections $\mathrm{fst}[\alpha]$ and $\mathrm{snd}[\beta]$). These connectives are given two dual interpretations — one following a call-by-value semantics and one following call-by-name — which turns out to reveal the hidden polarities of the connectives: under call-by-value conjunction and disjunction correspond equationally to the positive $\otimes$ and $\oplus$ and under call-by-name they correspond to the negative $\&$ and $⅋$ discussed here, but not vice versa (Downen and Zena M. Ariola, 2014). Our calculus also resembles the presentation of the calculus of *classical natural deduction* in Lovas and Crary (2006), based on work by Nanevski. That calculus contains products and sums, but does not differentiate between different polarizations. Instead, it uses the left introduction rules of the corresponding positive and right introduction rules of the corresponding negative connectives. They also don't have elimination rules in the core system but instead encode the right elimination rules in a manner similar to our diagonal encodings.

$\lambda\mu$ **Calculus.** The $\lambda\mu$ calculus Parigot, 1992a is a natural deduction style language that corresponds to classical logic. We conjecture that it can be very straightforwardly embedded into our calculus with the following compositional transformation:

$$\begin{array}{lcll}
[\![x]\!] & = & x & \\
[\![\lambda x.e]\!] & = & \lambda(x \cdot \alpha).\langle\, [\![e]\!] \mid \alpha\,\rangle & \alpha \textit{ fresh} \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] & \\
[\![\mu\beta.([\alpha]\ e)]\!] & = & \mu\beta.\langle\, [\![e]\!] \mid \alpha\,\rangle &
\end{array}$$

Wadler (2005) describes a translation from $\lambda\mu$ to his aforementioned dual calculus (and back); due to the absence of elimination forms, the translation is considerably more complicated.

Translating Natural Deduction to Sequent Calculus Gentzen (1935a) and Gentzen (1935b) describes a translation of derivations in the intuitionistic deduction system NJ into the intuitionistic sequent calculus LJ. In this translation, elimination rules are transformed into usages of the corresponding left rule for the connective and then an invocation of the cut rule, which means that normal forms are in general not translated to normal forms. Prawitz (1965, p.92) discusses a translation from natural deduction to sequent calculus that preserves normal forms, but at the expense of compositionality: The translation extends the sequent calculus derivation at its bottom when translating an introduction rule but from the top when an elimination rule is translated. Curien and Herbelin (2000) present two term-level translations $^{\mathcal{N}}$ and $^{>}$ that correspond to Prawitz' and Gentzen; Gentzen's proposals, respectively. These translations are similar to the elimination rule encodings of bi-expressibility, and in terms of continuation-passing style $^{>}$ is analogous to Hofmann and Streicher (1997)'s call-by-name CPS transformation whereas $^{\mathcal{N}}$ corresponds to Plotkin (1975)'s colon transformation.

Subtractive Logic. A dual to implication called subtraction, pseudo-difference, or co-implication is well-known in the domain of (bi-intuitionstic) logic Rauszer, 1974; Tranchini, 2012, but finding an intuitive operational interpretation turned out to be difficult. Crolard (2004) proposed a rather complicated operational interpretation as "coroutines". We suggest that the reason for the complication is that Crolard considered an asymmetrical language with no consumer language. We think that our completely symmetric rules for $\rightarrow$ and $\prec$, with all rules, together with their simple operational semantics, is an improvement over these works.

Linear Logic, Polarity, Data and Codata Types An important step in the development of the proof theory of sequent calculus was the discovery of linear logic by Girard (1987). Linear logic restricts the applicability of structural rules, like weakening and contraction, which makes it possible to use a resource reading of typing judgements Wadler, 1990: variables bound in the context are resources which are consumed in the construction of terms, which explains why they cannot be freely duplicated or discarded. A consequence of this resource interpretation is that the ordinary connectives of classical or intuitionistic logic have to be split into multiple connectives; e.g. $\wedge$ has to be split into $\otimes$ and $\&$, $\vee$ has to be split into $\oplus$ and $⅋$. Studying proof search for linear logic, Andreoli (1992) realized that these connectives fall into two classes, which he called synchronous and asynchronous; later this terminology changed to positive and negative polarity. In programming language terms, polarity corresponds to the distinction between data types which are defined via their constructors, and codata types Hagino, 1989; Downen, Sullivan, Zena M. Ariola, and Jones, 2019 which are defined via their observations/destructors. For example, while both $\oplus$ and $⅋$ are disjunctions, $\oplus$ corresponds to a data type defined with the help of two injections constructors, while $⅋$ corresponds to a codata type defined with one destructor bringing two continuations into scope. As argued by Zeilberger (2009) and many others, this distinction is important even in a system which does not enforce the linear use of variables because it determines evaluation order. We have illustrated this with examples where the availability of polarized connectives was critical.

When considering user-defined data and codata types, as we plan to do in future work, one has to specify generic mechanisms to construct and destruct terms of those types. Copattern matching Abel, Pientka, Thibodeau, and Setzer, 2013 was introduced as a generic mechanism for constructing inhabitants of codata types, dually to how pattern matching is used to destruct inhabitants of data types. For example, Zeilberger (2008b) provided a calculus where pattern matching and copattern matching, constructors and destructors are the only term-level constructs. In the context of the sequent calculus, Downen and Zena M. Ariola (2021) provide a variant of the $\mu/\tilde{\mu}$ calculus with user provided data and codata types, but consider only the left and right introduction forms of the sequent calculus, and no elimination forms. The calculus presented in this chapter could be similarly presented with user-defined data and codata type declarations, with introduction and elimination forms, left and right, derived from these declarations. In fact, a lot of the sets of rules presented here were considered by us in this more general form first, but we defer a full development of that idea to future work.

One-Sided versus Two-Sided The calculus we present here is a *two sided* sequent calculus, in the sense that we use both sides of the sequent separated by $\vdash$: producers live on the right-hand side and consumers live on the left. This distinction can quickly be summarized by the cut rule used in our core calculus:

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash f \overset{\mathbf{con}}{:} T}{\Gamma \vdash \langle\, e \mid f \,\rangle}$$

Importantly, this rule promises any producer e of a type T can interact with any consumer f of the *same type* T. But this isn't the only way to arrange interaction in a sequent calculus. A popular variant in the setting of classical linear logic (Girard, 1987) is a *one sided* sequent calculus, which only ever uses a single side of the sequent throughout. This "restriction" can be made without loss of expressivity because of involutive negation in classical (linear) logic, for every proposition A there is a dual proposition $A^{\perp}$ such that $A^{\perp\perp} = A$, such that having A on the left of $\vdash$ is the same as having $A^{\perp}$ on the right. This involutive negation corresponds to the duality of types, here written as T°, which can be used to formulate a one-sided language. Focusing again on the iconic cut rule, we have two more possibilities for arranging a one-sided version of the calculus as

$$\frac{\vdash e : T \mid \Delta \qquad \vdash f : T^{\circ} \mid \Delta'}{\langle\, e \mid f \,\rangle : (\vdash \Delta, \Delta')} \qquad\qquad \frac{\Gamma \vdash e : T \qquad \Gamma' \vdash f : T^{\circ}}{\langle\, e \mid f \,\rangle : (\Gamma, \Gamma' \vdash)}$$

by putting *all* types on the right of $\vdash$ (Munch-Maccagnoni, 2009) (as is popular in linear logic) or *all* variables to the left of $\vdash$ and the expression to the right (Spiwack, 2014) (more popular in the programming languages community). The important thing to notice about these one-sided cut rules is the promise that any producer of a type T can interact with any *other producer* of the *opposite type* T°.

The idea of connecting producers to other producers fits with our idea of consumer/producer polymorphism in section 5.5. Essentially, the point is that if the duality is complete enough, and if consumers of T are completely interchangeable with producers of T°, then two producers (or two consumers) of opposite types should be able to interact directly with one another. In such a setting there is no difference between T consumers and T° producers, so Munch-Maccagnoni (2009) eliminates the distinction between the commands $\langle e \mid f \rangle$ and $\langle f \mid e \rangle$. We conjecture that a complete story of consumer/producer polymorphism should elaborate on this one-sided view of sequents.

Left calculus Right calculi are abounds in the literature behind both the fields of logic and programming languages. Intro calculi frequently appear in the study of proof theory, and are growing in number to help understand and implement programming languages, too. However, examples of left calculi can scarcely be found in the literature.

The idea of left elimination rules has been presented before by Carraro, Ehrhard, and Salibra (2012), which give a calculus similar to $\overline{\lambda}\mu\tilde{\mu}$ with a left rule for introducing function call stacks, but instead of ordinary λ-abstractions, it contains projections that access the argument and the return-pointer in a call stack. These projections are similar to the left elimination rule for functions presented here, by the analogy that a **Case** extracting the two components of a pair of two things is similar to projections from that pair to each component individually.

Independently, Nakazawa and Nagai (2014) introduces the same combination of left introduction and eliminations for functions in the context of the $\Lambda\mu$-calculus (Groote, 1994), a variant of Parigot's $\lambda\mu$-calculus which collapses the distinction between commands and producers. This collapse improves the completeness properties of the rewriting theory Saurin, 2005 and elevates μ to a much more expressive *delimited control operator* Herbelin and Ghilezan, 2008, and the use of left elimination rules were key to reconciling extensionality (expressed by the λ-calculus' η law) of delimited control in $\Lambda\mu$ with standard properties like confluence.

Elimination calculus We know of two calculi which correspond to what we call an elimination calculus. Negri's uniform calculus for classical linear logic (cf. Negri (2002) and Negri and Von Plato (2001, p. 213ff.)) is a termfree logical calculus with only elimination rules. Negri uses the term "general introduction rule" for what we call left elimination rules. Natural deduction and sequent calculus can be obtained by instantiating major and minor premises of these elimination rules with an instance of the axiom rule. This is similar to our bi-expressibility rules, but since she doesn't have to consider term assignment, she also doesn't have to deal with the activation and deactivation of formulas.

Parigot's free deduction Parigot, 1992b, a precursor to his $\lambda\mu$-calculus Parigot, 1992a, also contains only elimination rules, and he also obtains natural deduction and sequent calculus by instantiating major and minor premises by the axiom rule. In distinction to Negri, he also provides a term system with constructs which can be seen as precursors to both the μ and $\tilde{\mu}$-abstractions of Curien and Herbelin (2000).

Communication calculi and session types There is a well-known relationship between linear logic and session types, both in its intuitionistic Caires and Pfenning, 2010 and its classical Wadler, 2014 version. These session type systems are based on some form of communication calculus, like the π-calculus, and provide a type system for the communication channels. There are two relationships between session type systems and our work. First, there is a relationship between the duality operation T° on types and a similar operation on session types: The dual of a session type for sending some data is a session type for receiving some data of that type, the dual of a session type for choosing between various options is a session type for offering those choices, and so on. The second relationship concerns the non-local character of reduction in a communication calculus: The term which wants to send some information on a channel might be at some distance from the term for receiving information on that channel, but they nevertheless interact in a reduction step. This can also be observed in the elimination calculus, if we would formalize the reduction rules for that system directly. For example Parigot (1992b), who considers reduction rules for a system based on elimination rules, uses the following version of the axiom rule (which is derivable in our system)

$$\frac{}{x : A, x^\perp \overset{\text{con}}{:} A \vdash \langle\, x \mid x^\perp \,\rangle}\ \textsc{Axiom}$$

A cut in his system then consists between an elimination rule which uses x as the major premise, and an elimination rule which uses $x^\perp$ as the major premise, but they don't have to stand directly next to each other. The resulting reduction system is closely reminiscent of a communication step happening between two ends of a channel. We plan to investigate both these aspects in more detail in future work.

Relations between rules Our bi-expressibility principle concerns the "diagonal" relation between right-intro/left-elim and left-intro/right-elim. There are other sanity principles for rule pairs, but they concern the relation between introduction and elimination rules and are of particular interest in proof-theoretic semantics, such as *invertibility* and *harmony* Schroeder-Heister, 2018.

Transformations between consumers and producers Our approach to consumer-producer polymorphism is related to but very different from whole-program transformations (generalizations of defunctionalization and refunctionalization) that transform data types into codata types or vice versa (Rendel, Trieflinger, and Ostermann, 2015 and chapters 3 and 4). Consumer-producer polymorphism allows us to use the same program in two ways, once as a consumer and once as a producer. When viewed as a macro code generator, it is a compositional transformation. The aforementioned generalizations of de- and refunctionalization, on the other hand, transform a whole program in a way that can be viewed as a matrix transposition Ostermann and Jabs, 2018, while preserving its operational behavior. For instance, when defunctionalizing a codata type into a data type, all copattern-matches on a destructor in the whole program are turned into a single pattern match.

5.9. Conclusions

Programming abstractions based on sequent calculus have, despite their attractive symmetry and expressive power, seen only limited influence on functional language design. We have opened up the design space of natural deduction and sequent calculus by considering all four kinds of rules and the four natural subcalculi. We have analyzed the interdependency between program structure and rule choice and have argued that offering all rules to the programmer maximizes expressiveness and allows a natural and modular program structure. We have proposed a constructive sanity check for the rules, bi-expressibility, and have shown how the dualities between the available connectives can be deepened in the form of a uniform syntax and consumer/producer polymorphism.

6. A Tale of Two Transformations: Administrative Normal Forms and Focusing

The content of this chapter is based on the following publication:

> David Binder and Thomas Piecha (2022). "Administrative Normal Forms and Focusing for Lambda Calculi". In: *Logically Speaking. A Festschrift for Marie Duží.* Ed. by Pavel Materna and Bjørn Jespersen. Vol. 49. Tributes. College Publications

In this chapter I introduce the correspondence between two normal forms and their normalization procedures: administrative normal forms and the ANF-transformation on the one hand, and focused normal forms and static focusing on the other. The administrative normal form applies to the natural-deduction based calculi that I presented in part I, whereas the focused normal form belongs to the sequent calculi presented in part II. These normal forms were invented for different purposes, compiler optimizations in the case of the ANF-transformation and proof search in the case of focusing, but they are structurally very similar. Both transformations bring proofs into a normal form where functions and constructors are only applied to values and where computations are sequentialized. Researchers familiar with both the ANF transformation and focusing are likely aware that these two transformations are similar in nature. The contribution of this chapter consists in making this relationship completely explicit for the first time.

We will use the λ-calculus and the $\lambda\mu\tilde{\mu}$-calculus, which are related by a translation function from λ-terms Λ to $\lambda\mu\tilde{\mu}$-terms $\Lambda_{\mu\tilde{\mu}}$. The two calculi that we consider in this chapter do not support arbitrary data and codata types. Instead, I have specialized the presentation to just the type of functions and products. For the λ-calculus we define the administrative normal form Λ^{ANF}, together with a transformation from Λ to Λ^{ANF}. In distinction to the usual presentation of the ANF-transformation, we divide this transformation into two parts by using an intermediate normal form Λ^{Q} between Λ and Λ^{ANF}. For the $\Lambda_{\mu\tilde{\mu}}$-calculus we define the so-called *focused normal form* $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ (which corresponds to the subsyntax LKQ of Curien and Herbelin, 2000). The focusing transformation from $\Lambda_{\mu\tilde{\mu}}$ to $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ is adapted from Downen and Zena M. Ariola, 2018a. We define a new normal form $\Lambda^{\mathrm{ANF}}_{\mu\tilde{\mu}}$ for $\lambda\mu\tilde{\mu}$-terms, which exactly mirrors the syntactic restrictions that characterize the administrative normal form Λ^{ANF} for λ-terms.

Our main result is depicted in fig. 6.1. We show how the ANF-transformation on λ-terms corresponds to static focusing of $\lambda\mu\tilde{\mu}$-terms. The first part of the ANF-transformation corresponds precisely to the static focusing transformation. That is, it commutes with focusing via the translation function up to α-equivalence. The second part of the ANF-transformation can be simulated in the $\lambda\mu\tilde{\mu}$-calculus by μ-reductions.

$$\begin{tikzcd}
\Lambda \arrow[r, "\text{translation}"] \arrow[d, "\text{ANF-transformation (1)}"'] & \Lambda_{\mu\tilde{\mu}} \arrow[d, "\text{focusing}"] \\
\Lambda^{\mathrm{Q}} \arrow[r, "\text{translation}"] \arrow[d, "\text{ANF-transformation (2)}"'] & \Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}} \arrow[d, "\mu\text{-reduction}"] \\
\Lambda^{\mathrm{ANF}} \arrow[r, "\text{translation}"] & \Lambda^{\mathrm{ANF}}_{\mu\tilde{\mu}}
\end{tikzcd}$$

Figure 6.1.: The relationship between the ANF-tranformation of λ-terms and focusing of $\lambda\mu\tilde{\mu}$-terms.

The rest of this chapter is structured as follows. In section 6.1 we present the main idea using an informal example. In section 6.2 we formalize the syntax and type assignment rules for the λ-calculus and the $\lambda\mu\tilde{\mu}$-calculus, and in section 6.3 we give the translation from the former to the latter. In section 6.4 we provide the call-by-value operational semantics for both calculi. We introduce the ANF-transformation in section 6.5 and static focusing in section 6.6. The main result is presented in section 6.7 and summarized in section 6.8, which also contains an outlook to future work. The proofs of the main theorems can be found in section 6.9.

6.1. An informal example

We explain the main idea with an informal example. Consider the following program

$$\pi_2(\pi_1(1,4),3)$$

which consists of natural numbers 1, 4 and 3, pair constructors $(\llcorner,\llcorner)$ and projections $\pi_1\ \llcorner$ and $\pi_2\ \llcorner$ on the first and second element of a pair, respectively.

We expect this program to evaluate to the natural number 3. Using call-by-name we could immediately evaluate this program to its final value 3. However, using call-by-value we first have to evaluate the argument of π_2 to the value $(1,3)$ by evaluating $\pi_1(1,4)$ to 1.

There are different ways to formalize the evaluation of a term within a context. Here we choose the method of evaluation contexts (cf. Felleisen and Hieb (1992) and section 6.4.1 below). An *evaluation context* $E[-]$ is a term with a placeholder $\square$, which is to be filled with the outermost redex to be evaluated next. We will use the symbol $\doteq$ throughout to express syntactic equality up to α-equivalence (i.e., up to the renaming of bound variables).

In our example, this allows us to evaluate the outermost redex $\pi_1(1,4)$ within the context $E[-] \doteq \pi_2(\Box, 3)$ as follows:

$$\text{If } \pi_1(1,4) \triangleright 1, \text{ then } \pi_2(\pi_1(1,4),3) \doteq E[\pi_1(1,4)] \triangleright E[1] \doteq \pi_2(1,3).$$

The translation (cf. definition 6.3.1) of the program $\pi_2(\pi_1(1,4),3)$ into the $\lambda\mu\tilde{\mu}$-calculus (cf. section 6.2.3) results in the program

$$\mu\alpha.\langle\, (\mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle, 3) \mid \pi_2\, \alpha\,\rangle.$$

We can recognize many familiar constructs from the initial program. We still have natural numbers 1, 4 and 3, the pair constructor $(\textvisiblespace, \textvisiblespace)$ and projections π_1 and π_2, but they are now organized and nested in a very different way with the help of two new constructs.

The first new construct is the *cut* $\langle \textvisiblespace \mid \textvisiblespace \rangle$ which is used to oppose a *proof* (or *proof term*) of a proposition with its *refutation* (or *refutation term*). In our example, we use the cut to oppose a proof $(1,4)$ of the type $\mathbb{N} \wedge \mathbb{N}$ with a refutation $\pi_1\, \beta$ of the same type, where we assume that the covariable β stands for some unknown refutation of type $\mathbb{N}$. The reduction rules of the $\lambda\mu\tilde{\mu}$-calculus always replace a cut by another cut, and in the case of pairs the reduction rule allows to replace $\langle\, (1,4) \mid \pi_1\, \beta\,\rangle$ by the new cut $\langle\, 1 \mid \beta\,\rangle$.

The second new construct is the μ-abstraction $\mu\alpha.\textvisiblespace$. We have more to say about this construct in section 6.2.3, but for now it suffices to say that we use $\mu\alpha.\langle \textvisiblespace \mid \textvisiblespace \rangle$ to introduce a subcomputation (represented by the cut $\langle \textvisiblespace \mid \textvisiblespace \rangle$) returning to the output named by the covariable α. For example, in order to represent the subcomputation $2+2$, we use the term $\mu\alpha.\langle\, 2+2 \mid \alpha\,\rangle$, which evaluates to $\mu\alpha.\langle\, 4 \mid \alpha\,\rangle$.

We cannot evaluate the program $\mu\alpha.\langle\, (\mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle, 3) \mid \pi_2\, \alpha\,\rangle$ directly to its final value, since one can only evaluate cuts $\langle \textvisiblespace \mid \textvisiblespace \rangle$, whereas this program has the form of a μ-abstraction. This can be resolved by introducing a third construct, namely the *toplevel output* Top, which enables us to embed any μ-program in a cut whose second element is Top. Furthermore, a $\tilde{\mu}$-abstraction $\tilde{\mu}x.\langle \textvisiblespace \mid \textvisiblespace \rangle$ has to be used, which binds a value to the variable x in the subcomputation $\langle \textvisiblespace \mid \textvisiblespace \rangle$.

The example program then evaluates in the following way:

$$\begin{aligned}
&\langle\, \mu\alpha.\langle\, (\mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle, 3) \mid \pi_2\, \alpha\,\rangle \mid \mathsf{Top}\,\rangle && (6.1)\\
&\triangleright \langle\, (\mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle, 3) \mid \pi_2\, \mathsf{Top}\,\rangle && (6.2)\\
&\triangleright \langle\, \mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle \mid \tilde{\mu}x.\langle\, (x,3) \mid \pi_2\, \mathsf{Top}\,\rangle\,\rangle && (6.3)\\
&\triangleright \langle\, (1,4) \mid \pi_1(\tilde{\mu}x.\langle\, (x,3) \mid \pi_2\, \mathsf{Top}\,\rangle)\,\rangle && (6.4)\\
&\triangleright \langle\, 1 \mid \tilde{\mu}x.\langle\, (x,3) \mid \pi_2\, \mathsf{Top}\,\rangle\,\rangle && (6.5)\\
&\triangleright \langle\, (1,3) \mid \pi_2\, \mathsf{Top}\,\rangle && (6.6)\\
&\triangleright \langle\, 3 \mid \mathsf{Top}\,\rangle && (6.7)
\end{aligned}$$

Note that in step (6.5) we project from $(1,4)$ to 1 without being in an evaluation context. The evaluation within an evaluation context is instead simulated by steps (6.4) and (6.6).

That is, steps (6.4) to (6.6) correspond to the single evaluation step

$$\pi_2(\pi_1(1,4),3) \triangleright \pi_2(1,3).$$

This sort of evaluation within a context, which is present in both the λ-calculus and the $\lambda\mu\tilde{\mu}$-calculus, poses no problem from a theoretical point of view. However, from a practical point of view, it is very inefficient to apply this kind of operational semantics since the search for a redex requires in general to traverse deeply into a term. Moreover, evaluations of this kind render the implementation of various compiler optimizations (cf. Sabry and Felleisen, 1992; Flanagan, Sabry, B. F. Duba, and Felleisen, 1993) more difficult. These difficulties can be avoided by using certain normal forms, for example, the so-called *administrative normal form* (*A-normal form* or *ANF*)[1] for the λ-calculus, and the *focused normal form* for the $\lambda\mu\tilde{\mu}$-calculus.

The ANF of the first example program

$$\pi_2(\pi_1(1,4),3)$$

is

$$\textbf{let } x = \pi_1(1,4) \textbf{ in } (\textbf{let } y = \pi_2(x,3) \textbf{ in } y), \tag{A}$$

whereas the focused normal form of the second program

$$\mu\alpha.\langle\, (\mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle, 3) \mid \pi_2\, \alpha\,\rangle$$

is

$$\mu\alpha.\langle\, \mu\beta.\langle\, (1,4) \mid \pi_1\, \beta\,\rangle \mid \tilde{\mu}x.\langle\, (x,3) \mid \pi_2\, \alpha\,\rangle\,\rangle. \tag{F}$$

Comparing the ANF (A) with the focused normal form (F) makes the structural similarity between the two normal forms apparent: in both cases the subcomputation $\pi_1(1,4)$ (resp. $\langle\, (1,4) \mid \pi_1\, \beta\,\rangle$) was lifted out and then bound to the variable x in the subsequent computation $\pi_2(x,3)$ (resp. $\langle\, (x,3) \mid \pi_2\, \alpha\,\rangle$). The difference between (A) and (F) consists in the use of let-constructs in the λ-calculus on the one hand and the use of μ- and $\tilde{\mu}$-constructs in the $\lambda\mu\tilde{\mu}$-calculus on the other hand.

6.2. Syntax and type assignment

We present the syntax and type-assignment rules of the λ-calculus and of the $\lambda\mu\tilde{\mu}$-calculus. The syntax for types is the same in both calculi.

Definition 6.2.1 (Types)**.** There are three kinds of *types* τ:

$$\tau ::= X \quad\mid\quad \tau \to \tau \quad\mid\quad \tau \wedge \tau.$$

That is, we have *atomic types* X, *implication types* $\tau \to \tau$ and *conjunction types* $\tau \wedge \tau$.

[1]While the "A" in "A-normal" originally had no special meaning, it was later given the meaning of "administrative normal form", due to the administrative redexes it introduces.

6.2.1. The λ-calculus

We use the standard simply typed λ-calculus with conjunction and a let-construct (cf., e.g., Benjamin C. Pierce, 2002). Since we only consider a call-by-value evaluation strategy, the values consist of variables, λ-abstractions and tuples of values.

Definition 6.2.2. The *syntax* Λ *of the* λ*-calculus* is defined as follows, where x are *term variables*:

1. *Terms:* $e ::= x \;\mid\; \lambda x.e \;\mid\; e\,e \;\mid\; (e,e) \;\mid\; \pi_1\, e \;\mid\; \pi_2\, e \;\mid\; \mathbf{let}\ x = e\ \mathbf{in}\ e.$
2. *Values:* $v ::= \lambda x.e \;\mid\; (v,v) \;\mid\; x.$

A *judgement* is a sequent of the form $\Gamma \vdash e : \tau$, where Γ is a (possibly empty) set of declarations $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

Definition 6.2.3. The *type assignment rules of the* λ*-calculus* are:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}\ \textsc{Let}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma \wedge \tau}\ \textsc{Pair} \qquad \frac{\Gamma \vdash e : \tau_1 \wedge \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i}\ \textsc{Proj}$$

Note that rule Proj comprises the two cases where either $i = 1$ or $i = 2$.

There are no structural rules since weakening and contraction are implicit. Note that the rule Let is derivable since any term **let** $x = e_1$ **in** e_2 can be replaced by $(\lambda x.e_2)e_1$ without changing the type in the conclusion of a type assignment. However, let-bindings are used to make the evaluation order explicit; we will come back to this point in section 6.6.

6.2.2. Towards the $\lambda\mu\tilde{\mu}$-calculus

The λ-calculus corresponds to natural deduction for the $\{\to, \wedge\}$-fragment of intuitionistic logic. The $\lambda\mu\tilde{\mu}$-calculus Curien and Herbelin, 2000 was introduced as a system that corresponds to the classical sequent calculus, in which sequents have the form $\Gamma \vdash \Delta$ with (possibly empty) sets Γ, Δ of formulas on either side of the sequent symbol $\vdash$.

The usual interpretation of a valid classical sequent $\Gamma \vdash \Delta$ can be expressed as "If all the formulas in Γ are true, then at least one of the formulas in Δ is true." This interpretation has to be refined in order to understand the correspondence between the $\lambda\mu\tilde{\mu}$-calculus and the classical sequent calculus. The refinement consists in distinguishing three variants of the sequent $\Gamma \vdash \Delta$:

1. $\Gamma \vdash [\phi], \Delta$

   "If all $\gamma \in \Gamma$ are true and all $\delta \in \Delta$ are false, then ϕ is true."

2. $\Gamma, [\phi] \vdash \Delta$

   "If all $\gamma \in \Gamma$ are true and all $\delta \in \Delta$ are false, then ϕ is false."

3. $\Gamma \vdash \Delta$

   "The assumption that all $\gamma \in \Gamma$ are true and all $\delta \in \Delta$ are false is contradictory."

The formula in square brackets $[\phi]$ is called the *active* formula of the sequent. There can be at most one active formula in any sequent.

The $\lambda\mu\tilde{\mu}$-calculus has one syntactic category and one judgement form for each of these three interpretations:

1. The active formula ϕ in the succedent of a sequent $\Gamma \vdash [\phi], \Delta$ is assigned to a *term* e, and the corresponding *judgement form* is

   $$\Gamma \vdash e : \phi \quad | \quad \Delta.$$

   Here the symbol $\quad | \quad$ singles out a formula ϕ for which the proof e is currently constructed (cf. Curien and Herbelin, 2000).

2. The active formula ϕ in the antecedent of a sequent $\Gamma, [\phi] \vdash \Delta$ is assigned to a *coterm* s, and the corresponding *judgement form* is

   $$\Gamma \quad | \quad s : \phi \vdash \Delta.$$

   In this case, the symbol $\quad | \quad$ singles out a formula ϕ for which the refutation s is currently constructed.

3. A sequent $\Gamma \vdash \Delta$ with no active formula is interpreted by a *command* c, and the corresponding *judgement form* is

   $$c : (\Gamma \vdash \Delta).$$

   This judgement form can be read as follows: "If all $\gamma \in \Gamma$ are true and all $\delta \in \Delta$ are false, then c is a contradiction and a well-typed command."

6.2.3. The $\lambda\mu\tilde{\mu}$-calculus

We consider the syntax of the $\lambda\mu\tilde{\mu}$-calculus. We have to partition the set of λ-terms into the three syntactic categories of the $\lambda\mu\tilde{\mu}$-calculus, namely terms, coterms and commands. The basic idea is that the introduction forms $\lambda x.e$ and (e, e) (which correspond to the introduction rules in natural deduction) will remain terms of the $\lambda\mu\tilde{\mu}$-calculus. On the other hand, the elimination forms $\pi_i e$ and $e\,e$ (which correspond to the elimination rules in natural deduction) will become coterms. The terms of the $\lambda\mu\tilde{\mu}$-calculus therefore comprise the introduction forms $\lambda x.t$ and (t, t) of the λ-calculus, whereas the coterms comprise the elimination forms $\pi_i\ s$ and $t \cdot s$.

There are different ways to understand a coterm $t \cdot s$. First, since an implication $\phi \to \tau$ is false if ϕ is true and τ is false, one can interpret $t \cdot s$ as a constructive refutation of an implication $\phi \to \tau$, consisting of a proof t of ϕ and a refutation s of τ. Alternatively, in a computational context, $t \cdot s$ can be thought of as a stack frame in a call stack with argument t on top and s being the rest of the stack.

There is only one form of command in the $\lambda\mu\tilde{\mu}$-calculus: the *cut* $\langle t \mid s \rangle$, which combines a term with a coterm. The cut rule can be interpreted as a primitive way to construct a contradiction, namely by providing both a proof and a refutation of the same formula.

This leaves us with the two remaining constructs of μ- and $\tilde{\mu}$-abstraction, which, again, can be understood in two different ways. First, from a logical point of view, the μ-construct encodes a form of *reductio ad absurdum* at the level of judgements:

$$\frac{\begin{array}{c}[\phi \text{ is false}]\\ \vdots \\ \text{contradiction}\end{array}}{\phi \text{ is true}}\ (\mu)$$

This explains why the addition of μ-abstraction makes the logic classical. The $\tilde{\mu}$-construct, on the other hand, encodes the logical inference

$$\frac{\begin{array}{c}[\phi \text{ is true}]\\ \vdots \\ \text{contradiction}\end{array}}{\phi \text{ is false}}\ (\tilde{\mu})$$

Both inferences are on the level of judgements and do not involve logical constants; neither absurdity $\bot$ nor negation $\neg$ are used.

Second, from an operational point of view we see that $\tilde{\mu}$ behaves very similarly to the let-construct of the λ-calculus. In a command $\langle t \mid \tilde{\mu}x.c \rangle$, the $\tilde{\mu}$-abstraction is used to bind the term t in the remaining computation c. The μ-construct behaves similarly to control operators like call/cc or $\mathcal{C}$ (cf. Zena M. Ariola and Herbelin, 2003; Griffin, 1989).

Definition 6.2.4. The *syntax* $\Lambda_{\mu\tilde{\mu}}$ *of the* $\lambda\mu\tilde{\mu}$*-calculus* is defined as follows, where x are *term variables* and α are *coterm variables*:

1. *Terms:* $t ::= x \quad | \quad \lambda x.t \quad | \quad (t,t) \quad | \quad \mu\alpha.c.$
2. *Coterms:* $s ::= \alpha \quad | \quad t \cdot s \quad | \quad \pi_1\, s \quad | \quad \pi_2\, s \quad | \quad \tilde{\mu}x.c.$
3. *Commands:* $c ::= \langle t \mid s \rangle.$
4. *Values:* $w ::= \lambda x.t \quad | \quad (w,w) \quad | \quad x.$

In addition to term variable contexts $\Gamma \doteq \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, we now have to consider also covariable contexts $\Delta \doteq \{\alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n\}$.

Definition 6.2.5. The *type-assignment rules of the* $\lambda\mu\tilde{\mu}$*-calculus* for the three judgement forms

1. *term typing:* $\Gamma \vdash t : \tau \quad | \quad \Delta$,

2. *coterm typing:* $\Gamma \mid s : \tau \vdash \Delta$, and
3. *command typing:* $c : (\Gamma \vdash \Delta)$

are the following:

Term typing

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \mid \Delta}\ \textsc{Var}_x$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau \mid \Delta}{\Gamma \vdash \lambda x.t : \sigma \to \tau \mid \Delta}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \mid \Delta \qquad \Gamma \vdash t_2 : \tau_2 \mid \Delta}{\Gamma \vdash (t_1, t_2) : \tau_1 \wedge \tau_2 \mid \Delta}\ \textsc{Pair}$$

$$\frac{c : (\Gamma \vdash \alpha : \tau, \Delta)}{\Gamma \vdash \mu\alpha.c : \tau \mid \Delta}\ \textsc{Mu}$$

Coterm typing

$$\frac{}{\Gamma \mid \alpha : \tau \vdash \alpha : \tau, \Delta}\ \textsc{Var}_\alpha$$

$$\frac{\Gamma \vdash t : \tau \mid \Delta \qquad \Gamma \mid s : \sigma \vdash \Delta}{\Gamma \mid t \cdot s : \tau \to \sigma \vdash \Delta}\ \textsc{App}$$

$$\frac{\Gamma \mid s : \tau_i \vdash \Delta}{\Gamma \mid \pi_1\, s : \tau_1 \wedge \tau_2 \vdash \Delta}\ \textsc{Proj}$$

$$\frac{c : (\Gamma, x : \tau \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c \vdash \Delta}\ \textsc{Mu}_\sim$$

Command typing

$$\frac{\Gamma \vdash t : \tau \mid \Delta \qquad \Gamma \mid s : \tau \vdash \Delta}{\langle\, t \mid s\,\rangle : (\Gamma \vdash \Delta)}\ \textsc{Cut}$$

6.3. Translating λ-terms to $\lambda\mu\tilde{\mu}$-terms

We introduce a compositional translation from λ-terms to $\lambda\mu\tilde{\mu}$-terms and show that it preserves typeability.

Definition 6.3.1. The *translation* $[\![-]\!] : \Lambda \to \Lambda_{\mu\tilde{\mu}}$ is defined as follows:

$$\begin{aligned}
[\![x]\!] &:\coloneqq x && (\mathcal{T}_1)\\
[\![\lambda x.e]\!] &:\coloneqq \lambda x.[\![e]\!] && (\mathcal{T}_2)\\
[\![(e_1, e_2)]\!] &:\coloneqq ([\![e_1]\!], [\![e_2]\!]) && (\mathcal{T}_3)\\
[\![e_1\ e_2]\!] &:\coloneqq \mu\alpha.\langle\, [\![e_1]\!] \mid [\![e_2]\!] \cdot \alpha\,\rangle && (\mathcal{T}_4)\\
[\![\pi_i\ e]\!] &:\coloneqq \mu\alpha.\langle\, [\![e]\!] \mid \pi_i\ \alpha\,\rangle && (\mathcal{T}_5)\\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] &:\coloneqq \mu\alpha.\langle\, [\![e_1]\!] \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha\,\rangle\,\rangle. && (\mathcal{T}_6)
\end{aligned}$$

In the last three clauses, the covariable α has to be fresh.

Let e be any expression of the λ-calculus typeable with type τ in a context Γ. Then the translation $[\![e]\!]$ is a term of the $\lambda\mu\tilde{\mu}$-calculus that is typeable with the same type τ (in the same context Γ of term variables and with an empty context of covariables).

Theorem 6.3.2. *For all e, τ and Γ: if $\Gamma \vdash e : \tau$, then $\Gamma \vdash [\![e]\!] : \tau \mid \emptyset$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash e : \tau$ in the λ-calculus. The cases for variables, tuples and λ-abstractions are trivial; we will only discuss the following interesting cases.

The first case is for projections. Assume that the last rule in the typing derivation of e is Proj. Then e has the form $\pi_i\, e$, whose translation is defined as $\mu\alpha.\langle\, [\![e]\!] \mid \pi_i\,\alpha \,\rangle$. We replace the λ-calulus derivation by the following $\lambda\mu\tilde{\mu}$-calculus derivation:

$$\dfrac{\dfrac{\begin{array}{c}\text{IH}\\ \Gamma \vdash [\![e]\!] : \tau_1 \wedge \tau_2 \mid \emptyset\end{array} \qquad \dfrac{\dfrac{}{\emptyset \mid \alpha : \tau_i \vdash \alpha : \tau_i}\,\textsc{Var}_\alpha}{\emptyset \mid \pi_i\,\alpha : \tau_1 \wedge \tau_2 \vdash \alpha : \tau_i}\,\textsc{Proj}}{\langle\, [\![e]\!] \mid \pi_i\,\alpha \,\rangle : (\Gamma \vdash \alpha : \tau_i)}\,\textsc{Cut}}{\Gamma \vdash \mu\alpha.\langle\, [\![e]\!] \mid \pi_i\,\alpha \,\rangle : \tau_i \mid \emptyset}\,\textsc{Mu}$$

The second interesting case is for function applications. Assume that the last rule in the derivation of $\Gamma \vdash e : \tau$ is App. Then e must have the form $e_1\, e_2$, whose translation is defined as $\mu\alpha.\langle\, [\![e_1]\!] \mid [\![e_2]\!] \cdot \alpha \,\rangle$. We replace the original derivation by:

$$\dfrac{\dfrac{\begin{array}{c}\text{IH}\\ \Gamma \vdash [\![e_1]\!] : \sigma \to \tau \mid \emptyset\end{array} \qquad \dfrac{\begin{array}{c}\text{IH}\\ \Gamma \vdash [\![e_2]\!] : \sigma \mid \emptyset\end{array} \qquad \dfrac{}{\emptyset \mid \alpha : \tau \vdash \alpha : \tau}\,\textsc{Var}_\alpha}{\emptyset \mid [\![e_2]\!] \cdot \alpha : \sigma \to \tau \vdash \alpha : \tau}\,\textsc{App}}{\langle\, [\![e_1]\!] \mid [\![e_2]\!] \cdot \alpha \,\rangle : (\Gamma \vdash \alpha : \tau)}\,\textsc{Cut}}{\Gamma \vdash \mu\alpha.\langle\, [\![e_1]\!] \mid [\![e_2]\!] \cdot \alpha \,\rangle : \tau \mid \emptyset}\,\textsc{Mu}$$

The last case is for let-bindings. Assume that the last rule in the derivation of $\Gamma \vdash e : \tau$ is Let. Then e must have the form **let** $x = e_1$ **in** e_2, whose translation is defined as $\mu\alpha.\langle\, [\![e_1]\!] \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha \,\rangle \,\rangle$. We replace the original derivation by:

$$\dfrac{\dfrac{\begin{array}{c}\text{IH}\\ \Gamma \vdash [\![e_1]\!] : \sigma \mid \emptyset\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\text{IH}\\ \Gamma, x : \sigma \vdash [\![e_2]\!] : \tau \mid \emptyset\end{array} \qquad \dfrac{}{\emptyset \mid \alpha : \tau \vdash \alpha : \tau}\,\textsc{Var}_\alpha}{\langle\, [\![e_2]\!] \mid \alpha \,\rangle : (\Gamma, x : \sigma \vdash \alpha : \tau)}\,\textsc{Cut}}{\Gamma \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha \,\rangle \vdash \alpha : \tau}\,\textsc{Mu}_{\sim}}{\langle\, [\![e_1]\!] \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha \,\rangle \,\rangle : (\Gamma \vdash \alpha : \tau)}\,\textsc{Cut}}{\Gamma \vdash \mu\alpha.\langle\, [\![e_1]\!] \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha \,\rangle \,\rangle : \tau \mid \emptyset}\,\textsc{Mu}$$

□

We will also need the following lemma about the translation of values:

Lemma 6.3.3 (Translation preserves values)**.** *An expression e is a value of the λ-calculus if, and only if, $[\![e]\!]$ is a value of the $\lambda\mu\tilde{\mu}$-calculus.*

Proof. By inspection of the relevant cases. □

6.4. Call-by-value operational semantics

We introduce the evaluation rules for the λ-calculus and for the $\lambda\mu\tilde{\mu}$-calculus.

6.4.1. Evaluation in the λ-calculus

For the λ-calculus we first have to define how to reduce immediate redexes. We do this in definition 6.4.1. One can note how all three rules implement the *call-by-value* strategy: a function application $(\lambda x.e_1)e_2$ can only be reduced if e_2 is a value; a projection $\pi_i(e_1, e_2)$ can only be reduced if both e_1 and e_2 are values; and a let-binding $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ can only be reduced if e_1 is a value.

Definition 6.4.1. The *call-by-value evaluation rules for the λ-calculus* are:

$$\begin{aligned} (\lambda x.e)\, v &\rhd e[v/x] && (\beta_{\to}) \\ \pi_i(v_1, v_2) &\rhd v_i && (\beta_{\wedge}) \\ \mathbf{let}\ x = v\ \mathbf{in}\ e &\rhd e[v/x]. && (\beta_{\mathrm{let}}) \end{aligned}$$

These rules are not sufficient, since none of the rules are applicable to the term $(\pi_1(v_1, v_2), v_3)$, for example. We therefore need to extend them to allow for the reduction of redexes within a term. Furthermore, since we want evaluations to be deterministic, we must extend definition 6.4.1 in such a way that there is always exactly one redex within a term which can be evaluated next. For example, in a tuple (e_1, e_2) we must specify whether we want to evaluate e_1 or e_2 first (and similarly for function applications $e_1\, e_2$).

We specify deterministic evaluations within a context by using the concept of evaluation contexts, introduced in Felleisen and Hieb, 1992. An evaluation context $E[-]$ is a term with one argument place marked by the symbol $\square$, which indicates where we evaluate the next immediate redex. Deterministic evaluation is ensured by a *unique decomposition lemma*:

> Every term e that is not a value can be uniquely decomposed into an evaluation context $E[-]$ and an immediate redex e' such that $e \doteq E[e']$.

Definition 6.4.2. The syntax of *evaluation contexts* $E[-]$ is defined as follows:

$$E[-] ::= \square \quad | \quad E\, e \quad | \quad v\, E \quad | \quad (E, e) \quad | \quad (v, E) \quad | \quad \mathbf{let}\ x = E\ \mathbf{in}\ e \quad | \quad \pi_i\, E.$$

Evaluation contexts now allow us to properly define evaluation within a context:

$$e \rhd e' \implies E[e] \rhd E[e']. \qquad \text{(Congruence)}$$

6.4.2. Evaluation in the $\lambda\mu\tilde{\mu}$-calculus

For the $\Lambda_{\mu\tilde{\mu}}$-calculus, we again first introduce the rules for evaluating immediate redexes. The choice of the *call-by-value* evaluation strategy is manifested in the following ways: first, a redex $\langle\, \lambda x.t \mid e \cdot s\,\rangle$ can only be reduced if the function argument e is a value; second, a redex $\langle\, (e_1, e_2) \mid \pi_i\, s\,\rangle$ can only be reduced if both e_1 and e_2 are values; third, the *critical pair* $\langle\, \mu\alpha.c_1 \mid \tilde{\mu}x.c_2\,\rangle$, which could a priori be reduced to either $c_1[\tilde{\mu}x.c_2/\alpha]$ or $c_2[\mu\alpha.c_2/x]$, is resolved by requiring in the rule $(\tilde{\mu})$ that a redex $\langle\, e \mid \tilde{\mu}x.c\,\rangle$ can only be reduced if e is a value.

Definition 6.4.3. The *call-by-value evaluation rules for the $\lambda\mu\tilde{\mu}$-calculus* are:

$$\langle\, \lambda x.t \mid v \cdot s \,\rangle \rhd \langle\, t[v/x] \mid s \,\rangle \tag{$\beta_\to$}$$
$$\langle\, (v_1, v_2) \mid \pi_i\, s \,\rangle \rhd \langle\, v_i \mid s \,\rangle \tag{$\beta_\wedge$}$$
$$\langle\, \mu\alpha.c \mid s \,\rangle \rhd c[s/\alpha] \tag{μ}$$
$$\langle\, v \mid \tilde{\mu}x.c \,\rangle \rhd c[v/x]. \tag{$\tilde{\mu}$}$$

These rules are, again, not complete. For example, there is no rule applicable to the cut $\langle\, (\mu\alpha.c, v) \mid \pi_i\, s \,\rangle$, since the first element of the tuple is not yet a value. Instead of the evaluation contexts $E[-]$, we will add focusing contexts $F[-]$ and *dynamic focusing rules* ς. The focusing contexts play the role of the evaluation contexts for the λ-calculus, while the ς-rules correspond to the rule (Congruence).

Definition 6.4.4. The syntax of *focusing contexts* $F[-]$ is defined as follows:

$$F[-] ::= (\Box, t) \quad\mid\quad (w, \Box) \quad\mid\quad \Box \cdot s.$$

Definition 6.4.5. We extend the evaluation rules of definition 6.4.3 by the following *dynamic focusing rules*:

$$\langle\, F[t] \mid s \,\rangle \rhd \langle\, t \mid \tilde{\mu}x.\langle\, F[x] \mid s \,\rangle \,\rangle \quad \text{(if } t \text{ is not a value)} \tag{ς_1}$$
$$\langle\, v \mid F[t] \,\rangle \rhd \langle\, t \mid \tilde{\mu}x.\langle\, v \mid F[x] \,\rangle \,\rangle \quad \text{(if } t \text{ is not a value).} \tag{ς_2}$$

6.5. The ANF-transformation

While the evaluation rules presented in section 6.4 are sufficient for purely theoretical investigations into the reduction theory of the λ-calculus and the $\lambda\mu\tilde{\mu}$-calculus, they are less ideal for other purposes. In particular, they are not ideal for generating efficient code that can be run on a real computer. For example, consider the congruence rule in definition 6.4.1. Its operational meaning implies that we have to *search* for the next redex in the term, and this redex can appear nested at an arbitrary depth within the term. If we implemented this search procedure naively for each reduction step, then the resulting program would be very inefficient indeed.

Various methods to efficiently evaluate terms of the λ-calculus have been proposed, for both the call-by-value and call-by-name evaluation orders. One of these methods is the compilation to *abstract machines*[2], like the SEK, SECD or Krivine machine, which provide much more efficient means of evaluating λ-terms. The evaluation of commands of the $\lambda\mu\tilde{\mu}$-calculus is, in fact, very similar to the evaluation of machine states of an abstract machine. Another class of methods for compiling terms of the ordinary λ-calculus is based on a translation into the so-called *continuation-passing style* (CPS), which was introduced in a seminal paper by John Charles Reynolds (1972). These

[2]For an introduction to the theory of abstract machines, cf. Felleisen, Findler, and Flatt, 2009.

translations have been studied both in logic, where they correspond to double negation translations (cf. Sørensen and Urzyczyn, 2006), and in the theory of optimizing compilers (cf. Appel, 1992).

One important variation of these CPS translations is the so-called *ANF-transformation*, which was introduced by Sabry and Felleisen (1992) and later elaborated by Flanagan, Sabry, B. F. Duba, and Felleisen (1993). We first introduce the syntax of the administrative normal form in definition 6.5.1. The ANF-transformation itself is introduced in Definitions 6.5.3 and 6.5.7.

Definition 6.5.1. The *syntax of the administrative normal form* Λ^{ANF} is defined as follows:

1. *Values:* $v ::= \lambda x.e \quad | \quad (v, v) \quad | \quad x.$
2. *Computations:* $c ::= v \quad | \quad v\, v \quad | \quad \pi_1\, v \quad | \quad \pi_2\, v.$
3. *Terms:* $e ::= c \quad | \quad \textbf{let}\ x = c\ \textbf{in}\ e.$

The administrative normal form has two characteristic properties. The first is reflected in the syntax of computations c: a projection π_i can only be applied to a value, and, similarly, a function application $v_1\, v_2$ can only be formed between two values. This excludes terms like $\pi_1(x, \pi_2(y, z))$ or $(\pi_1(f, g))(\pi_2(x, y))$. The second property is reflected in the syntax of terms e: a let-expression $\textbf{let}\ x = c\ \textbf{in}\ e$ can only bind the result of a computation c to a variable x. Let-expressions cannot bind other let-expressions, that is, expressions like $\textbf{let}\ x = (\textbf{let}\ y = e_1\ \textbf{in}\ e_2)\ \textbf{in}\ e_3$ are excluded by the second property.

Usual presentations of the ANF-transformation enforce both properties in a single transformation from Λ to Λ^{ANF}. Instead, we present the transformation to administrative normal form as a two-part transformation:

$$\Lambda \xrightarrow{\quad \mathcal{A} \quad} \Lambda^{\mathrm{Q}} \xrightarrow{\quad \mathcal{L} \quad} \Lambda^{\mathrm{ANF}}.$$

The first part consists of a function $\mathcal{A} : \Lambda \to \Lambda^{\mathrm{Q}}$ that enforces only the first of the two characteristic properties described above. A second transformation $\mathcal{L} : \Lambda^{\mathrm{Q}} \to \Lambda^{\mathrm{ANF}}$ then enforces the second property. By presenting the ANF-transformation in this way, we can make the relation to focusing clearer. In section 6.7 we will show that the first part of this transformation corresponds to focusing, whereas the second part of the transformation can be simulated by μ-reductions in $\Lambda_{\mu\tilde{\mu}}$.

Definition 6.5.2. The *syntax of the intermediate normal form* Λ^{Q} is defined as follows:

1. *Values:* $v ::= \lambda x.e \quad | \quad (v, v) \quad | \quad x.$
2. *Terms:* $e ::= v \quad | \quad \textbf{let}\ x = e\ \textbf{in}\ e \quad | \quad e\, v \quad | \quad \pi_1\, e \quad | \quad \pi_2\, e.$

Note that definition 6.5.2 only guarantees that pairs (v, v) consist of values, and that functions are always applied to values in a function application $e\, v$. The two transformations $\mathcal{A}$ and $\mathcal{L}$ are introduced in turn.

6.5.1. From Λ to Λ^{Q}

Recall that the first property that we want to enforce is that pairs consist of syntactic values, and that in function applications the function argument is already a value. The transformation $\mathcal{A}$ defined next guarantees the first property by binding any non-value argument which would violate this property to a fresh variable in a let-binding. For example, the term $\pi_1(\pi_2(x, y))$ is transformed by generating a fresh variable z, and binding the computation $\pi_2(x, y)$ to z in the computation $\pi_1\ z$: $\mathcal{A}(\pi_1(\pi_2(x, y))) :\triangleq$ **let** $z = \pi_2(x, y)$ **in** $\pi_1\ z$.

Definition 6.5.3. The *transformation* $\mathcal{A} : \Lambda \to \Lambda^{Q}$ is defined as follows:

$$\begin{aligned}
\mathcal{A}(x) &:\triangleq x && (\mathcal{A}_1)\\
\mathcal{A}(\lambda x.e) &:\triangleq \lambda x.\mathcal{A}(e) && (\mathcal{A}_2)\\
\mathcal{A}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) &:\triangleq \mathbf{let}\ x = \mathcal{A}(e_1)\ \mathbf{in}\ \mathcal{A}(e_2) && (\mathcal{A}_3)\\
\mathcal{A}(\pi_i\ e) &:\triangleq \pi_i(\mathcal{A}(e)) && (\mathcal{A}_4)
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}((v_1, v_2)) &:\triangleq (\mathcal{A}(v_1), \mathcal{A}(v_2)) && (\mathcal{A}_5)\\
\mathcal{A}((v_1, e_2)) &:\triangleq \mathbf{let}\ x = \mathcal{A}(e_2)\ \mathbf{in}\ (\mathcal{A}(v_1), x) && (\mathcal{A}_6)\\
\mathcal{A}((e_1, v_2)) &:\triangleq \mathbf{let}\ x = \mathcal{A}(e_1)\ \mathbf{in}\ (x, v_2) && (\mathcal{A}_7)\\
\mathcal{A}((e_1, e_2)) &:\triangleq \mathbf{let}\ x = \mathcal{A}(e_1)\ \mathbf{in}\ (\mathbf{let}\ y = \mathcal{A}(e_2)\ \mathbf{in}\ (x, y)) && (\mathcal{A}_8)
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}(e_1\ v_2) &:\triangleq \mathcal{A}(e_1)\ \mathcal{A}(v_2) && (\mathcal{A}_9)\\
\mathcal{A}(e_1\ e_2) &:\triangleq \mathbf{let}\ x = \mathcal{A}(e_2)\ \mathbf{in}\ \mathcal{A}(e_1)\ x. && (\mathcal{A}_{10})
\end{aligned}$$

Remark 6.5.4. Among the clauses of definition 6.5.3, the clauses $(\mathcal{A}_5)$ to $(\mathcal{A}_7)$ are subsumed by $(\mathcal{A}_8)$. Similarly, the clause $(\mathcal{A}_9)$ is subsumed by $(\mathcal{A}_{10})$. This redundancy is an optimization which guarantees that the transformation behaves as the identity function on terms that are already in Λ^{Q}.

Example 6.5.5. The result of the transformation

$$\mathcal{A}(\pi_1(\pi_1(\pi_1(x_1, x_2), x_3), x_4))$$

is the term

$$\mathbf{let}\ z_1 = (\mathbf{let}\ z_2 = \pi_1(x_1, x_2)\ \mathbf{in}\ \pi_1(z_2, x_3))\ \mathbf{in}\ \pi_1(z_1, x_4),$$

where z_1 and z_2 are variables that are generated during the transformation. This example shows that the result of $\mathcal{A}$ is, in general, not yet in Λ^{ANF}.

6.5.2. From Λ^{Q} to Λ^{ANF}

The second property which we want to enforce is that in a let-construct **let** $x = c$ **in** e the computation bound to the variable x must be of a restricted form. This will be guaranteed by the transformation $\mathcal{L}$, given by definition 6.5.7. In order to define this transformation, we need to define a meta-level operation @ that operates on continuations k and values v from Λ^{ANF}:

Definition 6.5.6. *Continuations* are defined as follows:

$$k ::= \mathrm{id} \quad | \quad \overline{\lambda v}.\mathbf{let}\ x = \pi_i\,\overline{v}\ \mathbf{in}\ e \quad | \quad \overline{\lambda v}.\mathbf{let}\ x = \overline{v}\,v\ \mathbf{in}\ e \quad | \quad \overline{\lambda v}.\mathbf{let}\ x = \overline{v}\ \mathbf{in}\ e,$$

where e and v range over expressions and values from Λ^{ANF}.

The *meta-level operation* @ takes a continuation k and a value v from Λ^{ANF} and returns an expression of Λ^{ANF}. It is evaluated as follows:

$$\mathrm{id}\mathbin{@}v \mathrel{:\hat{=}} v \tag{@$_1$}$$

$$\overline{\lambda v}.\mathbf{let}\ x = \pi_i\,\overline{v}\ \mathbf{in}\ e\mathbin{@}v \mathrel{:\hat{=}} \mathbf{let}\ x = \pi_i\,v\ \mathbf{in}\ e \tag{@$_2$}$$

$$\overline{\lambda v}.\mathbf{let}\ x = \overline{v}\,v_2\ \mathbf{in}\ e\mathbin{@}v_1 \mathrel{:\hat{=}} \mathbf{let}\ x = v_1\,v_2\ \mathbf{in}\ e \tag{@$_3$}$$

$$\overline{\lambda v}.\mathbf{let}\ x = \overline{v}\ \mathbf{in}\ e\mathbin{@}v \mathrel{:\hat{=}} \mathbf{let}\ x = v\ \mathbf{in}\ e. \tag{@$_4$}$$

Using this technical tool we can now define the transformation $\mathcal{L}$.

Definition 6.5.7. The *transformation* $\mathcal{L} : \Lambda^{Q} \to \Lambda^{ANF}$ is given as follows:

Values

$$\mathcal{L}(x) \mathrel{:\hat{=}} x \tag{$\mathcal{L}_1$}$$

$$\mathcal{L}(\lambda x.e) \mathrel{:\hat{=}} \lambda x.\mathcal{L}_{\mathrm{id}}(e) \tag{$\mathcal{L}_2$}$$

$$\mathcal{L}((v_1, v_2)) \mathrel{:\hat{=}} (\mathcal{L}(v_1), \mathcal{L}(v_2)) \tag{$\mathcal{L}_3$}$$

Terms

$$\mathcal{L}(e) \mathrel{:\hat{=}} \mathcal{L}_{\mathrm{id}}(e) \tag{$\mathcal{L}_4$}$$

$$\mathcal{L}_k(e_1\,v_2) \mathrel{:\hat{=}} \mathcal{L}_{\overline{\lambda v}.\mathbf{let}\ x=\overline{v}\,\mathcal{L}(v_2)\ \mathbf{in}\ k\mathbin{@}x}(e_1) \tag{$\mathcal{L}_5$}$$

$$\mathcal{L}_k(\pi_i\,e) \mathrel{:\hat{=}} \mathcal{L}_{\overline{\lambda v}.\mathbf{let}\ x=\pi_i\,\overline{v}\ \mathbf{in}\ k\mathbin{@}x}(e) \tag{$\mathcal{L}_6$}$$

$$\mathcal{L}_k(v) \mathrel{:\hat{=}} k\mathbin{@}\mathcal{L}(v) \tag{$\mathcal{L}_7$}$$

$$\mathcal{L}_k(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \mathrel{:\hat{=}} \mathcal{L}_{\overline{\lambda v}.\mathbf{let}\ x=\overline{v}\ \mathbf{in}\ \mathcal{L}_k(e_2)}(e_1). \tag{$\mathcal{L}_8$}$$

Example 6.5.8. As an example of the transformation $\mathcal{L}$, consider the term (from example 6.5.5)

$$\mathbf{let}\ z_1 = (\mathbf{let}\ z_2 = \pi_1(x_1, x_2)\ \mathbf{in}\ \pi_1(z_2, x_3))\ \mathbf{in}\ \pi_1(z_1, x_4),$$

which can be transformed into Λ^{ANF} as follows:

$$\begin{aligned}
&\mathcal{L}_{\mathrm{id}}(\mathbf{let}\ z_1 = (\mathbf{let}\ z_2 = \pi_1(x_1, x_2)\ \mathbf{in}\ \pi_1(z_2, x_3))\ \mathbf{in}\ \pi_1(z_1, x_4))\\
&\quad = \mathcal{L}_{\overline{\lambda v_1}.\mathbf{let}\ z_1=\overline{v_1}\ \mathbf{in}\ \mathcal{L}_{\mathrm{id}}(\pi_1(z_1,x_4))}(\mathbf{let}\ z_2 = \pi_1(x_1, x_2)\ \mathbf{in}\ \pi_1(z_2, x_3))\\
&\quad = \mathcal{L}_{\overline{\lambda v_1}.\mathbf{let}\ z_1=\overline{v_1}\ \mathbf{in}\ \pi_1(z_1,x_4)}(\mathbf{let}\ z_2 = \pi_1(x_1, x_2)\ \mathbf{in}\ \pi_1(z_2, x_3))\\
&\quad = \mathcal{L}_{\overline{\lambda v_2}.\mathbf{let}\ z_2=\overline{v_2}\ \mathbf{in}\ \mathcal{L}_{\overline{\lambda v_1}.\mathbf{let}\ z_1=\overline{v_1}\ \mathbf{in}\ \pi_1(z_1,x_4)}(\pi_1(z_2,x_3))}(\pi_1(x_1, x_2))\\
&\quad = \mathcal{L}_{\overline{\lambda v_2}.\mathbf{let}\ z_2=\overline{v_2}\ \mathbf{in}\ (\mathbf{let}\ z_1=\pi_1(z_2,x_3)\ \mathbf{in}\ \pi_1(z_1,x_4))}(\pi_1(x_1, x_2))\\
&\quad = \mathbf{let}\ z_2 = \pi_1(x_1, x_2)\ \mathbf{in}\ (\mathbf{let}\ z_1 = \pi_1(z_2, x_3)\ \mathbf{in}\ \pi_1(z_1, x_4)).
\end{aligned}$$

The term was transformed into Λ^{ANF} by (in a certain way) moving the let-binding of z_2 to the outside of the let-binding of z_1.

6.6. The focusing transformation

In distinction to the dynamic focusing rules of definition 6.4.5, we now consider only static focusing. We first introduce the focused subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ as a subset of $\Lambda_{\mu\tilde{\mu}}$ (definition 6.2.4).[3]

Definition 6.6.1. The *focused subsyntax* $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ for the call-by-value evaluation strategy is defined as follows:

1. *Terms:* $t ::= w \quad | \quad \mu\alpha.c.$
2. *Coterms:* $s ::= \alpha \quad | \quad w \cdot s \quad | \quad \pi_1\, s \quad | \quad \pi_2\, s \quad | \quad \tilde{\mu}x.c.$
3. *Commands:* $c ::= \langle\, t \mid s \,\rangle.$
4. *Values:* $w ::= \lambda x.t \quad | \quad (w, w) \quad | \quad x.$

The focused subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ differs in two respects from $\Lambda_{\mu\tilde{\mu}}$. First, terms t must now either be values w or abstractions $\mu\alpha.c$. This excludes terms like $(\mu\alpha.c, t)$ and $(t, \mu\alpha.c)$ from the subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$, which are part of the syntax of terms of definition 6.2.4. This corresponds precisely to the restriction that constructors can only be applied to values. Second, the syntax of coterms has been changed by requiring the function argument in a coterm $t \cdot s$ to be a value; that is, we require $w \cdot s$. This corresponds to the requirement that functions can syntactically only be applied to values.

Lemma 6.6.2. *For any term $e \in \Lambda^{Q}$, $[\![e]\!] \in \Lambda^{Q}_{\mu\tilde{\mu}}$.*

Proof. By induction on e.

1. Case $e \doteq$ **let** $x = e_1$ **in** e_2: the translation of e is $\mu\alpha.\langle\, [\![e_1]\!] \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha \,\rangle \,\rangle$. Using the induction hypothesis for e_1 and e_2, this term is in the subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$.
2. If e is of the form $e_1 v_2$, then the translation of e is $\mu\alpha.\langle\, [\![e_1]\!] \mid [\![v_2]\!]\cdot\alpha \,\rangle$. By lemma 6.3.3, $[\![v_2]\!]$ is a value, and by the induction hypothesis both $[\![e_1]\!]$ and $[\![v_2]\!]$ are in the subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$, so the resulting term is in the subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$.
3. If e is of the form $\pi_i e_1$, then $[\![\pi_i e_1]\!]$ is $\mu\alpha.\langle\, [\![e_1]\!] \mid \pi_i\,\alpha \,\rangle$. Using the induction hypothesis for e_1, $[\![e_1]\!]$ is in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$. Therefore $[\![\pi_i\, e_1]\!]$ is also in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$.
4. If $e \doteq v$, then we have to distinguish the following cases:
   a) If $v \doteq x$, then $[\![x]\!] \doteq x$, which is in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$.

[3] $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ corresponds to the subsyntax LKQ defined in Curien and Herbelin, 2000.

b) If $v \doteq (v_1, v_2)$, then $[\![(v_1, v_2)]\!] \doteq ([\![v_1]\!], [\![v_2]\!])$. By lemma 6.3.3, both $[\![v_i]\!]$ are values, and by the induction hypothesis they are in the subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$. Therefore $[\![v]\!]$ is also in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$.

c) If $v \doteq \lambda x.e$, then $[\![\lambda x.e]\!] \doteq \lambda x.[\![e]\!]$. By the induction hypothesis $[\![e]\!]$ is in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$, therefore $[\![v]\!]$ is also in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$. □

The subsyntax does not restrict the set of derivable sequents, since any term, coterm or command in the unrestricted syntax can be translated into the focused subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ by using the following *static* focusing transformation.

Definition 6.6.3. The *static focusing transformation* $\mathcal{F} : \Lambda_{\mu\tilde{\mu}} \to \Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ is defined as follows:

Terms

$$\mathcal{F}(x) :\doteq x \tag{$\mathcal{F}_1$}$$
$$\mathcal{F}(\mu\alpha.c) :\doteq \mu\alpha.\mathcal{F}(c) \tag{$\mathcal{F}_2$}$$
$$\mathcal{F}(\lambda x.e) :\doteq \lambda x.\mathcal{F}(e) \tag{$\mathcal{F}_3$}$$
$$\mathcal{F}((w_1, w_2)) :\doteq (\mathcal{F}(w_1), \mathcal{F}(w_2)) \tag{$\mathcal{F}_4$}$$
$$\mathcal{F}((w_1, t_2)) :\doteq \mu\alpha.\langle\, \mathcal{F}(t_2) \mid \tilde{\mu}x.\langle\, (\mathcal{F}(w_1), x) \mid \alpha \,\rangle\,\rangle \tag{$\mathcal{F}_5$}$$
$$\mathcal{F}((t_1, w_2)) :\doteq \mu\alpha.\langle\, \mathcal{F}(t_1) \mid \tilde{\mu}x.\langle\, (x, \mathcal{F}(w_2)) \mid \alpha \,\rangle\,\rangle \tag{$\mathcal{F}_6$}$$
$$\mathcal{F}((t_1, t_2)) :\doteq \mu\alpha.\langle \mathcal{F}(t_1) \mid \tilde{\mu}x.\langle \mu\beta.\langle \mathcal{F}(t_2) \mid \tilde{\mu}y.\langle (x, y) \mid \beta \rangle\rangle \mid \alpha \rangle\rangle \tag{$\mathcal{F}_7$}$$

Coterms

$$\mathcal{F}(\alpha) :\doteq \alpha \tag{$\mathcal{F}_8$}$$
$$\mathcal{F}(\tilde{\mu}x.c) :\doteq \tilde{\mu}x.\mathcal{F}(c) \tag{$\mathcal{F}_9$}$$
$$\mathcal{F}(\pi_i\, s) :\doteq \pi_i\, \mathcal{F}(s) \tag{$\mathcal{F}_{10}$}$$
$$\mathcal{F}(w \cdot s) :\doteq \mathcal{F}(w) \cdot \mathcal{F}(s) \tag{$\mathcal{F}_{11}$}$$
$$\mathcal{F}(t \cdot s) :\doteq \tilde{\mu}x.\langle\, \mathcal{F}(t) \mid \tilde{\mu}y.\langle\, x \mid y \cdot \mathcal{F}(s) \,\rangle\,\rangle \tag{$\mathcal{F}_{12}$}$$

Commands

$$\mathcal{F}(\langle\, t \mid s \,\rangle) :\doteq \langle\, \mathcal{F}(t) \mid \mathcal{F}(s) \,\rangle \tag{$\mathcal{F}_{13}$}$$
$$\mathcal{F}(\langle\, t_1 \mid t_2 \cdot s \,\rangle) :\doteq \langle\, \mathcal{F}(t_2) \mid \tilde{\mu}x.\langle\, \mu\alpha.\langle\, \mathcal{F}(t_1) \mid x \cdot \alpha \,\rangle \mid \mathcal{F}(s) \,\rangle\,\rangle. \tag{$\mathcal{F}_{14}$}$$

In general, when several clauses are applicable, the most specific clause should be applied. The clauses ($\mathcal{F}_4$), ($\mathcal{F}_5$) and ($\mathcal{F}_6$) are subsumed by the more general clause ($\mathcal{F}_7$), and ($\mathcal{F}_{11}$) is subsumed by the clause ($\mathcal{F}_{12}$). The presence of these additional clauses guarantees that $\mathcal{F}$ behaves as the identity function when it is applied to a term, coterm or command that is already in the subsyntax $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$. With these optimizations, our definition corresponds to the one given in Downen and Zena M. Ariola, 2018a, Fig. 18, with the exception of the clause ($\mathcal{F}_{14}$). The additional clause ($\mathcal{F}_{14}$) is necessary to guarantee that the functions $[\![-]\!]$, $\mathcal{A}$ and $\mathcal{F}$ commute up to α-equivalence, as shown by theorem 6.7.1. Without the clause ($\mathcal{F}_{14}$), theorem 6.7.1 has to be slightly weakened to theorem 6.7.2.

Lemma 6.6.4 ($\mathcal{F}$ preserves typeability). *For all terms t, coterms s and commands c:*

1. *If $\Gamma \vdash t : \tau \mid \Delta$, then $\Gamma \vdash \mathcal{F}(t) : \tau \mid \Delta$.*
2. *If $\Gamma \mid s : \tau \vdash \Delta$, then $\Gamma \mid \mathcal{F}(s) : \tau \vdash \Delta$.*
3. *If $c : (\Gamma \vdash \Delta)$, then $\mathcal{F}(c) : (\Gamma \vdash \Delta)$.*

Proof. By simultaneous structural induction on t, s and c, respectively. □

6.7. The main result

As explained in section 6.5, the ANF-transformation can be split into a purely local transformation $\mathcal{A}$ and a global transformation $\mathcal{L}$. We show what these two parts correspond to in the $\lambda\mu\tilde{\mu}$-calculus, and prove how the ANF-transformation on λ-terms relates to static focusing of $\lambda\mu\tilde{\mu}$-terms.

6.7.1. The correspondence between $\mathcal{A}$ and $\mathcal{F}$

Theorem 6.7.1 (Focusing reflects the ANF-transformation). *For all λ-terms e, we have $\mathcal{F}(\llbracket e \rrbracket) \doteq \llbracket \mathcal{A}(e) \rrbracket$.*

Proof. See section 6.9. □

If we omit the focusing rule ($\mathcal{F}_{14}$) from definition 6.6.3, then theorem 6.7.1 no longer holds up to syntactic equality ($\doteq$). Instead, the following weaker result (theorem 6.7.2) holds for $\eta\mu$-equality $\equiv$, which includes η-equivalence

$$\tilde{\mu}x.\langle\, x \mid s \,\rangle \equiv_\eta s \qquad \text{(for } x \text{ not free in } s\text{).}$$

Theorem 6.7.2 (Focusing reflects the ANF-transformation; case $\equiv$). *For all λ-terms e, we have $\mathcal{F}(\llbracket e \rrbracket) \equiv \llbracket \mathcal{A}(e) \rrbracket$.*

Proof. See section 6.9. □

6.7.2. Simulating $\mathcal{L}$ in the $\lambda\mu\tilde{\mu}$-calculus

Our main contention in this section is that a special purpose transformation like $\mathcal{L}$ is not necessary in $\Lambda_{\mu\tilde{\mu}}$. In order to transform from $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$ to $\Lambda^{\mathrm{ANF}}_{\mu\tilde{\mu}}$ we only have to apply μ-reductions and $\tilde{\mu}$-expansions. More concretely, the effect that $\mathcal{L}$ has on a term, namely to globally reorganize the ordering of let-bindings, can be simulated by simply reducing μ-redexes in the image of the translation. In order to illustrate this central point, let us come back to Examples 6.5.5 and 6.5.8. Recall that we showed in example 6.5.8 that $\mathcal{L}$ has the effect of changing the order of the two let-bindings of z_1 and z_2:

$$\begin{aligned}&\mathcal{L}(\textbf{let } z_1 = (\textbf{let } z_2 = \pi_1(x_1,x_2) \textbf{ in } \pi_1(z_2,x_3)) \textbf{ in } \pi_1(z_1,x_4))\\ &\qquad\qquad\qquad \doteq \textbf{let } z_2 = \pi_1(x_1,x_2) \textbf{ in } (\textbf{let } z_1 = \pi_1(z_2,x_3) \textbf{ in } \pi_1(z_1,x_4)).\end{aligned}$$

This can be simulated as follows:

$$\begin{aligned}&[\![\textbf{let } z_1 = (\textbf{let } z_2 = \pi_1(x_1,x_2) \textbf{ in } \pi_1(z_2,x_3)) \textbf{ in } \pi_1(z_1,x_4)]\!]\\ &\doteq \mu\alpha.\underline{\langle\, \mu\beta.\langle\, [\![\pi_1(x_1,x_2)]\!] \mid \tilde{\mu}z_2.\langle\, [\![\pi_1(z_2,x_3)]\!] \mid \beta\,\rangle\,\rangle \mid \tilde{\mu}z_1.\langle [\![\pi_1(z_1,x_4)]\!] \mid \alpha\rangle\,\rangle}\\ &\triangleright \mu\alpha.\langle\, [\![\pi_1(x_1,x_2)]\!] \mid \tilde{\mu}z_2.\langle\, [\![\pi_1(z_2,x_3)]\!] \mid \tilde{\mu}z_1.\langle\, [\![\pi_1(z_1,x_4)]\!] \mid \alpha\,\rangle\,\rangle\,\rangle\\ &\triangleleft \mu\alpha.\langle\, [\![\pi_1(x_1,x_2)]\!] \mid \tilde{\mu}z_2.\underline{\langle\, \mu\beta.\langle\, [\![\pi_1(z_2,x_3)]\!] \mid \tilde{\mu}z_1.\langle\, [\![\pi_1(z_1,x_4)]\!] \mid \beta\,\rangle\,\rangle \mid \alpha\,\rangle}\,\rangle\\ &\doteq [\![\textbf{let } z_2 = \pi_1(x_1,x_2) \textbf{ in } (\textbf{let } z_1 = \pi_1(z_2,x_3) \textbf{ in } \pi_1(z_1,x_4))]\!].\end{aligned}$$

Next, we define the subsyntax $\Lambda_{\mu\tilde{\mu}}^{\mathrm{ANF}}$, which differs from $\Lambda_{\mu\tilde{\mu}}^{\mathrm{Q}}$ (definition 6.6.1) in two aspects. First, commands are now required to consist of a *value* and a coterm instead of a term and a coterm, i.e., they do not contain any μ-redexes. Second, the coterms for projections and function applications are required to give an explicit name to the value they bind in the coterm they contain, i.e., they are $\tilde{\mu}$-expanded.

Definition 6.7.3. The *focused subsyntax* $\Lambda_{\mu\tilde{\mu}}^{\mathrm{ANF}}$ for the call-by-value strategy is defined as follows:

1. *Terms:* $t ::= w \quad | \quad \mu\alpha.c.$

2. *Coterms:* $s ::= \alpha \quad | \quad w \cdot \tilde{\mu}x.c \quad | \quad \pi_1\, \tilde{\mu}x.c \quad | \quad \pi_2\, \tilde{\mu}x.c \quad | \quad \tilde{\mu}x.c$

3. *Commands:* $c ::= \langle\, w \mid s\,\rangle.$

4. *Values:* $w ::= \lambda x.t \quad | \quad (w,w) \quad | \quad x.$

We have to refine definition 6.3.1.

Definition 6.7.4. The *refined translation* $[\![-]\!]^* : \Lambda^{\mathrm{ANF}} \to \Lambda_{\mu\tilde{\mu}}^{\mathrm{ANF}}$ is defined as the first function in the following set of mutually defined recursive functions:

First function (on expressions)

$$*[\![e]\!]^* :\simeq \mu\alpha.[\![e]\!]^*_\alpha \qquad (\mathcal{T}_1^*)$$

Second function (on expressions)

$$*[\![\textbf{let } x = c \textbf{ in } e]\!]^*_s :\simeq [\![c]\!]^*_{\tilde{\mu}x.[\![e]\!]^*_s} \qquad (\mathcal{T}_2^*)$$

$$[\![v_1\ v_2]\!]^*_s :\simeq \langle\, [\![v_1]\!]^* \mid [\![v_2]\!]^* \cdot s \,\rangle \qquad (\mathcal{T}_3^*)$$

$$[\![\pi_i\ v]\!]^*_s :\simeq \langle\, [\![v]\!]^* \mid \pi_i\ s \,\rangle \qquad (\mathcal{T}_4^*)$$

$$[\![v]\!]^*_s :\simeq \langle\, [\![v]\!]^* \mid s \,\rangle \qquad (\mathcal{T}_5^*)$$

Third function (on values)

$$*[\![x]\!]^* :\simeq x \qquad (\mathcal{T}_6^*)$$

$$[\![(v_1, v_2)]\!]^* :\simeq ([\![v_1]\!]^*, [\![v_2]\!]^*) \qquad (\mathcal{T}_7^*)$$

$$[\![\lambda x.e]\!]^* :\simeq \lambda x.[\![e]\!]^*. \qquad (\mathcal{T}_8^*)$$

Lemma 6.7.5. *For all terms $e \in \Lambda^{ANF}$, $[\![e]\!]^* \in \Lambda^{ANF}_{\mu\tilde{\mu}}$.*

Proof. By induction on terms e. □

Theorem 6.7.6. *For all terms $e \in \Lambda^Q$, $[\![\mathcal{L}(e)]\!]^* \equiv_\mu [\![e]\!]$.*

Proof. See section 6.9. □

6.8. Summary and outlook

We can summarize our results in the following diagram, where both the lower and the upper part are commutative.

$$\begin{tikzcd}
\Lambda \text{ (Def. 6.2.2)} \arrow[r, "\llbracket - \rrbracket"] \arrow[d, "\mathcal{A} \text{ (Def. 6.5.3)}"'] \arrow[dr, phantom, "\text{(theorem 6.7.1)}"] & \Lambda_{\mu\tilde{\mu}} \text{ (Def. 6.2.4)} \arrow[d, "\mathcal{F} \text{ (Def. 6.6.3)}"] \\
\Lambda^{\mathrm{Q}} \text{ (Def. 6.5.2)} \arrow[r, "\llbracket - \rrbracket"] \arrow[d, "\mathcal{L} \text{ (Def. 6.5.7)}"'] \arrow[dr, phantom, "\text{(theorem 6.7.6)}"] & \Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}} \text{ (Def. 6.6.1)} \arrow[d, "\mu\text{-reduction}"] \\
\Lambda^{\mathrm{ANF}} \text{ (Def. 6.5.1)} \arrow[r, "\llbracket - \rrbracket^*"] & \Lambda^{\mathrm{ANF}}_{\mu\tilde{\mu}} \text{ (Def. 6.7.3)}
\end{tikzcd}$$

These results are embedded in a wider conceptual context. By the Curry-Howard correspondence, natural deduction for intuitionistic logic (more precisely, the $\{\to, \wedge\}$-fragment) corresponds to the λ-calculus on the one side, and the sequent calculus for classical logic corresponds to the $\lambda\mu\tilde{\mu}$-calculus on the other side. Our results thus establish a bridge between natural deduction for intuitionistic logic with its computational interpretation on the one side and the classical sequent calculus with its computational interpretation on the other side.

We would like to extend this work in two directions. The first concerns the asymmetry of the translation $\llbracket - \rrbracket$, which is only a mapping from Λ to $\Lambda_{\mu\tilde{\mu}}$, but not vice versa. In order to provide a translation in the opposite direction, from $\Lambda_{\mu\tilde{\mu}}$ to Λ, we will have to extend the λ-calculus with control operators. The seond extension concerns the treatment of evaluation orders other than call-by-value. While the treatment of call-by-name seems to be straightforward, the study of call-by-need (cf. Zena M. Ariola, Maraist, Odersky, Felleisen, and Wadler, 1995; Launchbury, 1993) and its dual call-by-co-need (cf. Downen and Zena M. Ariola, 2018b) in both the λ-calculus and the $\lambda\mu\tilde{\mu}$-calculus seems to be promising.

6.9. Proof of the main theorems

Theorem 6.7.1 (Focusing reflects the ANF-transformation). *For all λ-terms e, we have $\mathcal{F}(\llbracket e \rrbracket) \doteq \llbracket \mathcal{A}(e) \rrbracket$.*

Proof. By induction on the structure of e.

1. Case $e \doteq x$: $\mathcal{F}(\llbracket x \rrbracket) \doteq x \doteq \llbracket \mathcal{A}(x) \rrbracket$.

2. Case $e \doteq \lambda x.e_1$:

$$\mathcal{F}(\llbracket \lambda x.e_1 \rrbracket) \doteq \lambda x.\mathcal{F}(\llbracket e_1 \rrbracket) \overset{\text{IH}}{\doteq} \lambda x.\llbracket \mathcal{A}(e_1) \rrbracket \doteq \llbracket \mathcal{A}(\lambda x.e_1) \rrbracket.$$

3. Case $e \doteq \pi_i\, e_1$:

$$\begin{aligned}
\mathcal{F}([\![\pi_i\, e_1]\!]) &\doteq \mathcal{F}(\mu\alpha.\langle\, [\![e_1]\!] \mid \pi_i\, \alpha\,\rangle) && (\mathcal{T}_5)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_1]\!]) \mid \pi_i\, \alpha\,\rangle && (\mathcal{F}_2,\, \mathcal{F}_{13},\, \mathcal{F}_{10},\, \mathcal{F}_8)\\
&\doteq \mu\alpha.\langle\, [\![\mathcal{A}(e_1)]\!] \mid \pi_i\, \alpha\,\rangle && (\text{IH})\\
&\doteq [\![\pi_i\, \mathcal{A}(e_1)]\!] && (\mathcal{T}_5)\\
&\doteq [\![\mathcal{A}(\pi_i\, e_1)]\!]. && (\mathcal{A}_4)
\end{aligned}$$

4. In case $e \doteq e_1\, e_2$, we have to distinguish two subcases:
   (i) Subcase $e \doteq e_1\, v_2$:

$$\begin{aligned}
\mathcal{F}([\![e_1\, v_2]\!]) &\doteq \mathcal{F}(\mu\alpha.\langle\, [\![e_1]\!] \mid [\![v_2]\!] \cdot \alpha\,\rangle) && (\mathcal{T}_4)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_1]\!]) \mid \mathcal{F}([\![v_2]\!]) \cdot \alpha\,\rangle && (\mathcal{F}_2,\, \mathcal{F}_{13},\, \mathcal{F}_{11},\, \mathcal{F}_8)\\
&\doteq \mu\alpha.\langle\, [\![\mathcal{A}(e_1)]\!] \mid [\![\mathcal{A}(v_2)]\!] \cdot \alpha\,\rangle && (\text{IH})\\
&\doteq [\![\mathcal{A}(e_1)\, \mathcal{A}(v_2)]\!] && (\mathcal{T}_4)\\
&\doteq [\![\mathcal{A}(e_1\, v_2)]\!]. && (\mathcal{A}_9)
\end{aligned}$$

   (ii) Subcase $e \doteq e_1\, e_2$:

$$\begin{aligned}
\mathcal{F}([\![(e_1\, e_2)]\!]) &\doteq \mathcal{F}(\mu\alpha.\langle\, [\![e_1]\!] \mid [\![e_2]\!] \cdot \alpha\,\rangle) && (\mathcal{T}_4)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_2]\!]) \mid \tilde{\mu}x.\langle\, \mu\beta.\langle\, \mathcal{F}([\![e_1]\!]) \mid x \cdot \beta\,\rangle \mid \alpha\,\rangle\,\rangle && (\mathcal{F}_{14},\, \mathcal{F}_2,\, \mathcal{F}_8)\\
&\doteq \mu\alpha.\langle [\![\mathcal{A}(e_2)]\!] \;\mid\; \tilde{\mu}x.\langle \mu\beta.\langle [\![\mathcal{A}(e_1)]\!] \;\mid\; x \cdot \beta\rangle \;\mid\; \alpha\rangle\rangle && (\text{IH})\\
&\doteq [\![\mathbf{let}\ x = \mathcal{A}(e_2)\ \mathbf{in}\ \mathcal{A}(e_1)\, x]\!] && (\mathcal{T}_4,\, \mathcal{T}_6)\\
&\doteq [\![\mathcal{A}(e_1\, e_2)]\!]. && (\mathcal{A}_{10})
\end{aligned}$$

5. In case $e \doteq (e_1, e_2)$, we have to distinguish four subcases:
   (i) Subcase $e \doteq (v_1, v_2)$:

$$\begin{aligned}
\mathcal{F}([\![(v_1, v_2)]\!]) &\doteq \mathcal{F}([\![v_1]\!], [\![v_2]\!]) && (\mathcal{T}_3)\\
&\doteq (\mathcal{F}([\![v_1]\!]), \mathcal{F}([\![v_2]\!])) && (\mathcal{F}_4)\\
&\doteq ([\![\mathcal{A}(v_1)]\!], [\![\mathcal{A}(v_2)]\!]) && (\text{IH})\\
&\doteq [\![(\mathcal{A}(v_1), \mathcal{A}(v_2))]\!] && (\mathcal{T}_3)\\
&\doteq [\![\mathcal{A}((v_1, v_2))]\!]. && (\mathcal{A}_5)
\end{aligned}$$

   (ii) Subcase $e \doteq (v_1, e_2)$:

$$\begin{aligned}
\mathcal{F}([\![(v_1, e_2)]\!]) &\doteq \mathcal{F}(([\![v_1]\!], [\![e_2]\!])) && (\mathcal{T}_3)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_2]\!]) \mid \tilde{\mu}x.\langle\, (\mathcal{F}([\![v_1]\!]), x) \mid \alpha\,\rangle\,\rangle && (\mathcal{F}_5)\\
&\doteq \mu\alpha.\langle\, [\![\mathcal{A}(e_2)]\!] \mid \tilde{\mu}x.\langle\, ([\![\mathcal{A}(v_1)]\!], x) \mid \alpha\,\rangle\,\rangle && (\text{IH})\\
&\doteq [\![\mathbf{let}\ x = \mathcal{A}(e_2)\ \mathbf{in}\ (\mathcal{A}(v_1), x)]\!] && (\mathcal{T}_3,\, \mathcal{T}_6)\\
&\doteq [\![\mathcal{A}((v_1, e_2))]\!]. && (\mathcal{A}_6)
\end{aligned}$$

(iii) Subcase $e \doteq (e_1, v_2)$:

$$\begin{aligned}
\mathcal{F}([\![(e_1, v_2)]\!]) &\doteq \mathcal{F}(([\![e_1]\!], [\![v_2]\!])) && (\mathcal{T}_3)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_1]\!]) \mid \tilde{\mu}x.\langle\, (x, \mathcal{F}([\![v_2]\!])) \mid \alpha \,\rangle\,\rangle && (\mathcal{F}_6)\\
&\doteq \mu\alpha.\langle\, [\![\mathcal{A}(e_1)]\!] \mid \tilde{\mu}x.\langle\, (x, [\![\mathcal{A}(v_2)]\!]) \mid \alpha \,\rangle\,\rangle && (\text{IH})\\
&\doteq [\![\mathbf{let}\ x = \mathcal{A}(e_1)\ \mathbf{in}\ (x, \mathcal{A}(v_2))]\!] && (\mathcal{T}_3,\ \mathcal{T}_6)\\
&\doteq [\![\mathcal{A}((e_1, v_2))]\!]. && (\mathcal{A}_7)
\end{aligned}$$

(iv) Subcase $e \doteq (e_1, e_2)$:

$$\begin{aligned}
&\mathcal{F}([\![(e_1, e_2)]\!])\\
&\doteq \mathcal{F}(([\![e_1]\!], [\![e_2]\!])) && (\mathcal{T}_3)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_1]\!]) \mid \tilde{\mu}x.\langle\, \mu\beta.\langle\, \mathcal{F}([\![e_2]\!]) \mid \tilde{\mu}y.\langle\, (x, y) \mid \beta \,\rangle\,\rangle \mid \alpha \,\rangle\,\rangle && (\mathcal{F}_7)\\
&\doteq \mu\alpha.\langle\, [\![\mathcal{A}(e_1)]\!] \mid \tilde{\mu}x.\langle\, \mu\beta.\langle\, [\![\mathcal{A}(e_2)]\!] \mid \tilde{\mu}y.\langle\, (x, y) \mid \beta \,\rangle\,\rangle \mid \alpha \,\rangle\,\rangle && (\text{IH})\\
&\doteq [\![\mathbf{let}\ x = \mathcal{A}(e_1)\ \mathbf{in}\ (\mathbf{let}\ y = \mathcal{A}(e_2)\ \mathbf{in}\ (x, y))]\!] && (\mathcal{T}_3,\ \mathcal{T}_6)\\
&\doteq [\![\mathcal{A}((e_1, e_2))]\!]. && (\mathcal{A}_8)
\end{aligned}$$

6. In case $e \doteq \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ we have:

$$\begin{aligned}
\mathcal{F}([\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]) &\doteq \mathcal{F}(\mu\alpha.\langle\, [\![e_1]\!] \mid \tilde{\mu}x.\langle\, [\![e_2]\!] \mid \alpha \,\rangle\,\rangle) && (\mathcal{T}_6)\\
&\doteq \mu\alpha.\langle\, \mathcal{F}([\![e_1]\!]) \mid \tilde{\mu}x.\langle\, \mathcal{F}([\![e_2]\!]) \mid \alpha \,\rangle\,\rangle && (\mathcal{F}_2,\ \mathcal{F}_{13},\ \mathcal{F}_9,\ \mathcal{F}_8)\\
&\doteq \mu\alpha.\langle\, [\![\mathcal{A}(e_1)]\!] \mid \tilde{\mu}x.\langle\, [\![\mathcal{A}(e_2)]\!] \mid \alpha \,\rangle\,\rangle && (\text{IH})\\
&\doteq [\![\mathbf{let}\ x = \mathcal{A}(e_1)\ \mathbf{in}\ \mathcal{A}(e_2)]\!] && (\mathcal{T}_6)\\
&\doteq [\![\mathcal{A}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)]\!]. && (\mathcal{A}_3)
\end{aligned}$$

□

Theorem 6.7.2 (Focusing reflects the ANF-transformation; case $\equiv$)**.** *For all λ-terms e, we have $\mathcal{F}([\![e]\!]) \equiv [\![\mathcal{A}(e)]\!]$.*

Proof. We only have to modify subcase 4(ii) in the proof of theorem 6.7.1. The modified proof is as follows (evaluated redexes are underlined).

4. In case $e \doteq e_1\, e_2$, we have to distinguish two subcases:

   (i) Subcase $e \doteq e_1\, v_2$: identical to the proof of theorem 6.7.1.

(ii) Subcase $e \doteq e_1\, e_2$:

$$
\begin{aligned}
\mathcal{F}(\llbracket(e_1\, e_2)\rrbracket) &\doteq \mathcal{F}(\mu\alpha.\langle\, \llbracket e_1\rrbracket \mid \llbracket e_2\rrbracket \cdot \alpha \,\rangle) && (\mathcal{T}_4)\\
&\doteq \mu\alpha.\underline{\langle \mathcal{F}(\llbracket e_1\rrbracket) \ \mid\ \tilde{\mu}y.\langle\, \mathcal{F}(\llbracket e_2\rrbracket) \mid \tilde{\mu}x.\langle\, y \mid x\cdot\alpha \,\rangle\,\rangle\rangle}\\
& && (\mathcal{F}_2, \mathcal{F}_{13}, \mathcal{F}_9, \mathcal{F}_1, \mathcal{F}_8, \mathcal{F}_{11})\\
&\rhd \mu\alpha.\langle\, \mathcal{F}(\llbracket e_2\rrbracket) \mid \tilde{\mu}x.\langle\, \mathcal{F}(\llbracket e_1\rrbracket) \mid x\cdot\alpha \,\rangle\,\rangle\\
&\overset{\text{IH}}{\equiv} \mu\alpha.\langle\, \llbracket\mathcal{A}(e_2)\rrbracket \mid \tilde{\mu}x.\langle\, \llbracket\mathcal{A}(e_1)\rrbracket \mid x\cdot\alpha \,\rangle\,\rangle\\
&\lhd \mu\alpha.\langle\, \llbracket\mathcal{A}(e_2)\rrbracket \mid \tilde{\mu}x.\underline{\langle\mu\beta.\langle\, \llbracket\mathcal{A}(e_1)\rrbracket \mid x\cdot\beta \,\rangle \ \mid\ \alpha\rangle}\,\rangle\\
&\doteq \llbracket\textbf{let } x = \mathcal{A}(e_2) \textbf{ in } \mathcal{A}(e_1)\, x\rrbracket && (\mathcal{T}_4 + \mathcal{T}_6)\\
&\doteq \llbracket\mathcal{A}(e_1\, e_2)\rrbracket. && (\mathcal{A}_{10})
\end{aligned}
$$

$\square$

In order to prove that for all $e \in \Lambda^{\mathrm{Q}}$, $\llbracket\mathcal{L}(e)\rrbracket^* =_\mu \llbracket e\rrbracket$ (theorem 6.7.6), we introduce a transformation $\mathcal{M}$ that simulates the effect of applying $\mathcal{L}$ on a term from Λ^{Q} on its translation in $\Lambda^{\mathrm{Q}}_{\mu\tilde{\mu}}$.

Definition 6.9.1. The *μ-normalization operation* $\mathcal{M} : \Lambda^{Q}_{\mu\tilde{\mu}} \to \Lambda^{ANF}_{\mu\tilde{\mu}}$ is defined by the following clauses:

Values

$$
\begin{aligned}
\mathcal{M}(x) &:\doteq x && (\mathcal{M}_1)\\
\mathcal{M}(\lambda x.e) &:\doteq \lambda x.\mathcal{M}(e) && (\mathcal{M}_2)\\
\mathcal{M}((w_1, w_2)) &:\doteq (\mathcal{M}(w_1), \mathcal{M}(w_2)) && (\mathcal{M}_3)
\end{aligned}
$$

Terms

$$
\begin{aligned}
\mathcal{M}(e) &:\doteq \mu\alpha.\mathcal{M}_\alpha(e) && (\mathcal{M}_4)\\
\mathcal{M}_s(w) &:\doteq \langle\, \mathcal{M}(w) \mid s \,\rangle && (\mathcal{M}_5)\\
\mathcal{M}_s(\mu\alpha.c) &:\doteq \mathcal{M}(c[s/\alpha]) && (\mathcal{M}_6)
\end{aligned}
$$

Coterms

$$
\begin{aligned}
\mathcal{M}(\alpha) &:\doteq \alpha && (\mathcal{M}_7)\\
\mathcal{M}(\pi_i\, s) &:\doteq \pi_i\, \tilde{\mu}x.\langle\, x \mid s \,\rangle && (\mathcal{M}_8)\\
\mathcal{M}(w\cdot s) &:\doteq w\cdot \tilde{\mu}x.\langle\, x \mid s \,\rangle && (\mathcal{M}_9)\\
\mathcal{M}(\tilde{\mu}x.\langle\, e \mid s \,\rangle) &:\doteq \tilde{\mu}x.\mathcal{M}_s(e) && (\mathcal{M}_{10})
\end{aligned}
$$

Computations

$$
\mathcal{M}(\langle\, e \mid s \,\rangle) :\doteq \mathcal{M}_{\mathcal{M}(s)}(e). \qquad (\mathcal{M}_{11})
$$

Definition 6.9.2. We define the operation $\_;\_$ which takes a continuation k (cf. definition 6.5.6) and a coterm s, and returns a coterm $k; s$:

$$\begin{aligned}
\mathrm{id}; s &:\doteq s && (\mathcal{C}_1)\\
\overline{\lambda v}.\mathbf{let}\ x = \pi_i\,\overline{v}\ \mathbf{in}\ e; s &:\doteq \pi_i\,\tilde{\mu}x.[\![e]\!]^*_s && (\mathcal{C}_2)\\
\overline{\lambda v}.\mathbf{let}\ x = \overline{v}\,v'\ \mathbf{in}\ e; s &:\doteq v' \cdot \tilde{\mu}x.[\![e]\!]^*_s && (\mathcal{C}_3)\\
\overline{\lambda v}.\mathbf{let}\ x = \overline{v}\ \mathbf{in}\ e; s &:\doteq \tilde{\mu}x.[\![v]\!]^*_s. && (\mathcal{C}_4)
\end{aligned}$$

Lemma 6.9.3. *We have* $[\![k \mathbin{@} v]\!]^*_s \doteq \langle\, [\![v]\!]^* \mid k; s \,\rangle$.

Proof. By case analysis on k:

1. Case $k \doteq \mathrm{id}$:

$$\begin{aligned}
[\![\mathrm{id} \mathbin{@} v]\!]^*_s &\doteq [\![v]\!]^*_s && (@_1)\\
&\doteq \langle\, [\![v]\!]^* \mid s \,\rangle && (\mathcal{T}_5^*)\\
&\doteq \langle\, [\![v]\!]^* \mid \mathrm{id}; s \,\rangle. && (\mathcal{C}_1)
\end{aligned}$$

2. Case $k \doteq \overline{\lambda v}.\mathbf{let}\ x = \pi_i\,\overline{v}\ \mathbf{in}\ e$:

$$\begin{aligned}
[\![\overline{\lambda v}.\mathbf{let}\ x = \pi_i\,\overline{v}\ \mathbf{in}\ e \mathbin{@} v]\!]^*_s &\doteq [\![\mathbf{let}\ x = \pi_i\,v\ \mathbf{in}\ e]\!]^*_s && (@_2)\\
&\doteq [\![\pi_i\,v]\!]^*_{\tilde{\mu}x.[\![e]\!]^*_s} && (\mathcal{T}_2^*)\\
&\doteq \langle\, [\![v]\!]^* \mid \pi_i\,\tilde{\mu}x.[\![e]\!]^*_s \,\rangle && (\mathcal{T}_4^*)\\
&\doteq \langle\, [\![v]\!]^* \mid \overline{\lambda v}.\mathbf{let}\ x = \pi_i\,\overline{v}\ \mathbf{in}\ e; s \,\rangle. && (\mathcal{C}_2)
\end{aligned}$$

3. Case $k \doteq \overline{\lambda v}.\mathbf{let}\ x = \overline{v}\,v'\ \mathbf{in}\ e$:

$$\begin{aligned}
[\![\overline{\lambda v}.\mathbf{let}\ x = \overline{v}\,v'\ \mathbf{in}\ e \mathbin{@} v]\!]^*_s &\doteq [\![\mathbf{let}\ x = v\,v'\ \mathbf{in}\ e]\!]^*_s && (@_3)\\
&\doteq [\![v\,v']\!]^*_{\tilde{\mu}x.[\![e]\!]^*_s} && (\mathcal{T}_2^*)\\
&\doteq \langle\, [\![v]\!]^* \mid [\![v']\!]^* \cdot \tilde{\mu}x.[\![e]\!]^*_s \,\rangle && (\mathcal{T}_3^*)\\
&\doteq \langle\, [\![v]\!]^* \mid \overline{\lambda v}.\mathbf{let}\ x = \overline{v}\,v'\ \mathbf{in}\ e; s \,\rangle. && (\mathcal{C}_3)
\end{aligned}$$

4. Case $k \doteq \overline{\lambda v}.\mathbf{let}\ x = \overline{v}\ \mathbf{in}\ e$:

$$\begin{aligned}
[\![\overline{\lambda v}.\mathbf{let}\ x = \overline{v}\ \mathbf{in}\ e \mathbin{@} v]\!]^*_s &\doteq [\![\mathbf{let}\ x = v\ \mathbf{in}\ e]\!]^*_s && (@_4)\\
&\doteq [\![v]\!]^*_{\tilde{\mu}x.[\![e]\!]^*_s} && (\mathcal{T}_2^*)\\
&\doteq \langle\, [\![v]\!]^* \mid \tilde{\mu}x.[\![e]\!]^*_s \,\rangle && (\mathcal{T}_5^*)\\
&\doteq \langle\, [\![v]\!]^* \mid \overline{\lambda v}.\mathbf{let}\ x = \overline{v}\ \mathbf{in}\ e; s \,\rangle. && (\mathcal{C}_4)
\end{aligned}$$

□

The μ-normalization operation $\mathcal{M}$ corresponds precisely to the transformation $\mathcal{L}$:

Lemma 6.9.4. *The following statements hold:*

1. *For all values $v \in \Lambda^Q$: $[\![\mathcal{L}(v)]\!]^* \doteq \mathcal{M}([\![v]\!])$.*
2. *For all expressions $e \in \Lambda^Q$: $[\![\mathcal{L}(e)]\!]^* \doteq \mathcal{M}([\![e]\!])$.*
3. *For all $e \in \Lambda^Q$, continuations k and coterms s:*

$$[\![\mathcal{L}_k(e)]\!]^*_s \doteq \mathcal{M}_{k;s}([\![e]\!]).$$

Proof. We prove these three statements by simultaneous induction. For the first statement, lemma 6.9.4(1), we use induction on v:

1. Case $v \doteq x$:

$$\begin{aligned}
[\![\mathcal{L}(x)]\!]^* &\doteq [\![x]\!]^* && (\mathcal{L}_1)\\
&\doteq x && (\mathcal{T}_6^*)\\
&\doteq \mathcal{M}(x) && (\mathcal{M}_1)\\
&\doteq \mathcal{M}([\![x]\!]). && (\mathcal{T}_1)
\end{aligned}$$

2. Case $v \doteq (v_1, v_2)$:

$$\begin{aligned}
[\![\mathcal{L}((v_1, v_2))]\!]^* &\doteq [\![(\mathcal{L}(v_1), \mathcal{L}(v_2))]\!]^* && (\mathcal{L}_3)\\
&\doteq ([\![\mathcal{L}(v_1)]\!]^*, [\![\mathcal{L}(v_2)]\!]^*) && (\mathcal{T}_7^*)\\
&\doteq (\mathcal{M}([\![v_1]\!]), \mathcal{M}([\![v_2]\!])) && (\text{IH for lemma 6.9.4(1)})\\
&\doteq \mathcal{M}(([\![v_1]\!], [\![v_2]\!])) && (\mathcal{M}_3)\\
&\doteq \mathcal{M}([\![(v_1, v_2)]\!]). && (\mathcal{T}_3)
\end{aligned}$$

3. Case $v \doteq \lambda x.e$:

$$\begin{aligned}
[\![\mathcal{L}(\lambda x.e)]\!]^* &\doteq [\![\lambda x.\mathcal{L}(e)]\!]^* && (\mathcal{L}_2)\\
&\doteq \lambda x.[\![\mathcal{L}(e)]\!]^* && (\mathcal{T}_8^*)\\
&\doteq \lambda x.\mathcal{M}([\![e]\!]) && (\text{IH for lemma 6.9.4(2)})\\
&\doteq \mathcal{M}(\lambda x.[\![e]\!]) && (\mathcal{M}_2)\\
&\doteq \mathcal{M}([\![\lambda x.e]\!]). && (\mathcal{T}_2)
\end{aligned}$$

For lemma 6.9.4(2), we show the following:

$$\begin{aligned}
[\![\mathcal{L}(e)]\!]^* &\doteq \mu\alpha.[\![\mathcal{L}(e)]\!]^*_\alpha && (\mathcal{T}_1^*)\\
&\doteq \mu\alpha.[\![\mathcal{L}_{\mathrm{id}}(e)]\!]^*_\alpha && (\mathcal{L}_4)\\
&\doteq \mu\alpha.\mathcal{M}_{\mathrm{id};\alpha}([\![e]\!]) && (\text{IH for lemma 6.9.4(3)})\\
&\doteq \mu\alpha.\mathcal{M}_\alpha([\![e]\!]) && (\mathcal{C}_1)\\
&\doteq \mathcal{M}([\![e]\!]). && (\mathcal{M}_4)
\end{aligned}$$

For lemma 6.9.4(3), we perform induction on e:

1. Case $e \bumpeq v$:

$$\begin{aligned}
\llbracket \mathcal{L}_k(v) \rrbracket^*_s &\bumpeq \llbracket k \mathbin{@} \mathcal{L}(v) \rrbracket^*_s && (\mathcal{L}_7)\\
&\bumpeq \langle\, \llbracket \mathcal{L}(v) \rrbracket^* \mid k; s \,\rangle && \text{(lemma 6.9.3)}\\
&\bumpeq \langle\, \mathcal{M}(\llbracket v \rrbracket) \mid k; s \,\rangle && \text{(IH for 1)}\\
&\bumpeq \mathcal{M}_{k;s}(\llbracket v \rrbracket). && (\mathcal{M}_5)
\end{aligned}$$

2. Case $e \bumpeq \textbf{let } x = e_1 \textbf{ in } e_2$:

$$\begin{aligned}
\llbracket \mathcal{L}_k(\textbf{let } x = e_1 \textbf{ in } e_2) \rrbracket^*_s &\bumpeq \llbracket \mathcal{L}_{\overline{\lambda v}.\textbf{let } x=\overline{v} \textbf{ in } \mathcal{L}_k(e_2)}(e_1) \rrbracket^*_s && (\mathcal{L}_8)\\
&\bumpeq \mathcal{M}_{\overline{\lambda v}.\textbf{let } x=\overline{v} \textbf{ in } \mathcal{L}_k(e_2);s}(\llbracket e_1 \rrbracket) && \text{(IH)}\\
&\bumpeq \mathcal{M}_{\tilde{\mu}x.\llbracket \mathcal{L}_k(e_2) \rrbracket^*_s}(\llbracket e_1 \rrbracket) && (\mathcal{C}_4)\\
&\bumpeq \mathcal{M}_{\tilde{\mu}x.\mathcal{M}_{k;s}(\llbracket e_2 \rrbracket)}(\llbracket e_1 \rrbracket) && \text{(IH)}\\
&\bumpeq \mathcal{M}_{\mathcal{M}(\tilde{\mu}x.\langle\, \llbracket e_2 \rrbracket \mid k;s \,\rangle)}(\llbracket e_1 \rrbracket) && (\mathcal{M}_{10})\\
&\bumpeq \mathcal{M}(\langle\, \llbracket e_1 \rrbracket \mid \tilde{\mu}x.\langle\, \llbracket e_2 \rrbracket \mid k; s \,\rangle \,\rangle) && (\mathcal{M}_{11})\\
&\bumpeq \mathcal{M}_{k;s}(\mu\alpha.\langle\, \llbracket e_1 \rrbracket \mid \tilde{\mu}x.\langle\, \llbracket e_2 \rrbracket \mid \alpha \,\rangle \,\rangle) && (\mathcal{M}_6)\\
&\bumpeq \mathcal{M}_{k;s}(\llbracket \textbf{let } x = e_1 \textbf{ in } e_2 \rrbracket). && (\mathcal{T}_6)
\end{aligned}$$

3. Case $e \bumpeq e\, v$:

$$\begin{aligned}
\llbracket \mathcal{L}_k(e\, v) \rrbracket^*_s &\bumpeq \llbracket \mathcal{L}_{\overline{\lambda v'}.\textbf{let } x=v'\, v \textbf{ in } k \mathbin{@} x}(e) \rrbracket^*_s && (\mathcal{L}_5)\\
&\bumpeq \mathcal{M}_{\overline{\lambda v'}.\textbf{let } x=v'\, v \textbf{ in } k \mathbin{@} x;s}(\llbracket e \rrbracket) && \text{(IH)}\\
&\bumpeq \mathcal{M}_{\llbracket v \rrbracket \cdot \tilde{\mu}x.\llbracket k \mathbin{@} x \rrbracket^*_s}(\llbracket e \rrbracket) && (\mathcal{C}_3)\\
&\bumpeq \mathcal{M}_{\llbracket v \rrbracket \cdot \tilde{\mu}x.\langle\, x \mid s \,\rangle}(\llbracket e \rrbracket) && \text{(lemma 6.9.3)}\\
&\bumpeq \mathcal{M}_{\mathcal{M}(\llbracket v \rrbracket \cdot k;s)}(\llbracket e \rrbracket) && (\mathcal{M}_9)\\
&\bumpeq \mathcal{M}(\langle\, \llbracket e \rrbracket \mid \llbracket v \rrbracket \cdot k; s \,\rangle) && (\mathcal{M}_{11})\\
&\bumpeq \mathcal{M}_{k;s}(\mu\alpha.\langle\, \llbracket e \rrbracket \mid \llbracket v \rrbracket \cdot \alpha \,\rangle) && (\mathcal{M}_6)\\
&\bumpeq \mathcal{M}_{k;s}(\llbracket e\, v \rrbracket). && (\mathcal{T}_4)
\end{aligned}$$

4. Case $e \bumpeq \pi_i\, e$:

$$\begin{aligned}
\llbracket \mathcal{L}_k(\pi_i\, e) \rrbracket^*_s &\bumpeq \llbracket \mathcal{L}_{\overline{\lambda v}.\textbf{let } x=\pi_i\, \overline{v} \textbf{ in } k \mathbin{@} x}(e) \rrbracket^*_s && (\mathcal{L}_6)\\
&\bumpeq \mathcal{M}_{\overline{\lambda v}.\textbf{let } x=\pi_i\, \overline{v} \textbf{ in } k \mathbin{@} x;s}(\llbracket e \rrbracket) && \text{(IH)}\\
&\bumpeq \mathcal{M}_{\pi_i\, \tilde{\mu}x.\llbracket k \mathbin{@} x \rrbracket^*_s}(\llbracket e \rrbracket) && (\mathcal{C}_2)\\
&\bumpeq \mathcal{M}_{\pi_i\, \tilde{\mu}x.\langle\, x \mid k;s \,\rangle}(\llbracket e \rrbracket) && \text{(lemma 6.9.3)}\\
&\bumpeq \mathcal{M}_{\mathcal{M}(\pi_i\, (k;s))}(\llbracket e \rrbracket) && (\mathcal{M}_8)\\
&\bumpeq \mathcal{M}(\langle\, \llbracket e \rrbracket \mid \pi_i\, (k; s) \,\rangle) && (\mathcal{M}_{11})\\
&\bumpeq \mathcal{M}_{k;s}(\mu\alpha.\langle\, \llbracket e \rrbracket \mid \pi_i\, \alpha \,\rangle) && (\mathcal{M}_6)\\
&\bumpeq \mathcal{M}_{k;s}(\llbracket \pi_i\, e \rrbracket). && (\mathcal{T}_5)
\end{aligned}$$

□

The effect of the application of $\mathcal{M}$ can be achieved by applying μ-reductions in commands and η-expansions in coterms:

Lemma 6.9.5. *For all terms, coterms and commands $t \in \Lambda^{Q}_{\mu\tilde{\mu}}$ we have $t \equiv_{\mu} \mathcal{M}(t)$.*

Proof. By inspection of the relevant clauses in definition 6.9.1. The combination of the clauses ($\mathcal{M}_6$) and ($\mathcal{M}_{11}$) corresponds to the reduction of μ-redexes. Coterms are η-expanded in the clauses ($\mathcal{M}_8$) and ($\mathcal{M}_9$). □

Theorem 6.7.6. *For all terms $e \in \Lambda^{Q}$, $[\![\mathcal{L}(e)]\!]^{*} \equiv_{\mu} [\![e]\!]$.*

Proof. By combining Lemmas 6.9.4 and 6.9.5. □

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. Rome, Italy: Association for Computing Machinery, pp. 27–38. ISBN: 9781450318327. URL: `https://doi.org/10.1145/2480359.2429075` (cit. on pp. 5, 46, 47, 50, 69, 79, 129).

Andreoli, Jean-Marc (1992). “Logic Programming with Focusing Proofs in Linear Logic”. In: *Journal of Logic and Computation* 2 (3), pp. 297–347. URL: `https://doi.org/10.1093/logcom/2.3.297` (cit. on pp. 109, 128).

Appel, Andrew W. (1992). *Compiling with Continuations.* Cambridge University Press (cit. on p. 144).

Ariola, Zena M. and Hugo Herbelin (2003). “Minimal classical logic and control operators”. In: *Automata, Languages and Programming.* Ed. by Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger. Springer, pp. 871–885 (cit. on p. 139).

Ariola, Zena M., John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler (1995). “A Call-by-Need Lambda Calculus”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’95. San Francisco, California, USA: Association for Computing Machinery, pp. 233–246. DOI: `10.1145/199448.199507` (cit. on pp. 39, 44, 152).

Atkey, Robert and Conor McBride (2013). “Productive Coprogramming with Guarded Recursion”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 197–208 (cit. on p. 59).

Barendregt, Hendrik Pieter (1981). *The Lambda Calculus: Its Syntax and Semantics.* New York, NY, USA: Elsevier (cit. on pp. 12, 32).

Bhanuka, Ishan, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser (Oct. 2023). “Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2. DOI: `10.1145/3622812`. URL: `https://doi.org/10.1145/3622812` (cit. on p. 19).

Binder, David, Julian Jabs, Ingo Skupin, and Klaus Ostermann (2019). “Decomposition Diversity with Symmetric Data and Codata”. In: *Proc. ACM Program. Lang.* 4.POPL. DOI: `10.1145/3371098`. URL: `https://doi.org/10.1145/3371098` (cit. on pp. 45, 99).

— (2022). *Data-Codata Symmetry and its Interaction with Evaluation Order.* DOI: `10.48550/ARXIV.2211.13004`. URL: `https://arxiv.org/abs/2211.13004` (cit. on p. 75).

Binder, David and Thomas Piecha (2017). “Popper’s Notion of Duality and His Theory of Negations”. In: *History and Philosophy of Logic* 38.2, pp. 154–189. DOI: `10.1080/01445340.2016.1278517`. eprint: `https://doi.org/10.1080/01445340.2016.1278517`. URL: `https://doi.org/10.1080/01445340.2016.1278517` (cit. on p. 6).

— (2021). “Popper on Quantification and Identity”. In: *Karl Popper’s Science and Philosophy.* Ed. by Zuzana Parusniková and David Merritt. Cham: Springer International Publishing, pp. 149–169. ISBN: 978-3-030-67036-8. DOI: `10.1007/978-3-030-67036-8_8`. URL: `https://doi.org/10.1007/978-3-030-67036-8_8` (cit. on p. 6).

— (2022). “Administrative Normal Forms and Focusing for Lambda Calculi”. In: *Logically Speaking. A Festschrift for Marie Duží.* Ed. by Pavel Materna and Bjørn Jespersen. Vol. 49. Tributes. College Publications (cit. on p. 133).

Binder, David, Thomas Piecha, and Peter Schroeder-Heister (2022). *The Logical Writings of Karl Popper.* Vol. 58. Trends in Logic. Springer Cham, p. 552 (cit. on p. 6).

Binder, David, Ingo Skupin, David Läwen, and Klaus Ostermann (2022). “Structural Refinement Types”. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development.* TyDe 2022. Ljubljana, Slovenia: Association for Computing Machinery, pp. 15–27. ISBN: 9781450394390. DOI: `10.1145/3546196.3550163`. URL: `https://doi.org/10.1145/3546196.3550163` (cit. on p. 19).

Binder, David, Ingo Skupin, Tim Süberkrüb, and Klaus Ostermann (Apr. 2024a). “Deriving Dependently-Typed OOP from First Principles”. In: *Proc. ACM Program. Lang.* 8.OOPSLA1. DOI: `10.1145/3649846`. URL: `https://doi.org/10.1145/3649846` (cit. on pp. 17, 45, 71).

— (2024b). “Deriving Dependently-Typed OOP from First Principles – Extended Version with Additional Appendices”. In: DOI: `10.48550/arXiv.2403.06707`. URL: `https://doi.org/10.48550/arXiv.2403.06707` (cit. on p. 8).

Binder, David, Marco Tzschentke, Marius Müller, and Klaus Ostermann (Aug. 2024). “Grokking the Sequent Calculus (Functional Pearl)”. In: *Proc. ACM Program. Lang.* 8.ICFP. DOI: `10.1145/3674639`. URL: `https://doi.org/10.1145/3674639` (cit. on pp. 17, 18, 39, 41, 42).

Boquist, Urban and Thomas Johnsson (1996). “The GRIN project: A highly optimising back end for lazy functional languages”. In: *Symposium on Implementation and Application of Functional Languages.* Springer, pp. 58–84 (cit. on p. 70).

Caires, Luis and Frank Pfenning (2010). “Session Types as Intuitionistic Linear Propositions”. In: *Proceedings of the 21st International Conference on Concurrency Theory.* CONCUR’10. Paris, France: Springer, pp. 222–236. ISBN: 3642153747. URL: `https://doi.org/10.1007/978-3-642-15375-4_16` (cit. on p. 131).

Carette, Jacques, Oleg Kiselyov, and Chung-Chieh Shan (2007). “Finally tagless, partially evaluated”. In: *Proceedings of the Asian Symposium on Programming Languages and Systems.* Springer, pp. 222–238. DOI: `10.1007/978-3-540-76637-7_15` (cit. on p. 47).

Carraro, Alberto, Thomas Ehrhard, and Antonino Salibra (2012). “The stack calculus”. In: *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012, Rio de Janeiro, Brazil, September 29-30, 2012.* Vol. 113.

EPTCS, pp. 93–108. URL: https://doi.org/10.48550/arXiv.1303.7331 (cit. on p. 130).

Church, Alonzo (1936). “An Unsolvable Problem of Elementary Number Theory”. In: *Journal of Symbolic Logic* 1.2, pp. 73–74. DOI: 10.2307/2268571 (cit. on pp. 1, 13).

— (1940). “A Formulation of the Simple Theory of Types”. In: *The Journal of Symbolic Logic* 5.2, pp. 56–68. ISSN: 00224812. URL: http://www.jstor.org/stable/2266170 (cit. on p. 13).

Cook, William R. (1990). “Object-Oriented Programming versus Abstract Data Types”. In: *Proceedings of the REX Workshop / School on the Foundations of Object-Oriented Languages*. Springer, pp. 151–178. DOI: 10.1007/BFb0019443. URL: https://doi.org/10.1007/BFb0019443 (cit. on pp. 46, 70, 80).

— (2009). “On Understanding Data Abstraction, Revisited”. In: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications: Onward! Essays*. Orlando: Association for Computing Machinery, pp. 557–572. DOI: 10.1145/1640089.1640133. URL: https://doi.org/10.1145/1640089.1640133 (cit. on pp. 47, 69, 80).

Crolard, Tristan (2004). “A Formulae-as-Types Interpretation of Subtractive Logic”. In: *Journal of Logic and Computation* 14 (4), pp. 529–570. URL: https://doi.org/10.1093/logcom/14.4.529 (cit. on pp. 115, 128).

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: Association for Computing Machinery, pp. 233–243. URL: https://doi.org/10.1145/357766.351262 (cit. on pp. 16, 77, 82, 99, 108, 115, 127, 128, 130, 133, 137, 138, 147).

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. Austin, Texas, USA: Association for Computing Machinery, pp. 1–12. ISBN: 978-1-4503-0485-6 (cit. on pp. 54, 56, 68).

Danvy, Olivier and Kevin Millikin (2009). “Refunctionalization at Work”. In: *Science of Computer Programming* 74.8, pp. 534–549. DOI: https://doi.org/10.1016/j.scico.2007.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0167642309000227 (cit. on pp. 46, 48, 49, 68, 99).

Danvy, Olivier and Lasse R. Nielsen (2001). “Defunctionalization at Work”. In: *Proceedings of the Conference on Principles and Practice of Declarative Programming*. Florence, pp. 162–174. ISBN: 1-58113-388-X. DOI: 10.1145/773184.773202. URL: https://doi.org/10.1145/773184.773202 (cit. on pp. 46, 48, 68, 99).

Dolan, Stephen (2017). “Algebraic Subtyping”. PhD thesis. University of Cambridge (cit. on p. 19).

Dolan, Stephen and Alan Mycroft (2017). “Polymorphism, subtyping, and type inference in MLsub”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: Association for Computing Machinery, pp. 60–72. ISBN: 9781450346603. DOI: 10.1145/3009837.3009882. URL: https://doi.org/10.1145/3009837.3009882 (cit. on p. 19).

Downen, Paul and Zena M. Ariola (2014). “The Duality of Construction”. In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. ESOP ’14. Berlin, Heidelberg: Springer, pp. 249–269. URL: `https://doi.org/10.1007/978-3-642-54833-8_14` (cit. on pp. 77, 127).

— (2018a). “A tutorial on computational classical logic and the sequent calculus”. In: *Journal of Functional Programming* 28. URL: `https://doi.org/10.1017/S0956796818000023` (cit. on pp. 109, 133, 148).

— (2018b). “Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing”. In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Ed. by Dan Ghica and Achim Jung. Vol. 119. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:23. ISBN: 978-3-95977-088-0. DOI: `10.4230/LIPIcs.CSL.2018.21`. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/9688` (cit. on pp. 84, 152).

— (Aug. 2020). “Compiling With Classical Connectives”. In: *Logical Methods in Computer Science* Volume 16, Issue 3. DOI: `10.23638/LMCS-16(3:13)2020`. URL: `https://lmcs.episciences.org/6740` (cit. on pp. 77, 78, 99).

— (2021). “Duality in Action”. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD*. Ed. by Naoki Kobayashi. Vol. 195. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 1–32. URL: `https://doi.org/10.4230/LIPIcs.FSCD.2021.1` (cit. on p. 129).

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*, pp. 74–88 (cit. on p. 17).

Downen, Paul, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones (2019). “Codata in Action”. In: *European Symposium on Programming*. ESOP ’19. Springer, pp. 119–146. URL: `https://doi.org/10.1007/978-3-030-17184-1_5` (cit. on pp. 48, 69, 70, 79, 128).

Dummett, Michael (1991). *The Logical Basis of Metaphysics*. Harvard University Press (cit. on p. 6).

Felleisen, Matthias, Robert Bruce Findler, and Matthew Flatt (2009). *Semantics Engineering with PLT Redex*. MIT Press (cit. on p. 143).

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi (2001). *How to Design Programs. An Introduction to Computing and Programming*. Cambridge, Massachusetts London, England: MIT Press (cit. on p. 107).

Felleisen, Matthias, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba (1987). “A syntactic theory of sequential control”. In: *Theoretical Computer Science* 52.3, pp. 205–237. ISSN: 0304-3975. DOI: `10.1016/0304-3975(87)90109-5`. URL: `https://doi.org/10.1016/0304-3975(87)90109-5` (cit. on p. 16).

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. DOI: `https://doi.org/10.1016/0304-3975(92)90014-7`. URL:

`https://www.sciencedirect.com/science/article/pii/0304397592900147` (cit. on pp. 134, 142).

Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen (1993). "The Essence of Compiling with Continuations". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation.* PLDI '93. Albuquerque, New Mexico, USA: Association for Computing Machinery, pp. 237–247 (cit. on pp. 136, 144).

Frege, Gottlob (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle: Verlag von Louis Nebert (cit. on p. 11).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley Publishing Co. (cit. on p. 47).

Gentzen, Gerhard (1935a). "Untersuchungen über das logische Schließen. I." In: *Mathematische Zeitschrift* 35, pp. 176–210. English translation in Gentzen, 1969 (cit. on pp. 6, 13, 128).

— (1935b). "Untersuchungen über das logische Schließen. II." In: *Mathematische Zeitschrift* 39, pp. 405–431. English translation in Gentzen, 1969 (cit. on pp. 6, 13, 128).

— (1969). *The collected papers of Gerhard Gentzen.* Ed. by Manfred Egon Szabo. Amsterdam: North-Holland Publishing Co. (cit. on p. 165).

Gibbons, Jeremy (2021). "How to design co-programs". In: *Journal of Functional Programming* 31. URL: `https://doi.org/10.1017/S0956796821000113` (cit. on p. 107).

Giménez, Eduardo (1996). "An application of co-inductive types in Coq: Verification of the alternating bit protocol". In: *Types for Proofs and Programs.* Ed. by Stefano Berardi and Mario Coppo. Berlin Heidelberg: Springer (cit. on p. 69).

Girard, Jean-Yves (1987). "Linear Logic". In: *Theoretical Computer Science* 50.1, pp. 1–101. ISSN: 0304-3975. URL: `https://doi.org/10.1016/0304-3975(87)90045-4` (cit. on pp. 128, 129).

— (2001). "Locus Solum: From the rules of logic to the logic of rules". In: *Mathematical Structures in Computer Science* 11.3, pp. 301–506. DOI: `10.1017/S096012950100336X` (cit. on p. 84).

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.* POPL '90. San Francisco, California, USA: Association for Computing Machinery, pp. 47–58. DOI: `10.1145/96709.96714` (cit. on pp. 16, 139).

Groote, Philippe de (1994). "On the Relation between the $\lambda\mu$-Calculus and the Syntactic Theory of Sequential Control". In: *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning.* LPAR '94. Berlin, Heidelberg: Springer, pp. 31–43. URL: `https://doi.org/10.1007/3-540-58216-9_27` (cit. on p. 130).

Grust, Torsten, Nils Schweinsberg, and Alexander Ulrich (2013). "Functions are data too: defunctionalization for PL/SQL". In: *Proceedings of the VLDB Endowment* 6.12, pp. 1214–1217 (cit. on p. 70).

Hagino, Tatsuya (1989). “Codatatypes in ML”. In: *Journal of Symbolic Computation* 8.6, pp. 629–650. URL: `https://doi.org/10.1016/S0747-7171(89)80065-3` (cit. on pp. 69, 79, 128).

Herbelin, Hugo and Silvia Ghilezan (2008). “An Approach to Call-by-Name Delimited Continuations”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’08. San Francisco, California, USA: Association for Computing Machinery, pp. 383–394. URL: `https://doi.org/10.1145/1328438.1328484` (cit. on p. 130).

Hilbert, David (1931). “Die Grundlegung der Elementaren Zahlenlehre”. In: *Mathematische Annalen* 104.1, pp. 485–494 (cit. on p. 99).

Hindley, James Roger and Jonathan Paul Seldin (2008). *Lambda-Calculus and Combinators an Introduction.* Cambridge University Press (cit. on pp. 12, 37).

Hofmann, Martin and Thomas Streicher (1997). “Continuation models are universal for $\lambda\mu$-calculus”. In: *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 – July 2, 1997.* IEEE Computer Society, pp. 387–395. URL: `https://doi.org/10.1109/LICS.1997.614964` (cit. on p. 128).

Howard, William Alvin (1980). “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Ed. by Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan. Academic Press (cit. on p. 13).

Huang, Yulong and Jeremy Yallop (June 2023). “Defunctionalization with Dependent Types”. In: *Proc. ACM Program. Lang.* 7.PLDI. DOI: `10.1145/3591241`. URL: `https://doi.org/10.1145/3591241` (cit. on p. 71).

Hughes, John (1989). “Why functional programming matters”. In: *The computer journal* 32.2, pp. 98–107 (cit. on p. 9).

Igarashi, Atsushi, Benjamin C Pierce, and Philip Wadler (2001). “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3, pp. 396–450 (cit. on p. 23).

Jacobs, Bart (1995). “Objects and Classes, Coalgebraically”. In: *Object Orientation with Parallelism and Persistence.* Ed. by Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek. USA: Springer, pp. 83–103. ISBN: 978-1-4613-1437-0. DOI: `10.1007/978-1-4613-1437-0_5`. URL: `https://doi.org/10.1007/978-1-4613-1437-0_5` (cit. on p. 69).

Jaśkowski, Stanisław (1934). “On the Rules of Suppositions in Formal Logic”. In: *Studia Logica* 1, pp. 5–32 (cit. on p. 13).

Johann, Patricia and Janis Voigtländer (2004). “Free Theorems in the Presence of Seq”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’04. Venice, Italy: Association for Computing Machinery, pp. 99–110. ISBN: 158113729X. DOI: `10.1145/964001.964010`. URL: `https://doi.org/10.1145/964001.964010` (cit. on p. 77).

Johnsson, Thomas (1985). “Lambda lifting: Transforming programs to recursive equations”. In: *Functional Programming Languages and Computer Architecture.* Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer, pp. 190–203. ISBN: 978-3-540-39677-2 (cit. on p. 49).

Krishnamurthi, Shriram, Matthias Felleisen, and Daniel P. Friedman (1998). "Synthesizing Object-Oriented and Functional Design to Promote Re-Use". In: *Proceedings of the 12th European Conference on Object-Oriented Programming.* ECCOP '98. Berlin, Heidelberg: Springer, pp. 91–113. ISBN: 3-540-64737-6. URL: `http://dl.acm.org/citation.cfm?id=646155.679709` (cit. on p. 46).

Laforgue, Paul and Yann Régis-Gianas (2017). "Copattern Matching and First-class Observations in OCaml, with a Macro". In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming.* PPDP '17. New York, NY, USA: Association for Computing Machinery (cit. on pp. 48, 69, 70).

Lämmel, Ralf and Ondrej Rypacek (July 2008). "The Expression Lemma". In: *Proceedings of the Conference on Mathematics of Program Construction.* Springer (cit. on p. 69).

Landin, Peter John (Feb. 1965). "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *Commun. ACM* 8.2, pp. 89–101. ISSN: 0001-0782. DOI: `10.1145/363744.363749`. URL: `https://doi.org/10.1145/363744.363749` (cit. on p. 16).

Launchbury, John (1993). "A Natural Semantics for Lazy Evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, pp. 144–154. DOI: `10.1145/158511.158618` (cit. on pp. 44, 152).

Lovas, William and Karl Crary (2006). "Structural Normalization for Classical Natural Deduction". In: *Draft* (cit. on p. 127).

MacQueen, David, Robert Harper, and John Reppy (2020). "The history of Standard ML". In: *Proceedings of the ACM on Programming Languages* 4.HOPL, pp. 1–100 (cit. on p. 1).

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 (cit. on p. 6).

Munch-Maccagnoni, Guillaume (Sept. 2009). "Focalisation and Classical Realisability". In: *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL.* Ed. by Erich Grädel and Reinhard Kahle. CSL '09. Coimbra, Portugal: Springer, pp. 409–423. URL: `https://doi.org/10.1007/978-3-642-04027-6_30` (cit. on pp. 129, 130).

Nakazawa, Koji and Tomoharu Nagai (2014). "Reduction System for Extensional Lambda-mu Calculus". In: *Rewriting and Typed Lambda Calculi.* Ed. by Gilles Dowek. Cham: Springer International Publishing, pp. 349–363. URL: `https://doi.org/10.1007/978-3-319-08918-8_24` (cit. on p. 130).

Negri, Sara (2002). "Varieties of Linear Calculi". In: *J. Philos. Log.* 31.6, pp. 569–590. URL: `https://doi.org/10.1023/A:1021264102972` (cit. on p. 130).

Negri, Sara and Jan Von Plato (2001). *Structural Proof Theory.* Cambridge University Press. URL: `https://doi.org/10.1017/CBO9780511527340` (cit. on p. 130).

Oliveira, Bruno C. d. S. and William R. Cook (2012). "Extensibility for the Masses: Practical Extensibility with Object Algebras". In: *Proceedings of the 26th Euro-*

pean Conference on Object-Oriented Programming. ECOOP'12. Berlin, Heidelberg: Springer, pp. 2–27 (cit. on p. 69).

Ostermann, Klaus, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen (2022a). "Introduction and Elimination, Left and Right". In: *Proc. ACM Program. Lang.* 6.ICFP. DOI: `10.1145/3547637`. URL: `https://doi.org/10.1145/3547637` (cit. on pp. 18, 101).

— (2022b). *Introduction and Elimination, Left and Right – Coq Formalization.* DOI: `10.5281/zenodo.6685674`. URL: `https://doi.org/10.5281/zenodo.6685674` (cit. on p. 101).

Ostermann, Klaus and Julian Jabs (2018). "Dualizing Generalized Algebraic Data Types by Matrix Transposition". In: *European Symposium on Programming.* Springer, pp. 60–85. URL: `https://doi.org/10.1007/978-3-319-89884-1_3` (cit. on pp. 71, 76, 78, 81, 90, 99, 131).

Parigot, Michel (1992a). "$\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction". In: *Logic Programming and Automated Reasoning.* Ed. by Andrei Voronkov. Berlin, Heidelberg: Springer, pp. 190–201 (cit. on pp. 16, 127, 130).

— (1992b). "Free deduction: An analysis of "Computations" in classical logic". In: *Logic Programming.* Ed. by A. Voronkov. Berlin, Heidelberg: Springer, pp. 361–380. URL: `https://doi.org/10.1007/3-540-55460-2_27` (cit. on pp. 130, 131).

Parreaux, Lionel (2020). "The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 4.ICFP, pp. 1–28 (cit. on p. 19).

Pierce, Benjamin C. (2002). *Types and Programming Languages.* MIT Press (cit. on p. 137).

Plotkin, Gordon D. (1975). "Call-By-Name, Call-By-Value and the λ-Calculus". In: *Theoretical Computer Science* 1 (2), pp. 125–159. URL: `https://doi.org/10.1016/0304-3975(75)90017-1` (cit. on p. 128).

Pottier, François and Nadji Gauthier (Mar. 2006). "Polymorphic Typed Defunctionalization and Concretization". In: *Higher-Order and Symbolic Computation* 19.1, pp. 125–162. ISSN: 1388-3690. DOI: `10.1007/s10990-006-8611-7`. URL: `https://doi.org/10.1007/s10990-006-8611-7` (cit. on pp. 76, 99).

Prawitz, Dag (1965). *Natural Deduction: A Proof-Theoretical Study.* Stockholm, Göteborg, Uppsala: Almqvist & Wiksell (cit. on p. 128).

— (1985). "Remarks on Some Approaches to the Concept of Logical Consequence". In: *Synthese* 62.2, pp. 153–171. DOI: `10.1007/BF00486044`. URL: `https://doi.org/10.1007/BF00486044` (cit. on p. 6).

Rauszer, Cecylia (1974). "A formalization of the propositional calculus of H-B logic". In: *Studia Logica* 33, pp. 23–34 (cit. on p. 128).

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 2015. Vancouver, BC, Canada: Association for Computing Machinery, pp. 269–279. ISBN: 9781450336697. DOI: `10.`

1145/2784731.2784763. URL: https://doi.org/10.1145/2784731.2784763 (cit. on pp. 48, 50, 69, 70, 81, 99, 131).

Reynolds, John C. (1975). "User-defined types and procedural data structures as complementary approaches to data abstraction". In: *New Directions in Algorithmic Languages 1975*. Ed. by Stephen Schuman. IFIP Working Group 2.1 on Algol. Rocquencourt, France: INRIA, pp. 157–168 (cit. on p. 46).

Reynolds, John Charles (1972). "Definitional Interpreters for Higher-Order Programming Languages". In: *ACMConf*. Boston: Association for Computing Machinery, pp. 717–740. URL: https://doi.org/10.1145/800194.805852 (cit. on pp. 46, 48, 68, 99, 102, 143).

Sabry, Amr and Matthias Felleisen (1992). "Reasoning about Programs in Continuation-Passing Style." In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. San Francisco, California, USA: Association for Computing Machinery, pp. 288–298 (cit. on pp. 39, 136, 144).

Saurin, Alexis (2005). "Separation with Streams in the $\lambda\mu$-calculus". In: *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*. LICS '05. USA: IEEE Computer Society, pp. 356–365. URL: https://doi.org/10.1109/LICS.2005.48 (cit. on p. 130).

Schroeder-Heister, Peter (2018). "Proof-Theoretic Semantics". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2018. Metaphysics Research Lab, Stanford University (cit. on pp. 6, 131).

Sestoft, Peter (2001). "Demonstrating Lambda Calculus Reduction". In: *Electronic Notes in Theoretical Computer Science* 45. MFPS 2001, Seventeenth Conference on the Mathematical Foundations of Programming Semantics, pp. 424–432. DOI: https://doi.org/10.1016/S1571-0661(04)80973-3 (cit. on pp. 38, 44).

Shivers, Olin and David Fisher (2004). "Multi-Return Function Call". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ICFP '04. New York, NY, USA: Association for Computing Machinery, pp. 79–89. URL: https://doi.org/10.1145/1016848.1016864 (cit. on p. 103).

Sørensen, Morten Heine and Paweł Urzyczyn (2006). *Lectures on the Curry-Howard Isomorphism*. Vol. 149. Studies in Logic and the Foundations of Mathematics. Elsevier (cit. on pp. 13, 144).

Spiwack, Arnaud (2014). "A Dissection of L". Unpublished draft (cit. on pp. 121, 129).

Stoughton, Allen (1988). "Substitution revisited". In: *Theoretical Computer Science* 59.3, pp. 317–325. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(88)90149-1. URL: https://www.sciencedirect.com/science/article/pii/0304397588901491 (cit. on p. 25).

Takahashi, Masako (1995). "Parallel Reductions in Lambda-Calculus". In: *Information and Computation* 118.1, pp. 120–127. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1995.1057. URL: https://www.sciencedirect.com/science/article/pii/S0890540185710577 (cit. on p. 33).

Tranchini, Luca (2012). "Natural Deduction for Dual-intuitionistic Logic". In: *Studia Logica* 100.3, pp. 631–648. URL: https://doi.org/10.1007/s11225-012-9417-8 (cit. on pp. 105, 114, 115, 128).

Troelstra, Anne Sjerp and Helmut Schwichtenberg (2000). *Basic Proof Theory, Second Edition*. Cambridge University Press (cit. on p. 10).

Turner, David A (1979). “Another Algorithm for Bracket Abstraction”. In: *The Journal of Symbolic Logic* 44.2, pp. 267–270. ISSN: 00224812. URL: `http://www.jstor.org/stable/2273733` (cit. on p. 12).

Wadler, Philip (1990). “Linear Types Can Change the World!” In: *Programming Concepts and Methods*. North-Holland (cit. on p. 128).

— (Nov. 1998). “The Expression Problem”. Note to Java Genericity mailing list. URL: `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt` (cit. on pp. 6, 46).

— (2003). “Call-by-value is dual to call-by-name”. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’03. Uppsala, Sweden: Association for Computing Machinery, pp. 189–201. ISBN: 1-58113-756-7. URL: `https://doi.org/10.1145/944705.944723` (cit. on pp. 70, 77, 99, 103, 109, 127).

— (2005). “Call-by-Value Is Dual to Call-by-Name - Reloaded”. In: *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*. Ed. by Jürgen Giesl. Vol. 3467. Lecture Notes in Computer Science. Springer, pp. 185–203. URL: `https://doi.org/10.1007/978-3-540-32033-3_15` (cit. on p. 127).

— (2014). “Propositions as sessions”. In: *J. Funct. Program.* 24.2-3, pp. 384–418. URL: `https://doi.org/10.1145/2398856.2364568` (cit. on p. 131).

Whitehead, Alfred North and Bertrand Russell (1910). *Principia Mathematica*. Vol. 1. Cambridge University Press (cit. on p. 11).

Zeilberger, Noam (2008a). “Focusing and higher-order abstract syntax”. In: *ACM SIGPLAN Notices* 43.1, pp. 359–369 (cit. on p. 99).

— (2008b). “On the Unity of Duality”. In: *Annals of Pure and Applied Logic* 153.1-3, pp. 66–96. DOI: `https://doi.org/10.1016/j.apal.2008.01.001` (cit. on pp. 77, 78, 80, 84, 99, 129).

— (2009). “The Logical Basis of Evaluation Order and Pattern-Matching”. PhD thesis. USA: Carnegie Mellon University. ISBN: 9781109163018 (cit. on pp. 6, 109, 128).

Index